Use of Accessible Information to Improve Industrial Security Testing

Zur Erlangung des akademischen Grades einer

Doktorin der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

M.Sc.

Anne Borcherding

aus Karlsruhe, Deutschland

Tag der mündlichen Prüfung:	18.11.2024
Erster Gutachter:	Prof. Dr.-Ing. habil. Jürgen Beyerer
Zweiter Gutachter:	Prof. Dr. Eric Bodden

Abstract

Since the discovery of the Stuxnet cyber attack that damaged several nuclear centrifuges in 2010, it has become evident that industrial control and automation systems are increasingly being targeted. Most of these attacks are executed via the networks that connect Operational Technology (OT) components, such as controllers and actuators. Since OT components are widely deployed and serve as an interface between the network and the physical world, attacks on them can cause damage to production assets and harm humans. In addition, the accessibility of OT components has increased with the adoption of standard internet communication protocols, reducing the need for specialized domain knowledge. Therefore, OT components require particularly strong security measures, especially with respect to their network interfaces.

To secure an OT component, security needs to be considered throughout the entire development lifecycle, including system-level security testing of the OT component. In practice, a tester may not have control over all parts of an OT component, as they are often assembled from multiple third-party components. Consequently, information internal to the OT component may be inaccessible, making *blackbox* testing—where no internal knowledge is assumed—essential.

Industrial blackbox testing employs various testing techniques to assess the interfaces of an OT component. Prominent approaches include Web Vulnerability Scanners (WVSs) and network protocol fuzzers. Similar to other testing techniques, commercial and academic WVSs are primarily designed for general-purpose Information Technology (IT) systems, which typically have extensive resources that allow them to efficiently handle heavy workloads. However, as WVSs are increasingly applied to web applications running on

resource-constrained OT components, they exhibit several shortcomings. For instance, WVSs are often unable to handle crashes caused by overloading the OT component.

Network protocol fuzzing follows the general approach of sending potentially malformed inputs to a System under Test (SuT) via its network interfaces, while monitoring its behavior for crashes and anomalies. In contrast to existing graybox approaches, the majority of blackbox fuzzing approaches do not incorporate observations of the SuT's behavior into their test case generation, making blackbox testing generally less effective.

This dissertation advances the field of blackbox security testing for OT components by presenting and evaluating novel approaches that leverage information accessible before and during blackbox testing to enhance testing performance. For instance, it leverages information on known vulnerabilities and generated network traffic. The contributions of this dissertation are referred to by their names, such as `HitM`, in the following. Key contributions include consolidating knowledge from past tests to improve future blackbox fuzzing with `ClusterCrash`, and utilizing network traffic or crashing services to advance test case generation with `Palpebratum` and `Smevolution`. As a basis, `NeDaP` introduces and assesses methods for preprocessing network traffic. `HitM` successfully leverages information on the OT component's behavior as expressed by network responses and crashing services to transparently improve the performance of existing WVSs. `SWaTEval` presents a framework for stateful web application testing, utilizing network responses to gain information on the internal state machine of the OT component. Additionally, this work contributes to graybox fuzzing with `StateBandit`, which leverages graybox information to apply a reinforcement learning agent to stateful network fuzzing, and `MEMA`, which examines the impact of performance metrics on fuzzer evaluations.

Through these contributions, this dissertation demonstrates how the limited information that is accessible before and during blackbox testing can be leveraged to improve test performance, especially by applying graybox testing techniques to a blackbox setting. The work resulted in the discovery and disclosure of eight critical vulnerabilities in OT components.

Kurzfassung

Seit der Entdeckung des Cyberangriffs Stuxnet, der 2010 mehrere Gaszentrifugen beschädigte, wird deutlich, dass industrielle Steuerungs- und Automatisierungssysteme zunehmend Ziel komplexer Angriffe sind. Die meisten dieser Angriffe erfolgen über Netzwerke, die Komponenten der Betriebstechnik (engl. *Operational Technology* (OT)) wie Steuerungen und Aktoren miteinander verbinden. Da OT-Komponenten weit verbreitet sind und als Schnittstelle zwischen dem Netzwerk und der physischen Welt dienen, können Angriffe auf sie Schäden an Produktionsanlagen verursachen und Menschen verletzen. Darüber hinaus sind OT-Komponenten durch den zunehmenden Einsatz von Standard-Internetprotokollen leichter zugänglich geworden, da weniger domänenspezifisches Wissen vorausgesetzt ist. Daher erfordern OT-Komponenten besondere Sicherheitsmaßnahmen, insbesondere im Hinblick auf ihre Netzwerkschnittstellen.

Um eine OT-Komponente zu schützen, muss ihre Sicherheit während des gesamten Entwicklungslebenszyklus berücksichtigt werden, was insbesondere Schwachstellentests auf Systemebene einschließt. In der Praxis ist es jedoch oft nicht möglich, dass während eines solchen Tests alle Bestandteile einer OT-Komponente kontrolliert werden können, da diese oft aus Komponenten von Drittanbietern zusammengesetzt sind. Folglich können interne Informationen der OT-Komponente unzugänglich sein, was *Blackbox*-Tests – bei denen kein internes Wissen vorausgesetzt wird – unerlässlich macht.

Beim industriellen Blackbox-Testen werden verschiedene Techniken eingesetzt um die verschiedenen Schnittstellen einer OT-Komponente zu testen. Gängige Ansätze sind Web-Verwundbarkeitsscanner (engl. *Web Vulnerability Scanners* (WVSs)) und Netzwerkprotokoll-Fuzzer. Analog zu anderen Ansätzen sind kommerzielle und akademische WVS in der Regel für Systeme der

Informationstechnik (engl. *Information Technology* (IT)) konzipiert, die über umfangreiche Ressourcen verfügen und daher hohe Lasten effizient bewältigen können. Da WVS jedoch zunehmend für Webanwendungen eingesetzt werden, die lokal auf ressourcenbeschränkten OT-Komponenten laufen, werden verschiedene Mängel deutlich. Beispielsweise reagieren WVS oft nicht auf Abstürze einer OT-Komponente, die durch Überlastung verursacht werden.

Netzwerkprotokoll-Fuzzing folgt dem allgemeinen Ansatz, potenziell fehlerhafte Eingaben über die Netzwerkschnittstelle an ein zu testendes System (engl. *System under Test* (SuT)) zu senden und gleichzeitig dessen Verhalten auf Abstürze und Anomalien zu überwachen. Im Gegensatz zu Ansätzen des Graybox-Fuzzing, werden bei den meisten Ansätzen des Blackbox-Fuzzing Beobachtungen des Verhaltens des SuTs nicht in die Testfallgenerierung einbezogen. Das führt dazu, dass Blackbox-Tests im Allgemeinen weniger effektiv sind.

Diese Dissertation leistet einen Beitrag zum Gebiet des Sicherheitstestens von OT-Komponenten, indem sie neue Ansätze vorstellt und evaluiert, die Informationen nutzen, die vor und während des Blackbox-Testens zugänglich sind, um die Testleistung zu verbessern. Solche nutzbaren Informationen sind zum Beispiel bekannte Schwachstellen oder der Netzwerkverkehr, der während eines Tests erzeugt wird. Die Beiträge dieser Dissertation werden im Folgenden mit ihren Namen, wie zum Beispiel `HitM`, bezeichnet.

Zu den wichtigsten Beiträgen zählt die Zusammenführung von Erkenntnissen zur Verbesserung zukünftiger Blackbox-Tests, wie es mit `ClusterCrash` vorgestellt wird. Ein weiterer Beitrag besteht aus der Nutzung von Informationen über das Verhalten der OT-Komponente, das sich im Netzwerkverkehr oder in abstürzenden Diensten äußert. Diese Informationen werden von `Palpebratum` und `Smevolution` genutzt, um die Testfallgenerierung zu verbessern. Als Grundlage hierfür führt `NeDaP` Methoden zur Vorverarbeitung des Netzwerkverkehrs ein. `HitM` hingegen nutzt Informationen über das Verhalten der OT-Komponente, in Form von Kommunikationspaketen und abstürzenden Diensten, um die Leistung bestehender WVS transparent zu verbessern. `SWaTEval` ist ein Framework zum Testen von zustandsbehafteten Webanwendungen, das Kommunikationspakete der OT-Komponente verwendet, um Informationen über die internen Zustände der OT-Komponente zu erhalten.

Darüber hinaus leistet diese Dissertation mit `StateBandit` und MEMA einen Beitrag zum Graybox-Fuzzing. `StateBandit` verwendet Informationen, die nur in einem Graybox-Test verfügbar sind, um einen Reinforcement-Learning-Agenten für Netzwerk-Fuzzing einzusetzen, während MEMA den Einfluss von Metriken auf Fuzzer-Evaluationen untersucht.

Durch diese Beiträge zeigt die vorliegende Dissertation, wie die begrenzten Informationen, die vor und während eines Blackbox-Tests verfügbar sind, genutzt werden können, um die Testleistung zu verbessern, insbesondere durch den Einsatz von fortgeschrittenen Techniken aus dem Graybox-Testen. Die vorliegende Arbeit führte zur Entdeckung und Veröffentlichung von acht kritischen Schwachstellen in OT-Komponenten.

Contents

1 Introduction

Cyber attacks such as `Stuxnet` and `Triton` showcase that critical infrastructure and industrial automation and control systems are being targeted. While `Triton` was detected in 2017 before any serious harm could be done [DiP18], `Stuxnet` damaged several nuclear centrifuges in 2010 [Bae17]. For these attacks to be successful, the Operational Technology (OT) components used have to be vulnerable. Anton et al. analyzed six types of Programmable Logic Controllers (PLCs) and found that 4,822 PLCs of these types were accessible via the internet and that the majority of these PLCs are vulnerable to at least one vulnerability [Ant21]. In addition, the industrial cybersecurity company Dragos showed in their recent report that 80% of the vulnerabilities they analyzed were vulnerabilities of low level OT components, including PLCs, sensors, and industrial controllers [Dra23a].

One driver for an increased vulnerability of OT components is the advancing connectivity in production plants. With higher connectivity, more possible entry points are made available for external attackers and thus the attack surface increases [Ant21]. The German Federal Office for Information Security reports an increasing number of vulnerabilities that are concerned with the connectivity of the OT components, such as intrusion via remote maintenance access or infection with malware via internet and intranet [Fed22]. According to Dragos, the number of findings with respect to external connections or network segmentation decreased in 2022, but these findings are nevertheless relevant in a significant portion of the analyses [Dra22]. In 2023, 16% of the vulnerabilities analyzed by Dragos were exploitable over the network [Dra23a].

In addition, communication protocols and hardware are becoming more standardized and more parts of an OT component, such as network stacks, are shared by multiple manufacturers [Dou23]. While this development is beneficial for interoperability and interchangeability, it also reduces the domain knowledge required to design and execute cyber attacks with a broad target scope. For example, the vulnerability group Ripple20 that includes 19 vulnerabilities of one specific network stack, affected a plethora of manufacturers and devices since this stack is used very commonly. According to the researchers who published this vulnerability group, 31 manufacturers confirmed that their devices are affected by the vulnerabilities as of 25th October 2020 [Koh20a]. A research group at Forescout analyzed seven open source network stacks and found 33 vulnerabilities in four of those stacks [San21c]. Furthermore, they analyzed the use of these affected stacks and came to the conclusion that millions of devices are potentially affected by the published vulnerabilities. Most of these potentially affected devices are used in Information Technology (IT) or Internet of Things (IoT) environments, but the authors also classify 19% of the devices as OT components. This again shows how cyber attacks can be used to target several types of devices, spanning a wide range of application domains. It shows that the convergence of IT and OT leads to more vulnerabilities in OT components and thus increases the possibility for a successful cyber attack [Haj21].

One approach to decrease the possibility of a successful attack is to reduce the number of vulnerabilities in an OT component. To this end, there are several approaches and starting points which are incorporated in the secure development lifecycle as specified by the international standard IEC 62443 [Int19]. The secure development lifecycle requires that the security of the developed OT component is considered during the whole lifecycle. This includes avoiding design vulnerabilities by security considerations during the design phase, and avoiding implementation vulnerabilities by following secure coding practices. Furthermore, one important part is to conduct tests to verify and validate the security of an implemented OT component.

For a tester, who has the task to automatically test an OT component for vulnerabilities and other security related issues, several challenges arise that are specific to the industrial setting. Of these challenges, three are the most prominent. A tester (1) needs to perform *blackbox* security tests, (2) needs to consider the special requirements of the industrial setting such as proprietary communication protocols, and (3) needs to choose a Test Tool (TT) which finds as many vulnerabilities as possible.

1.1 Outline

The overall objective of this doctoral work is to make blackbox security testing of OT components more effective and efficient while considering the special requirements of an industrial setting. In the following section, the motivation for this objective as well as the three challenges of a tester mentioned above will be detailed. Subsequently, we will discuss the research questions that drive this doctoral work in Section 1.3, present the contributions in Section 1.4, clarify the scope in Section 1.5, and introduce a running example OT component used throughout this doctoral work in Section 1.6.

Following on the introduction presented in this chapter, Chapter 2 provides background on industrial security testing and Machine Learning (ML). This includes a general overview of industrial security (Section 2.1), and a presentation of the methods employed in testing for security in industrial use cases (Section 2.2). Moreover, details on the two testing techniques utilized in this doctoral work, namely Web Vulnerability Scanners (WVSs) (Section 2.3) and fuzzing (Section 2.4), are given. To establish the foundation for the ML-based approaches proposed in this doctoral work, Section 2.6 introduces the relevant concepts. Once this foundational background is set, Chapter 3 analyzes the information sources that are accessible in a blackbox security test.

Chapters 4 to 8 elaborate on the contributions of this doctoral work. Each of these chapters provides in-depth information on the specific problem statement, related work, the approach(es), and the evaluation. Chapter 4 presents

the contribution with respect to blackbox web security testing, HitM; Chapter 5 provides more details on the usage of public vulnerability information, ClusterCrash; Chapter 6 describes the analysis regarding network packet data preprocessing, NeDaP; Chapter 7 is concerned with the ML-based approaches to leverage information accessible in blackbox security testing; and Chapter 8 details the approaches to evaluating TT.

Following these detailed descriptions of the subtopics of this doctoral work, Chapter 9 consolidates these subtopics, and discusses the contributions and results with respect to the main research questions formulated in Section 1.3. Possible future research directions are examined in Chapter 10, and Chapter 11 summarizes this dissertation.

1.2 Challenges of Industrial Security Testing

To motivate the objectives of this doctoral work, the following paragraphs describe the key challenges of industrial security testing for OT components.

Blackbox Test Setting

In an ideal world, the tester of an OT component has full access to the OT component, including the source code and accompanying materials. With all the information given, the tester could perform efficient whitebox tests to find vulnerabilities of the OT component.

However, in real test settings, the tester does not have access to or control over all the components which are included in the OT component [Dou23]. Doumanidis et al. analyzed 48 firmwares of different PLCs from the four manufacturers ABB, Schneider Electric, Siemens, and WAGO [Dou23]. The authors dissected the firmwares to find out which third-party components are used in the firmwares and which of the components are shared between different firmwares of the same manufacturer and between different manufacturers. One of their findings is that the two firmwares of WAGO PLCs that where analyzed contain 302 and 343 third-party libraries, respectively. A Siemens

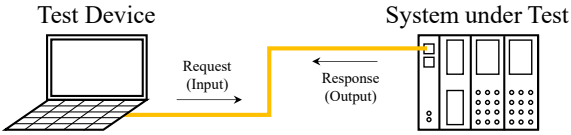


Figure 1.1: Basic setup of a blackbox security test via the Ethernet interface. The Test Device (TD) uses a Testing Tool (TT) to generate test cases, send these to the System under Test (SuT), and observe the SuT's behavior in response to this input. For example, such a test case or input could consist of a mutated File Transfer Protocol (FTP) packet. Then, the observed behavior of the SuT could be the corresponding FTP response.

PLC firmware included 720 third-party libraries. To thoroughly test one of these PLCs, a tester would have to test all of those libraries as well. For the closed source third-party libraries included in a PLC, a tester has no possibility to get enough information to perform a whitebox test. Thus, the tester needs to perform blackbox tests of the OT component. Moreover, blackbox tests are, for example, required by the international standard IEC 62443-4-2, which defines requirements for securing OT components [Int19].

In general, blackbox tests of OT components consist of blackbox tests conducted via the network interfaces provided by the System under Test (SuT). Figure 1.1 shows a high level overview of the general testing approach, aiming to set some of the terms used in the domain of industrial security testing. The OT component that is to be tested during the security test is referred to as the SuT, as opposed to the Test Device (TD), which is used to conduct the test by running a Testing Tool (TT). The TD is usually an industrial computer or a usual desktop computer. During testing, the TT sends some kind of input to the SuT. For example, this input could be a network packet or a sequence of network packets. Usually, the SuT reacts in some way to that input, for example by sending a response via the network. Now, the TT can analyze this response as well as the general behavior of the SuT in order to find anomalies in the SuT's behavior and to derive whether a vulnerability of the SuT was discovered. As a basic approach to analyzing the SuT's general behavior, the TT can send an Internet Control Message Protocol (ICMP) ping network packet to the SuT to check whether it is still responsive. If the SuT no longer responds to such a packet, the TT can assume that the network stack of the SuT has crashed, which is a strong indicator for a discovered vulnerability.

One of the testing approaches this doctoral work focuses on is *fuzzing* (see also Section 2.4). Fuzzing generally follows the approach to present inputs to the SuT that are possibly semantically or syntactically malformed [Man19]. One basic necessity of an efficient and effective fuzzing test is an optimization function to guide the test case generation. Generally speaking, this optimization function takes the behavior of the SuT with respect to one input and decides whether this input is interesting or not. An input, or test case, is deemed interesting, if it leads to behavior that is different from those of previous test cases. For further testing, usually only the interesting test cases are considered as a basis for new test cases. With this, the exploration of the SuT's behavior is aspired in order to discover as many anomalies and vulnerabilities as possible. To be more concrete, in whitebox fuzzing, the behavior of the SuT could be represented by statement coverage or branch coverage [Cer23]. In graybox fuzzing, in which the fuzzer usually has access to a binary of the SuT, usually the code coverage is used as a representation of the SuT's behavior [Böh21]. During blackbox fuzzing, only the behavior the SuT exhibits via its communication interfaces can be analyzed and used during the test. Even though the SuT exhibits some behavior to the outside, blackbox fuzzers usually either use only information on crashes to guide the test case generation or do not take any feedback into account [Man19].

This doctoral work aims to utilize more of the information accessible in blackbox testing to guide the fuzzing and thus to make the testing more effective. For this, we present several approaches which are, e.g., based on analyses of known vulnerabilities (Chapter 5), transparent injection of additional context information (Chapter 4), or automatic model inference (Chapter 7). See Section 1.4 for an overview of the contributions of this doctoral work, and Chapter 3 for an analysis of information accessible in blackbox testing.

Industrial Test Setting

The second challenge is based on the observation that TTs designed for traditional IT security testing might not fulfill the requirements of OT component testing. In the following, we will discuss two main requirements that distinguish testing IT systems from testing OT components.

The first requirement of industrial security tests that needs to be considered is concerned with the different interfaces an industrial SuT provides. An IT TT might not be able to observe all the outputs from an OT component. However, this is needed to thoroughly analyze the SuT's behavior [*Bor20*]. A TT has the general goal to find those states of a SuT in which the SuT does not behave as intended, i.e. in which it shows anomalous behavior. In blackbox testing, the TT is limited to observing the external behavior exhibited by the SuT. Anomalous behavior might, for example, be a different response behavior, including different response messages, additional response delay, or no response at all. The latter is usually a strong indicator of a crash. All interfaces of an OT component should be monitored for anomalous behavior to be able to decide whether a vulnerability has been discovered. This is especially true for the digital I/O interfaces that some OT components have.

For example, a bus coupler tested during this doctoral work (see Section 1.6) exhibits a vulnerability that can only be detected by observing the digital I/O interfaces during testing. This vulnerability can be exploited with a malformed Hypertext Transfer Protocol (HTTP) packet. When such a packet is sent to the bus coupler, none of the Ethernet-based interfaces shows a strongly anomalous behavior. Nevertheless, this packet leads to a crash of the digital I/O interface. For TTs that only observe the Ethernet-based interfaces, this vulnerability is not apparent. Upon discovery, this vulnerability was responsibly disclosed and published in cooperation with the manufacturer[1]. See Section 4.4 for a more detailed description.

[1] As a result, the vulnerability was assigned a unique Common Vulnerabilities and Exposures (CVE) identifier: CVE-2018-16994 [CVE18]

The second requirement that needs to be considered is concerned with the communication protocols that are used by industrial SuTs. An IT TT may not support the proprietary communication protocols used by the OT component, as those are not used in traditional IT environments [Kna14]. Nevertheless, a monitoring and observation of these protocols might give interesting insights into the behavior and the current state of the SuT.

The contributions of this doctoral work utilize two different approaches to address this challenge. Some concepts utilize specialized tools like the industrial security testing framework ISuTest® [Pfr17][*Pfr23*], which supports industrial communication protocols such as PROFINET. This framework was developed at Fraunhofer IOSB, and was used and further improved by this doctoral work (see Sections 4.3 and 7.4). Other contributions employ protocol independent approaches such as the behavior approximation using Hidden Markov Models (HMMs) presented in Section 7.5.

Choice of Testing Tool

The third challenge that arises for a tester of an OT component is the choice of TTs to use for the tests. Especially regarding TTs targeting standard internet communication protocols, a tester has to choose from a plethora of existing TTs. This presents a challenging choice, especially when using the TTs in an industrial setting, as shown by Pfrang et al. [*Pfr19b*]. The authors run five blackbox WVSs against seven OT components which provide locally running web servers. Blackbox WVSs are TTs which actively analyze Web Applications (WAs) by sending inputs to the application and analyzing the output, aiming to reveal, for example, Cross-site Scripting or SQL Injection vulnerabilities (see also Section 2.3). Pfrang et al. analyze the performance of the WVSs and come to the conclusion that each scanner finds at least one vulnerability that all other scanners miss [Pfr19b]. In support of this, Poncelet et al. evaluate several fuzzers against a network stack, showing that none of the evaluated fuzzers is uniformly better than the others [Pon22].

A conclusion is that a tester would need to employ all the scanners to find as many different vulnerabilities as possible. Additionally, any new approaches aiming to improve blackbox testing in general would need to be incorporated by each of the scanners. Driven by these insights, this doctoral work proposes approaches that improve the performance of TTs in a way that is transparent to the TTs or the general testing approach. This holds, for example, for the proxy-based approach to improve blackbox WVSs presented in Section 4.3 as well as for the behavior approximation using HMMs. The latter approach allows to apply existing graybox fuzzing approaches to blackbox test settings by transparently providing the information that would otherwise be missing in a blackbox test.

1.3 Problem Statement and Research Questions

As detailed above, a tester of OT components faces the following five main challenges.

Challenge 1 (Blackbox Testing)**.** *Blackbox testing needs to be performed caused by restricted access to parts of the SuT and by requirements from standards.*

Challenge 2 (Missing Information)**.** *In blackbox testing, information relevant to perform efficient and effective tests is missing.*

Challenge 3 (Insufficient Observations)**.** *Traditional TTs might not be able to observe all communication interfaces of the SuT, which is necessary to detect as many anomalies as possible.*

Challenge 4 (Insufficient Protocol Support)**.** *Traditional TTs usually do not support industrial communication protocols.*

Challenge 5 (Choice of Testing Tool)**.** *Several TTs need to be used in conjunction to achieve high vulnerability coverage.*

This dissertation proposes several approaches to utilize the limited information that is available in blackbox testing. As underlying requirement, these approaches consider the special requirements of OT components as presented above, aim to be transparent to the underlying TTs if used and to be independent of the concrete communication protocol. However, some of the approaches are specifically crafted to be used in the domain of WAs running on OT components and thus focus on HTTP as underlying communication protocol (Sections 4.3 and 8.3).

This doctoral work is driven by the following research questions. Furthermore, based on these main research questions, more specific research questions are formulated for the subtopics of this doctoral work. These specific research questions are presented and discussed in the corresponding chapters (see Sections 4.1, 5.1, 6.1, 7.1, and 8.1). A discussion of the general research questions, which consolidates the results of the subtopics, is presented in Chapter 9.

Research Question 1. *What sources of information are available for blackbox security testing?*

As stated, blackbox testing assumes that the tester has no access to internal information of the SuT, and thus is restricted to the information that the SuT exposes. We analyze which information sources are utilized by approaches from literature (see Chapter 3) and find that those information sources can be categorized along two dimensions: (1) time of accessibility, and (2) temporal variability. We differentiate between information sources that are accessible prior to the test, such as information on past vulnerabilities, and information sources that are accessible only during testing, such as the SuT's response to certain test cases. In addition, we differentiate between information that remains constant over time, such as the communication endpoints of a SuT, and information that changes over time, such as the network traffic observed during testing.

Research Question 2. *How can information that is accessible prior to the actual test be used to improve a blackbox security test?*

Some information sources are accessible prior to the test, and this doctoral work explores three approaches on how to utilize this information. First, we leverage information on previously published vulnerabilities. We analyze how this information can be structured with the objective to derive universal insights on which test cases to use during testing (see Chapter 5). Bases on these insights, we implement blackbox tests that lead to eleven findings related to the security of OT components (see Chapter 5). In addition, we utilize information on the communication endpoints of a SuT to transparently improve the performance of blackbox WVS in Chapter 4, and network traffic accessible before the test to train a model of the SuT in Section 7.5.

Research Question 3. *How can models of the SuT be derived from information sources during a blackbox security test?*

In contrast to the information sources accessible prior to the test, some information is only exposed during the test. This includes the behavior that the SuT exhibits in response to given inputs. One of the approaches to use these information sources during the test is to derive a model of the SuT which is then directly used to guide and evaluate the testing process. We approach Research Question 3 by analyzing several means to derive a model from the different information sources and by evaluating their impact on the testing performance. This includes training ML models on test data information, training an HMM on network traffic data, and training a Reinforcement Learning (RL) agent on graybox coverage information (see Chapter 7). Our experiments demonstrate that the performance of blackbox testing can indeed be improved by some of the models, showing that the models have the potential to successfully learn and represent information on the SuT.

Research Question 4. *How can TTs be evaluated and compared to competing approaches?*

Qualitative and quantitative evaluations are necessary to set newly proposed approaches into context and to show their capabilities in comparison to the current state-of-the-art. This doctoral work focuses on blackbox Stateful Web

Application Testing (SWAT) and stateless graybox fuzzing by (1) presenting an evaluation framework for SWAT in Section 8.3 and (2) showing that the choice of performance metrics has an impact on the relative assessment of graybox fuzzers in Section 8.4.

1.4 Contributions

In order to address the previously mentioned research questions, while also considering the challenges of industrial security testing, this doctoral work makes several contributions. An introduction to these contributions is given below, whereas detailed explanations of the contributions, including a detailed problem statement, the corresponding research questions, and information on related work, are provided in the corresponding chapters (Chapters 4 to 8). Thereafter, Chapter 9 summarizes the different subtopics and approaches of this doctoral work and links the contributions to the general research questions formulated above. Table 1.1 gives an overview on the contributions, their link to the challenges of OT component security testing, and lists the chapters in which more details are given on the respective contribution.

The publications produced over the course of this doctoral work are assigned to the corresponding contributions in the paragraphs below and, like the theses supervised during this doctoral work, are referenced in italics to make it easier for the reader to distinguish between own work and that of others.

Industrial Web Security

The first contribution is concerned with improving industrial security testing by facilitating efficient web security testing for industrial use cases. To that end, we first analyze which limitations WVSs show with respect to their applicability to OT component testing.

Contribution 1. *Analysis of the limitations of blackbox WVSs with respect to their applicability to OT component security testing.*

Based upon this analysis, we propose HitM, providing a proxy-based solution that modifies the input and output of existing blackbox WVSs, and adds additional observation capabilities [*Bor20*]. All these modifications are transparent to the underlying WVS, which makes it easy to apply HitM to arbitrary blackbox WVS. For our evaluation of HitM, we run six WVSs against five OT components. Then, we run the same configurations again, but enable the additional features provided by HitM. We show that the additional features lead to more true positive vulnerability reports and an increased Uniform Resource Locator (URL) coverage. These advantages come at the cost of increasing the time needed per test case, since each test case needs to be parsed and processed by the proxy before being forwarded to the SuT or TD, respectively. In addition, we run HitM against the bus coupler used as a running example for this doctoral work (see Section 1.6), revealing one previously unknown vulnerability that was only revealed by the features of HitM.

Contribution 2. *Implementation and evaluation of the proxy-based solution HitM which transparently facilitates the use of blackbox WVS in OT component security testing.*

HitM is designed for a blackbox test setting (Challenge 1 (Blackbox Testing)) and allows for observing all communication interfaces of the SuT (Challenge 3 (Insufficient Observations)). Due to its proxy-based design, HitM is transparent for the used WVS and such can be used together with arbitrary WVSs (Challenge 5 (Choice of Testing Tool)). Amongst other features, HitM injects additional information into the requests sent by the WVS. This information, for example, includes additional URLs taken from the sitemap of the WA (Challenge 2 (Missing Information)).

See Chapter 4 for details on HitM, its evaluation, and an in-depth discussion of the results. The publication of HitM builds the foundation of Chapter 4 [*Bor20*].

Table 1.1: Overview of the contributions of this doctoral work, including their link to the challenges that a tester of an OT component has and the research questions. The approaches are discussed in more detail in the chapters referred to in the table. Most of the contributions have been published, which is indicated by the corresponding reference. The abbreviation *tbp* indicates that the respective contribution is going to be published in the future.

Name	Description	Questions				Challenges					Chapt.	Ref.
		1	2	3	4	1	2	3	4	5		
`HitM`	Proxy-based solution to improve the performance of WVSs	✓	✓	✓		✓	✓	✓		✓	4	[*Bor20*]
`ClusterCrash`	Use of information on known vulnerabilities to improve future tests	✓	✓			✓	✓	✓			5	[*Bor22*]
`NeDaP`	Preprocessing of network packet data	✓		✓		✓			✓		6	tbp
`Smevolution`	Use of ML models to improve blackbox fuzzer guidance	✓		✓	✓	✓	✓	✓	✓	✓	7	[*Bor23b*]
`Palpebratum`	Use of HMMs to approximate the SuT's behavior	✓	✓	✓		✓	✓		✓	✓	7	tbp
`StateBandit`	Use of an RL agent for stateful network fuzzing	✓		✓			✓		✓		7	[*Bor23a*]
`SWaTEval`	Evaluation framework for SWAT			✓	✓		✓		✓		8	[*Bor23c*]
`MEMA`	Impact analysis for performance metrics in fuzzing evaluations			✓	✓		✓		✓		8	-

Blackbox Network Fuzzing

The second contribution, `ClusterCrash`, aims to use publicly available vulnerability information to improve future blackbox security tests. For this, we focus on communication protocol implementations used by OT components. As mentioned above, OT components are increasingly using protocols building upon standard internet protocols such as Internet Protocol (IP) and Transmission Control Protocol (TCP). This includes, for example, communication protocols such as HTTP, OPC Unified Architecture (OPC UA), and FTP (see Section 2.1.2). Thus, the security and reliability of TCP/IP network stacks built into an OT component is crucial and they need to be tested using blackbox tests. Literature shows several extensive graybox and whitebox tests of TCP/IP network stacks [Koh20a, Koh20b, San21c, Ser19], which we leverage to help improving future blackbox tests of OT component. We analyze the findings of these studies, cluster them by their root causes, and formulate Vulnerability Anti-Patterns (VAPs), which give details on the root causes of vulnerabilities in TCP/IP stacks.

Contribution 3. *Analysis and clustering of published TCP/IP stack vulnerabilities, and derivation of VAPs which formalize the underlying causes of these vulnerabilities.*

Then, we use this newly acquired knowledge to design and implement new blackbox tests which specifically test for the VAPs. To evaluate the performance of these new tests, we execute them against eight OT components of five different device classes. This evaluation shows that the tests are indeed able to reveal previously unknown vulnerabilities. Moreover, we show that vulnerabilities rising from one VAP can be found in different communication protocols and different device classes. We communicated the discovered vulnerabilities to the corresponding manufacturers, resulting in three publicly confirmed vulnerabilities [CVE21c, CVE21a, CVE21b].

Contribution 4. *Utilization of the newly developed VAPs to implement new tests, and an evaluation of those tests which revealed three confirmed previously unknown vulnerabilities of OT components.*

In summary, `ClusterCrash` shows how publicly available information on communication stack vulnerabilities can be leveraged to improve future blackbox tests (Challenge 1 (Blackbox Testing), Challenge 2 (Missing Information)). Furthermore, the resulting tests were incorporated to ISuTest® (see Section 2.2.3) and thus all communication interfaces of the SuT can be observed during testing (Challenge 3 (Insufficient Observations)).

Chapter 5 presents more details on `ClusterCrash`, building upon the corresponding publication [*Bor22*].

Network Data Processing

Another data source that can be exploited during blackbox testing is the network traffic generated by the SuT and the TT. We aim to make this information available for further usage by employing ML models, for which we need to preprocess the raw network packets first. With `NeDaP`, we analyze and evaluate three different approaches to network packet preprocessing with respect to the goal to perform model-based testing on the preprocessed data.

We base the preprocessing pipeline on two approaches from literature [Chi20, Lot20], and include three different approaches for dimensionality reduction: a Principal Component Analysis (PCA), an Autoencoder (AE), and a Convolutional Autoencoder (CAE). Our experiments show that the CAE performs better with respect to in-domain generalization, while the AE shows a better performance for out-of-domain generalization. For the use case of HMM-based fuzzing, the out-of-domain generalization is expected to be more important, since the models will be trained on user data, but then will be used on fuzzing data. This evaluation thus gives insights valuable for the further usage of the dimensionality reduction approaches, especially by `Palpebratum`, an approach presented in the next paragraph.

Contribution 5. *Analysis and evaluation of three approaches to network packet preprocessing with the goal of model-based security testing.*

The proposed preprocessing pipeline is designed for blackbox testing (Challenge 1 (Blackbox Testing)) and is independent of the underlying network protocol (Challenge 4 (Insufficient Protocol Support)).

Chapter 6 provides more details on our experiments with respect to network packet preprocessing.

Machine Learning based Blackbox Fuzzing

This doctoral work proposes three approaches to model-based testing, especially applying ML-based models to a blackbox test setting. These models exploit accessible information to improve the performance of blackbox testing of OT components.

The first approach, `Smevolution`, is concerned with blackbox network fuzzing and leverages information on test cases that have already been sent, and the SuT's reaction to those test cases [*Bor23b*]. `Smevolution` uses this information to train an ML model that is updated and refined during the test. This model is combined with an evolutionary testing algorithm and is used to improve the mutation strategy and the selection of new test cases. For our evaluation, we select three different ML models (namely Support Vector Machine (SVM), Decision Tree (DT), and Neural Network (NN)) of which each has distinct features that are beneficial for the testing process. Moreover, we integrate `Smevolution` to the security testing framework ISuTest® (see Section 2.2.3) and execute against an artificial OT component which incorporates known vulnerabilities. The main result of this evaluation is that using `Smevolution` leads to more crashes compared to a baseline employing a random approach. This indicates that the models are indeed capable of learning and representing information that can be used to improve blackbox testing.

Contribution 6. *Proposal, implementation, and evaluation of* `Smevolution`*, an approach to combine evolutionary fuzzing with ML models leveraging test data to help guiding the test process.*

Smevolution is suited for a blackbox test setting (Challenge 1 (Blackbox Testing)) and is independent of the tested network protocol (Challenge 4 (Insufficient Protocol Support)). The ML models analyze and process information on past test cases to provide relevant information for test case generation (Challenge 2 (Missing Information)), and could be used as external mutator by mutation-based fuzzers (Challenge 5 (Choice of Testing Tool)). Due to the integration in ISuTest®, Smevolution is also able to observe all communication interfaces of the SuT (Challenge 3 (Insufficient Observations)).

Details on Smevolution and the corresponding evaluation are presented in Section 7.4. These explanations are based on the publication of Smevolution [*Bor23b*].

The second approach, Palpebratum, uses the results from the aforementioned network packet preprocessing, NeDaP, and uses the preprocessed network packets to train an HMM which approximates the behavior of the SuT. One of the features of an HMM is that one can calculate the path through the model's states with the highest probability of leading to a given observation. We use this path information as a representation of the SuT's behavior and as a substitute for code coverage used in graybox testing. With this approximated information, we can facilitate the application of graybox fuzzing approaches in a blackbox test setting. To evaluate this approach, we integrate Palpebratum in the fuzzing library LibAFL [Fio22], and analyze how the performance of the HMM-based fuzzer compares to a random fuzzer and a blackbox fuzzer.

Our experiments show that the HMM-based fuzzers generate test cases that are more efficient that those generated by the baseline fuzzers. In this case, efficiency is measured as the average coverage that is achieved by a test case generated by the respective fuzzer. However, the final coverage achieved by the baseline fuzzers significantly outperforms the coverage achieved by the HMM-based fuzzers. Possible reasons for this is an underestimation of the coverage achieved by the HMM-based fuzzers, and the overhead introduced by the HMM. This overhead leads to a reduced number of test cases that are generated in a fixed time frame and thus the HMM-based fuzzers are likely to sent less test cases to the SuT than the baseline fuzzers. In addition, it shows that the fuzzer that uses the AE for preprocessing performs significantly

better than the fuzzer that uses the CAE. This supports the expectation that the out-of-domain generalization of the AE, as demonstrated with NeDaP, is beneficial for fuzzing.

Contribution 7. *Proposal, implementation, and evaluation of* `Palpebratum`*, an approach to approximate the SuT's behavior based on preprocessed network traffic, facilitating graybox fuzzing approaches in a blackbox test setting.*

`Palpebratum` is designed to be used in a blackbox test setting (Challenge 1 (Blackbox Testing)) and is independent of the underlying network protocols (Challenge 4 (Insufficient Protocol Support)). The HMM is used to approximate information that would otherwise be missing in a blackbox test scenario (Challenge 2 (Missing Information)). This approximated information can be used by graybox coverage-based fuzzers (Challenge 5 (Choice of Testing Tool)).

Section 7.5 details this approach and the evaluation that was conducted to assess its performance.

The third approach, `StateBandit`, is concerned with the state selection problem of stateful network protocol fuzzing. Stateful fuzzing recognizes that the network protocols typically behave in a stateful manner and explicitly decides which state of the SuT to test in the next fuzzing iteration [Ba22]. One challenge that arises is to choose the next state in a way that the fuzzing is efficient and effective (see also Section 2.5). We propose to approach this challenge by modeling the state selection as a RL problem, more specifically as a Multi-armed Bandit (MaB) problem. To this end, we deploy an agent which has the task to select .the state to be used in the next fuzzing iteration. In order to be able to give the agent some more information to base this selection on, we first deploy this approach in a graybox test setting. The agent receives a reward based on new coverage and on new crashes, which provides a more fine-grained view on the problem.

We evaluate `StateBandit` using the industrial communication protocol OPC UA in a graybox test setting and compare the results to the stateful fuzzer `AFLNet` [Pha20], which bases its state selection on a heuristic. Our evaluation shows that `AFLNet` leads to significantly better results than each of the analyzed

agent policies. We deduced from this observation that a blackbox agent is unlikely to have enough data to improve state selection, and decided not to pursue this direction of research any further in this doctoral work.

Contribution 8. *Proposal, implementation, and evaluation of* `StateBandit`*, an approach to delegate the state selection problem to a MaB agent.*

The agent's decisions are independent of the network protocol that is being tested (Challenge 4 (Insufficient Protocol Support)), and aim to provide additional guidance and information for the test (Challenge 2 (Missing Information)).

Despite the negative results, the insights have been published at the EuroS&P workshop on *Re-design Industrial Control Systems with Security* [*Bor23a*], and are detailed in Section 7.6.

Evaluation of Test Tools

Besides proposing new approaches to security testing, it is equally important to provide means to evaluate said approaches and to compare them to the state-of-the-art. All of the above approaches have been evaluated in accordance with general evaluation best practices and, where applicable, fuzzer evaluation best practices [Kle18]. Furthermore, this doctoral work contributes directly to the evaluation of TTs, namely on the evaluation of stateful WVSs and to the evaluation of stateless fuzzers.

The first contribution to the evaluation of TTs, `SWaTEval`, is concerned with blackbox WA testing of OT component. The main objective of `SWaTEval` is to facilitate evaluation of TTs for SWAT, with a special focus on automatic blackbox state machine inference. Such a state machine can build a vantage point for stateful WVSs. The literature includes approaches for the blackbox state machine inference of WAs, but it remains unclear how these approaches perform in comparison and how different design choices of these approaches impact the quality of the inferred state machine. Therefore, we develop an evaluation framework for stateful WA testing, `SWaTEval`.

Contribution 9. *Proposal, implementation, and evaluation of* SWaTEval*, an evaluation framework for SWAT.*

For the evaluation of SWaTEval, we focus on one design choice of automated state inference, namely the similarity measure used to decide whether two web pages are to be considered the same. We evaluate SWaTEval's performance by comparing results produced by SWaTEval with results from the literature. Moreover, we propose a new similarity measure and show that this newly proposed similarity measure leads to the highest number of correctly identified states in comparison to two similarity measures from literature.

Contribution 10. *Development of a new similarity measure for web pages and use of* SWaTEval *to analyze the impact of the choice of similarity measures on the performance of state machine inference.*

Section 8.3, which is based on the publication of SWaTEval [*Bor23c*], gives more details on SWaTEval and the corresponding evaluation.

The second contribution, MEMA, is concerned with the evaluation of stateless fuzzers. As a basis, we formulate the different dimensions of fuzzing evaluations, and analyze how these dimensions have been considered by literature.

Contribution 11. *Analysis and formulation of the dimensions of fuzzing evaluations.*

Our analysis reveals that the majority of fuzzing evaluations, and also best practices for fuzzing evaluations, use either the code coverage a fuzzer achieves or the number of bugs a fuzzer reveals as metric for the evaluation [Sch24]. However, recent literature suggests that the relative performance of fuzzers can differ, depending on which of those two metrics is chosen [Fio22]. In addition, there are other metrics known from traditional software benchmarking (such as e.g. memory utilization and CPU load), which have not been extensively used for fuzzing evaluations. Thus, we implement several new metrics for fuzzing evaluations, integrate them into a fuzzing benchmark tool, and evaluate

how the selection of metrics influences the relative rating of fuzzers. For this, we conduct an evaluation of three fuzzers against five targets, using six different metrics.

Contribution 12. *Integration of new metrics into an existing fuzzing benchmark framework and use of this framework to analyze the impact of these metrics.*

See Section 8.4 for more details on MEMA.

1.5 Research Scope

The overall objective of this doctoral work is to improve blackbox security testing for OT components while considering the challenges defined in Chapter 1. More specifically, this doctoral work focuses on blackbox testing for OT components via the Ethernet interface using WVSs and Fuzzing. In the following, we put this doctoral work into the context of other research directions to clarify the scope. Note that detailed discussions of related work are presented in the corresponding chapters of this dissertation (see Sections 4.1, 5.1, 6.1, 7.1, and 8.1).

Software Testing vs. Security Testing Security testing is generally seen as one part of a thorough testing strategy for software and hardware. The general goal of security tests is to test a system's vulnerabilities to threats [Ric16], while general software testing aims to identify errors or to determine whether the system fulfills certain requirements [Cer23]. The International Software Testing Qualification Board (ISTQB) states that security testing differs from other forms of software testing by two means [Ric16]: On the one hand, the test cases used in general software testing might not be suitable to find security issues. For example, software testing might aim to reach full code coverage during testing, while security testing aims to reveal vulnerabilities as early as possible [Wan20b]. On the other hand, the symptoms of security issues are different from those monitored in other functional tests.

However, for security testing of OT components, the border between general software testing and security testing is not as clear as for other SuTs. For OT components, one of the most important security requirements is the availability [Sto23]. A crash of an OT component is therefore generally assumed to be a security issue. Nevertheless, a crash of a SuT is also something general software testing tests for.

Moreover, security testing, as defined by the ISTQB, encompasses activities during the whole development lifecycle [Ric16]. Based on this definition, the security tests conducted in this doctoral work are *system tests* as we focus on testing OT components from a blackbox point of view (see also Section 2.2.1).

Web Vulnerability Scanners and Fuzzing. There are various approaches and tools to choose from if one wants to perform security tests [*Bor19*]. Two prominent approaches that also have a high relevance for finding vulnerabilities in blackbox OT components are WVSs and fuzzing [San21c, Int19][*Pfr19b*], which are also the focus of this doctoral work. The first testing technique, WVS, probes a WA and tries to find the inputs necessary to expose a vulnerability such as an SQL Injection or Cross-Site-Scripting (see e.g. [*Pfr19b*]). Applying traditional WVSs in industrial test settings poses additional challenges (see Section 4.3). The second testing technique, blackbox network fuzzing, applies fuzzing to network protocols. Fuzzing, in general, is the approach of randomly mutating given inputs to induce anomalous behavior in the SuT that may reveal the existence of a vulnerability (see Section 2.4).

Although we focus on the two testing techniques mentioned above, we recognize that there are other relevant approaches to blackbox testing of OT components, such as *vulnerability scanning* and *penetration testing*. Tools for automated vulnerability scanning of OT components check whether the SuT is likely to include known vulnerabilities. This can, for example, be deduced from fingerprinting the SuT and then performing a lookup on known vulnerabilities of this device or libraries that it uses. Commercial tools for OT component vulnerability scanning are available, for example, by Tenable [Ten23] and Greenbone [Gre23]. In contrast, the testing methods considered in this doctoral work, WVS and fuzzing, aim to find new and previously unknown vulnerabilities of the SuT. Furthermore, it is particularly important not to rely

solely on automated approaches, but to consider manual penetration testing to ensure high quality test results. This is also required by relevant standards and best practices [Int19, Cer23]. Note that during a penetration test, automated vulnerability detection tools and automated penetration testing frameworks, such as Metasploit [Met23], are used to support the manual analysis.

Ethernet Interfaces. In general, it is necessary to analyze the security of each of the interfaces a SuT provides. For this doctoral work, however, we focus on Ethernet-based network interfaces. This is based on the observation that, caused by the convergence of IT and OT, more and more devices include Ethernet-based communication interfaces which can possibly be reached from the whole network of the facility [Haj21]. These Ethernet-based interfaces are also the most approachable for attackers since they can be used and analyzed with less specialized tools. As a result, we assume that these interfaces are the ones that have the highest risk to be attacked, and thus focus on these interfaces in the following. In contrast, other approaches to perform security tests of industrial or embedded devices use side channel information [Spe19] or the debug interface [Eis23].

Software. This doctoral work focuses on security tests of the software of the OT component. In contrast, other works focus on the fuzzing the underlying hardware [*Su24*][Lae18].

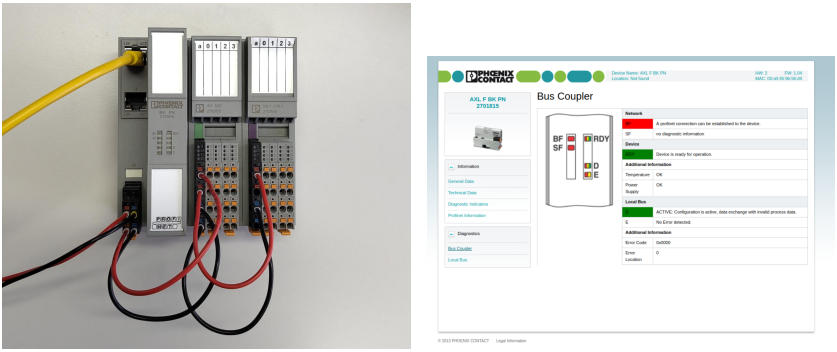
Models for Security Testing. Some of the approaches presented in this doctoral work aim to derive a model of the SuT based on blackbox test data such as the network traffic or previously seen test cases. The goal of these models is to find a balance between a high level of detail and efficient training and querying, with the main goal to improve the downstream security testing procedure, e.g., by guiding the test case generation. Particularly, this means that the models do not make the claim to be a fine-grained representation of the SuT or the used network protocol. This marks the difference to the research area of blackbox protocol reverse engineering (as e.g. used by Zheng et al. [Zhe22] and Bytes et al. [Byt23]). In protocol reverse engineering, the main goal is to find a model of the communication protocol that is as close to the actual protocol as possible. Taking this approach would introduce a high overhead before and possibly during testing.

1.6 Running Example OT Component

To clarify the motivation and the setting of this doctoral work, we introduce the running example of a bus coupler which should be tested for vulnerabilities. Note that this example was chosen to showcase the concepts and challenges discussed in this doctoral work. By no means is this type of OT component the only one which should be tested, or was showing vulnerabilities during the course of this doctoral work. Most of the devices that have been tested during this doctoral work showed vulnerabilities, not limited to a type of OT component or manufacturer.

The concrete OT component that is used as a running example is the bus coupler shown in Figure 1.2, the AXL F BK PN bus coupler, manufactured by Phoenix Contact GmbH & Co. KG . We refer to this bus coupler as BC_{ex} throughout this doctoral work.

In general, a bus coupler is an industrial device which has the task to transfer signals from one communication protocol to another. For example, a bus coupler can receive packets from a controller via the Ethernet-based industrial



(a) Picture of BC_{ex}. **(b)** Screenshot of the WA provided by BC_{ex}.

Figure 1.2: The bus coupler (BC_{ex}) used as a running example for this doctoral work. It is typically deployed in a setting where a controller is connected to the bus coupler via the Ethernet interface (yellow cable on the left). The bus coupler will then translate the control instructions it receives via Ethernet into digital I/O signals which are delivered by the digital I/O interfaces at the bottom right.

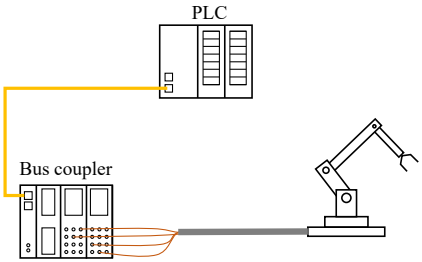


Figure 1.3: An exemplary use case in which the bus coupler from the running example could be deployed. The controller sends control signals to the bus coupler via an industrial communication protocol such as PROFINET. The bus coupler then translates these signals to digital signals, and sends them to the industrial robot.

protocol PROFINET and transform them into corresponding digital I/O signals which then control an industrial robot (see Figure 1.3). With this setting, the control instructions from the controller can be routed via Ethernet until a point right before the industrial robot. Only then are the instructions translated into digital I/O signals. In this setting, it is especially crucial that the digital I/O interfaces work as expected and are resilient against cyber attacks, as the availability and productivity of the bus coupler directly influences the industrial robot. Although safety measures should prevent the industrial robot from doing any harm to humans, an outage of the production steps performed by the industrial robot could potentially result in a significant economic loss to the production facility. See Section 2.1.1 for a more detailed discussion on the differences of safety and security.

BC_{ex} is a PROFINET based bus coupler, meaning that it expects commands via PROFINET, and translates these commands into digital I/O signals. In addition to the features crucial for its operation, this bus coupler also provides a WA and an FTP server. The WA displays information about the device and its current status but does not accept any user input. Regarding security, this is a good feature, since many vulnerabilities are based on faulty or unexpected user input [OWA21]. A screenshot of this WA is shown in Figure 1.2b. The FTP server provides access to some of the files on the bus coupler. BC_{ex} is especially suited for safety relevant use cases as it fulfills the requirements of the Safety Integrity Level (SIL) 3, which is the second highest SIL [Int10].

2 Background

The following sections establish the foundations relevant to the contributions of this doctoral work. Section 2.1 begins with an overview of relevant topics in the domain of industrial networks, emphasizing the distinction between safety and security as well as the network protocols utilized in this doctoral work. Section 2.2 explores industrial security testing, including a discussion of key terminology and practical considerations, and introduces the industrial security testing framework ISuTest®. Subsequent sections cover Web Vulnerability Scanners (WVSs) in Section 2.3, fuzzing in Section 2.4, and stateful testing in Section 2.5. Finally, Section 2.6 provides an overview of the Machine Learning (ML) methods applied in this doctoral work.

2.1 Industrial Networks

Ensuring the safety and security of an industrial facility requires consideration of all its components, including the Supervisory Control and Data Acquisition (SCADA) system, Operational Technology (OT) components, sensors, actors, and network components [Kna24]. Additionally, human factors that can lead to data theft and sabotage need to be taken into account [Pot23].

2.1.1 Safety and Security

Safety generally focuses on preventing harm that may result from random malfunctions or errors. In industrial environments, safety functionality is typically automated, ensuring that systems either operate correctly or transition to a *fail-safe* state. This fail-safe state usually halts production safely, such as by

stopping all moving parts in a safe position. This automated aspect of safety is known as *functional* safety [Int10]. The primary objective of functional safety is freedom of unacceptable risks of physical injury and of damage to people's health [Li18b]. Guidelines for implementing safety-relevant OT components are outlined in the international Standard IEC 61508 [Int10].

In contrast, *security* aims to prevent harm caused by malicious attackers [Pet20]. This requires addressing more targeted threats to mitigate potential damage. Additionally, security must consider additional assets such as confidential data and intellectual property. Generally, three key security requirements are confidentiality, integrity, and availability. In industrial environments, availability of systems is particularly critical [Sto23]. With IEC 62443, a standard for security in industrial environments has been published [Int19]. This standard includes several security measures for OT components, especially blackbox security testing.

To clarify the difference between safety and security in industrial environments, we take the safety protocol IO-Link Safety, that is commonly used in industrial environments, as an example. IO-Link Safety employs several measures to meet safety requirements, including a sequence number called *MCount* to ensure the correct order of the packets, and a Cyclic Redundancy Check (CRC) to ensure that messages have not been altered during transmission [Int22]. The CRC adds additional redundancy bits to a network packet that represent a checksum of the packet's data. The receiver of a network packet calculates the checksum of the received data and compares it to the CRC provided with the packet. This process allows the receiver to detect any data changes that occurred during transmission.

While both the sequence number and the CRC effectively detect transmission errors, they do not prevent data manipulation by an attacker. An attacker can intercept a network packet, alter the data, and then recalculate the CRC for the modified packet. This is feasible because all information required to compute the CRC is either publicly available, such as the generator polynomial, or included in the packet itself, such as the sequence counter [Int22]. Then, the attacker can include the recalculated CRC in the altered packet, making the alternation undetectable for the receiver.

With this attack, an attacker could alter the content of network packets exchanged between an emergency stop button and a Programmable Logic Controller (PLC). When the button is pressed, this information is usually transmitted via network to the PLC, which then instructs the production process to transition to a fail-safe state. However, if an attacker can manipulate these network packets, the emergency stop signal could be suppressed, effectively preventing the production process from halting. This could potentially result in physical injuries and damage to production facilities.

Although the measures provided by IO-Link Safety fulfill safety requirements, they do not meet security requirements such as the integrity of the communication. Thus, the communication is not protected from deliberate attacks.

2.1.2 Network Protocols

OT components can be accessed by connecting to one of the communication interfaces they provide. Thus, these interfaces serve as potential entry points for attackers and are therefore critical to the security of OT components. This doctoral work focuses on Ethernet-based communication interfaces and considers several network protocols. For each network protocol, a brief introduction is provided in this section, and additional sources for further reading are referenced.

Transmission Control Protocol (TCP)

TCP is a transport layer protocol of the internet protocol suite and is defined in RFC 793 [Pos81b]. It builds upon the Internet Protocol (IP), which is a network layer communication protocol defined in RFC 791 [Pos81a]. The corresponding datagrams can be routed and thus can be sent between different networks, which allows attackers to access systems remotely. As various modern protocols build upon TCP, several OT components include a corresponding network stack [San21c]. We analyze the security of TCP/IP stacks of OT components with `ClusterCrash` (see Chapter 5).

File Transfer Protocol (FTP)

FTP is an application layer protocol used for file transfer [Pos85]. It is a text-based protocol, which means that it sends commands and data in plain text via the network. In contrast to binary protocols such as OPC UA (see below), this allows for an easier interpretation and generation of FTP data. Our experiments on the preprocessing of network traffic and the blackbox fuzzing based on Hidden Markov Models (HMMs) are based on FTP (see Sections 6.4 and 7.5).

Hypertext Transfer Protocol (HTTP)

HTTP is an application layer protocol of the internet protocol suite which is used, for example, to provide Web Applications (WAs) [Fie99]. It is a stateless request-response protocol, in which a client typically requests some content, such as a web page, from a server. The server's response includes a response code which indicates the status of the request. For example, the status code 200 indicates that the request was processed successfully. The WVSs used within `HitM` as well as the automatic state machine inference of WAs both utilize HTTP (see Chapter 4).

OPC Unified Architecture (OPC UA)

OPC UA is a modern machine-to-machine communication protocol used in industrial environments [OPC22]. It provides three security modes with increasing security guarantees, and requires several steps to establish a full communication channel. We use the open source OPC UA network stack open62541 for our experiments on stateful fuzzing based on Multi-armed Bandit (MaB) agents (see Section 7.6).

2.2 Industrial Security Testing

Although industrial security needs to consider all systems within an industrial environment, this doctoral work specifically focuses on the security of OT components. Ensuring their security requires implementing a secure development lifecycle, which addresses security considerations throughout each stage of the development [Int19]. Aspects of this lifecycle include threat modeling, secure coding standards, security testing, and secure update management. This doctoral work specifically addresses security testing of OT components.

2.2.1 Terminology

In literature, the term *security testing* is used inconsistently. Based on the definition provided by the International Software Testing Qualification Board (ISTQB), security testing encompasses activities in each step of the development lifecycle [Ric16]. For example, defined requirements should be evaluated from a security perspective, and security-related design approaches should be considered during the design phase. Caselli et al. use a similar definition, including activities such as document-based design reviews and code reviews in their definition of security testing [Cas16]. In contrast, Felderer et al. propose a narrower definition, including only the dynamic verification of whether a system correctly implements the intended security properties [Fel16a]

For this doctoral work, we adopt the definition presented by Felderer et al. Established security properties that are considered in security testing include confidentiality, integrity, availability, authorization, and non-repudiation [Fel16a]. In industrial environments, and especially for testing OT components, the availability of the System under Test (SuT) is a crucial security property [Sto23]. If this property is not met by an OT component, an attacker could, for example, crash the OT component, potentially halting the corresponding production process.

2.2.2 Practical Considerations

In addition to the general challenges in industrial security testing discussed in Chapter 1, conducting industrial security tests involves several practical considerations, which are discussed in the following sections.

2.2.2.1 Resilience of OT Components

The most prominent observation during industrial security tests is that OT components are generally less resilient to high loads compared to Information Technology (IT) systems. As a result, several Test Tools (TTs) impose excessive load on OT components during testing, leading to overload and crashes of the SuT [*Pfr19b*]. While this also marks a finding of the security test, it prevents further testing. Therefore, for industrial security tests, TTs that can manage crashes of the SuT and allow to adapt the load put onto the SuT are more suitable.

2.2.2.2 Responsible Disclosure

If a security test reveals security-relevant findings, they should be responsibly disclosed to the corresponding manufacturer. The term *responsible disclosure* describes the approach of publishing findings while considering the manufacturer's interests [Mou23]. Although specific procedures and requirements may vary, responsible disclosure generally involves informing the manufacturer of the finding and providing a reasonable amount of time to address the issue before making the finding public. For instance, the Open Worldwide Application Security Project (OWASP) outlines rules for responsible disclosure, particularly concerning findings related to WAs [OWA24b].

Since the findings revealed in this doctoral work may affect critical infrastructure, we disclose our findings to the manufacturers and refrain from publishing them without the manufacturers' consent. For some findings, we contact the manufacturer directly, while for others, we choose to contact the manufacturer

through CERT@VDE[1], the Computer Emergency Response Team (CERT) of the German Verband der Elektrotechnik, Elektronik und Informationstechnik (VDE). This CERT acts as a general point of contact for reporting security issues related to OT components.

2.2.2.3 Common Vulnerability Scoring System (CVSS)

The Common Vulnerability Scoring System (CVSS)[2] is an established scoring system used to quantify the severity of vulnerabilities. It considers the (1) intrinsic characteristics of a vulnerability, (2) currently existing exploits, and (3) the environment of the system exhibiting this vulnerability. CVSS scores range from 0 to 10, where 0 represents no security issue and 10 denotes a critical vulnerability.

The CVSS score is the de facto standard for quantifying vulnerabilities and is widely used by institutions such as the National Vulnerability Database (NVD) [Sca08]. However, it should be noted that Spring et al. call for a revised approach to vulnerability scoring [Spr21]. The authors argue that there is no evidence supporting the robustness of the CVSS calculation and that the specification provides limited transparency on the formula used to calculate the CVSS score.

As CVSS continues to be a standard for vulnerability scoring, we assign a CVSS score to the vulnerabilities identified during this doctoral work. A summary of these vulnerabilities and their respective scores is provided in Section 9.2.1.

2.2.3 ISuTest®

ISuTest® is a framework for industrial blackbox security tests developed at Fraunhofer IOSB [*Pfr23*]. While the original version of ISuTest® was built based on the vulnerability scanner OpenVAS [Pfr17, Pfr18], ISuTest® 2.0 was built independently of other TTs.

[1] https://cert.vde.com/en/
[2] https://www.first.org/cvss/

ISuTest® allows for blackbox network fuzzing of OT components, and supports standard internet protocols, such as TCP, and industrial protocols, such as PROFINET. In its base version, ISuTest® generates test cases independently of the behavior of the SuT and uses fixed heuristics for test case mutation. To account for Challenge 3 (Insufficient Observations), ISuTest® allows to monitor the current status of the SuT via various Ethernet-based communication protocols and the digital I/O interface of the SuT if it provides one. This monitoring is used to analyze whether a SuT exhibits anomalies or crashes during the test. Moreover, ISuTest® supports using switch actuators which allow for an automatic power reset of the SuT. With this, OT components can be restarted automatically and thus can potentially recover from crashes triggered by the tests to allow further testing.

From a high-level perspective, testing an OT component with ISuTest® involves the following steps:

1 The tester connects and configures the SuT and ISuTest® such that ISuTest® can connect to the communication endpoints required for testing and monitoring. Then, the tester selects the network protocols to be tested, along with the number of test cases to be generated for each network packet field. Moreover, the tester determines the monitoring interval, which specifies how many test cases are executed before a monitoring cycle is conducted. During this monitoring cycle, ISuTest® verifies whether all defined communication endpoints of the SuT are functioning as expected. This might involve requesting a WA served by the SuT, or checking whether the digital I/O signals continue to output the expected values.

2 Then, ISuTest® generates the requested number of test cases for each protocol field using predefined heuristics. After each batch of test cases, as determined by the monitoring interval, a monitoring cycle is performed. If the monitoring indicates that the SuT crashed, it is restarted by performing a power reset. Anomalies and crashes observed during the monitoring cycles are presented to the tester.

3 Further, ISuTest® can analyze identified anomalies and crashes to provide a minimal set of test cases necessary to reproduce the crash or anomaly [*Dil20*].

For more detailed information on ISuTest®, refer to the corresponding publications [Pfr17, Pfr18] [*Pfr19b, Pfr19a*].

`HitM` and `Smevolution`, as presented in Sections 4.3 and 7.4, are implemented using ISuTest®, taking advantage of its fuzzing framework and monitoring capabilities. Additionally, the test scripts defined with `ClusterCrash` are integrated into ISuTest®, enabling the novel insights to be applied in future security tests with ISuTest®.

2.3 Web Vulnerability Scanners

The general approach of *vulnerability scanners* is to first scan a network to identify any systems connected to the network, as well as the operating systems and services they provide. Then, they compare this information to a database of known vulnerabilities [Hol11]. With this approach, vulnerability scanners identify whether systems incorporate known vulnerabilities, but they typically do not analyze whether these vulnerabilities can be exploited for the specific system. In addition, they do not test the systems for previously unknown vulnerabilities.

In contrast, a *Web Vulnerability Scanner (WVS)* focuses on finding known and previously unknown vulnerabilities in WAs by actively interacting with the WA. A WVS typically approaches this by (1) crawling the WA, (2) finding input possibilities, and (3) providing tailored inputs to the WA [Dou10]. Before starting the test, the WVS is provided with at least one initial Uniform Resource Locator (URL). It retrieves the corresponding web page, and then *crawls* the WA by following links and redirects to identify all reachable pages [Dou12]. In addition, the crawling identifies possibilities to input content to the WA, such as input fields. After the crawling, the acquired knowledge is used to identify vulnerabilities in the WA. To this end, the WVS provides input to the

WA, for example with the objective to reveal an SQL injection vulnerability. Subsequently, the WVS analyzes the responses of the SuT to identify if a vulnerability was found.

With this approach, WVSs potentially find both known and previously unknown vulnerabilities. Moreover, some WVSs incorporate approaches to identify the software libraries used by the WA and verify whether vulnerabilities are known for this specific software version. This corresponds to the approach of general vulnerability scanners as described above.

In this doctoral work, we transparently improve the performance of WVSs in an industrial test setting with `HitM` (see Section 4.3). Additionally, we introduce the modular evaluation framework for stateful WA testing `SWaTEval` (see Section 8.3).

We acknowledge that more diverse TTs are necessary for a thorough security test of a WA. For instance, OWASP provides an extensive guide on web security testing [Saa24], detailing various test approaches and TTs. Additionally, OWASP curates a list of both established open source and commercial WVSs [OWA20].

Terminology

While most authors, including OWASP, use the term *Web Vulnerability Scanner* to describe the aforementioned TTs (see, e.g., [Dou10, OWA20, Urb22]), the terminology is not consistent. For example, Fong et al. and Alassmi et al. use the term *Web Application Scanner* [Fon07a, Ala12], while Bau et al. and Alazmi et al. use the term *Web Application Vulnerability Scanner* [Bau10, Ala22].

For this doctoral work, we use the term *Web Vulnerability Scanner (WVS)* to refer to TTs that crawl and analyze a WA to identify previously known and unknown vulnerabilities, consistent with the majority of recent publications. In addition, we use the term *vulnerability scanner* to denote automatic TTs that analyze whether systems include known vulnerabilities by identifying provided services and comparing this with information from a vulnerability database.

Evaluation

Evaluating the performance of WVSs is an ongoing topic addressed by both academia and community projects. Notable efforts include the Web Application Vulnerability Scanner Evaluation Project (WAVSEP) by Chen [Che18], with an updated version published by Urbano et al. [Urb22], as well as the OWASP benchmark [OWA24a].

Section 6.4.1 provides an overview of evaluations conducted in the literature, analyzing the WVSs considered, the target WAs used for the evaluations, and the limitations of the WVS that were identified. Note that the novel framework presented in this doctoral work, `SWaTEval`, focuses on evaluating *stateful* WVSs (see Sections 2.5 and 8.3).

2.4 Fuzzing

Fuzzing is a testing technique that has been used for several years and continues to be relevant for revealing new bugs and vulnerabilities [Mil20]. It is particularly effective in finding bugs within network stacks, which are especially relevant for OT component security [San21c].

The general approach of fuzzing involves presenting inputs, which are called test cases, to the SuT. These test cases may be semantically or syntactically malformed. The behavior of the SuT in response to these test cases is monitored to identify anomalies and crashes [Man19].

Section 2.4.1 provides an overview of fuzzing approaches, and Section 2.4.2 presents a general description of the fuzzing process, including the key steps and decisions involved. For a comprehensive description of the fuzzing process, we refer to the works of Manès et al. [Man19], Giraud [*Gir20*], and Jiang et al. [Jia24]. Finally, we discuss methods to assess a fuzzer's performance (Section 2.4.3).

2.4.1 Categorization of Fuzzers

Fuzzing approaches can be classified along several dimensions, the most prominent of which are (1) the level of information available to the fuzzer, (2) its approach to generating test cases, (3) and the guidance provided to the fuzzer. As these three dimensions are particularly relevant to this doctoral work, we discuss them in more detail. For an extensive taxonomy of fuzzers, refer to the work by Mallissery et al. [Mal23].

Available Information

Fuzzers are classified into *whitebox*, *graybox*, and *blackbox* fuzzers [Man19]. Blackbox fuzzers have no access to any information internal to the SuT and therefore rely primarily on randomly generating new test cases, and concurrently monitoring the SuT for anomalies or crashes [Böh21]. As blackbox fuzzers also rely on potentially incomplete external information for this monitoring, they may reach inaccurate conclusions, such as failing to detect certain anomalies of the SuT [Mue18].

In contrast, whitebox fuzzers have full access to internal information such as the source code of the SuT. These fuzzers typically introduce more overhead than blackbox fuzzers and are more tailored to one specific programming language [Man19].

In between these two categories are graybox fuzzers, which have access to some information about the SuT [Böh21]. Typically, graybox fuzzers rely on program instrumentation to obtain feedback on which parts of the code where executed by the SuT in response to a test case. If a test case covers code that was not previously executed, it is used as a basis to generate new test cases. This approach allows the fuzzer to gradually explore deeper parts of the SuT's code. To observe the SuT's behavior to collect information used for this guidance, such as memory accesses, graybox fuzzers utilize *sanitizers* [Öst20].

In this doctoral work, we focus on a blackbox setting but aim to incorporate approaches from graybox fuzzing. Nevertheless, we consider graybox fuzzing for `StateBandit` to evaluate whether this novel approach can enhance fuzzing assuming access to graybox information (Section 7.6). Additionally, we focus our work with respect to fuzzing evaluations on graybox fuzzing, in order to address a broader range of fuzzers (Section 8.4).

Test Case Generation

Based on their test case generation approach, fuzzers can be divided into two categories: *generational* fuzzers and *mutational* fuzzers [Zha24b]. Generational fuzzers rely on a specification of the test cases, such as a context-free grammar or a format specification [Wan17]. This allows the fuzzers to generate test cases that pass integrity checks that may be deployed by the SuT.

In contrast, mutational fuzzers generate test cases by mutating an initial set of well-formed inputs, known as the *corpus* [Wan17]. Common mutations include replacing or deleting parts, or appending parts from other test cases [Tri23]. In graybox fuzzing, the corpus is typically extended by test cases that lead to new coverage during the fuzzing campaign, allowing subsequent test cases to be derived from these newly generated test cases as well.

In this doctoral work, we use a mutational fuzzing approach for `Smevolution`, `Palpebratum`, and `StateBandit` (Sections 7.4 to 7.6). The fuzzing test scripts generated with `ClusterCrash` utilize the structures derived from the Vulnerability Anti-Patterns (VAPs) and therefore follow a generational fuzzing strategy (Section 5.4).

Fuzzer Guidance

In graybox fuzzing, the two most prominent approaches to guide the fuzzing process are *coverage-guided* fuzzing and *directed* fuzzing. Coverage-guided fuzzing aims to maximize the code coverage achieved by the test cases sent to the SuT, typically measured in block coverage or in line coverage [Mal23].

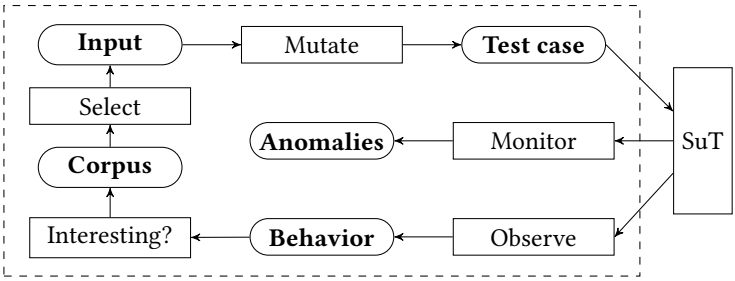


Figure 2.1: General process of coverage-guided mutational graybox fuzzing. The elements of the fuzzer are shown inside the dashed lines.

In contrast, directed fuzzing focuses on reaching a specific target location within the SuT, using this location as the primary target to guide the fuzzing process [Böh17].

2.4.2 Fuzzing Process

As we aim to improve blackbox fuzzing by incorporating methods from graybox fuzzing, we utilize the terms and concepts of mutational coverage-guided graybox fuzzing throughout this doctoral work, and present them below. We describe the graybox fuzzing process at a level suited for understanding the novel approaches presented in this doctoral work, while a more formalized description of the fuzzing process can be found in the publications by Manès et al. and Salls et al. [Man19, Sal20].

Figure 2.1 visualizes the general process of mutational coverage-guided graybox fuzzing. Usually, mutational graybox fuzzers assume access to a set of valid input files for the SuT, which are called the *seeds*. Depending on the SuT, this could be a set of PDF files or a set of network packets. However, it is also possible to start fuzzing without seeds. The initial seeds should be selected carefully, as research shows that they influence the performance of the fuzzer [Kle18].

One run of a fuzzer is called a fuzzing *campaign*. Usually, the end of a campaign is defined by a predetermined time budget [Böh21], although some approaches stop after the first crash of the SuT [Kle18], and Liyanage et al. suggest an

adaptive approach [Liy24]. During the fuzzing campaign, the fuzzer keeps track of a set of inputs, called the *corpus*. This corpus is initialized with the seeds and evolves during the campaign.

At the start of one fuzzing *cycle*, the fuzzer selects one input from the corpus based on a selection strategy. Then, this input is mutated using a predetermined or dynamic set of mutations to generate a new test case t. This test case t is then sent to the SuT. The behavior of the SuT is analyzed by two different means. On the one hand, the fuzzer *monitors* the behavior of the SuT to identify anomalies or crashes. For example, this can be supported by address sanitizers which monitor memory accesses by the SuT and alert the fuzzer if the SuT performs out-of-bounds accesses [Öst20]. Test cases leading to anomalies and crashes are stored to be reported to the tester. On the other hand, the fuzzer *observes* the code coverage achieved by t. If t achieves coverage in parts of the SuT that were not previously covered, t is deemed *interesting*, as it provides a way to reach new areas of the SuT's code. Consequently, t is added to the corpus and can be selected as a starting point for further mutations in subsequent fuzzing cycles.

In blackbox testing, the fuzzer's ability to monitor and observe the SuT is limited. As a result, most blackbox fuzzers do not incorporate observations of the SuT's behavior in the test case generation [Kle18], and only detect crashes of the SuT that are externally visible [Mue18]. In contrast, `Smevolution` and `Palpebratum` apply the graybox fuzzing process to a blackbox testing scenario (see Sections 7.4 and 7.5), showing how ML-based approaches can be leveraged for blackbox fuzzing.

2.4.3 Fuzzer Evaluation

Klees et al. [Kle18] defined best practices for fuzzing evaluations that are widely accepted by the fuzzing research community and were explicitly confirmed by Schloegel et al. [Sch24]. However, Schloegel et al. also highlight that many publications featuring fuzzing evaluations still exhibit shortcomings concerning these best practices. In Section 8.4.2, we discuss the dimensions

of and methodologies of fuzzing evaluations in more detail. In this doctoral work, we adhere to these best practices as far as they are applicable to the industrial use cases and provide justifications for any deviations.

As discussed by Gopinath et al., fuzzing literature has no strong agreement on what the terms *effectiveness*, *efficiency*, and *efficacy* mean in the context of fuzzing evaluations [Gop22]. We follow the terminology by Gopinath et al. and define the effectiveness of the test cases generated by a fuzzer as the average coverage a test case achieved.

2.5 Stateful Testing

Many SuTs are *stateful*, meaning their behavior is determined by an internal state machine, and their response to a specific input depends on the current state. Typically, the state of the SuT is influenced by the inputs from the fuzzer. If the fuzzer is unaware of the SuT's statefulness and how its inputs impact the SuT's state, it becomes significantly more challenging to interpret and utilize the SuT's behavior [Dan24].

State-aware fuzzing is particularly relevant for both network protocol fuzzing [Rui15, Pha20, Nat22, Ba22, Liu22, Pfe22, Amu23] and WA fuzzing [Dou12]. However, it also introduces additional challenges, as a stateful fuzzer needs to handle individual inputs as well as traces of inputs that potentially influence the state of the SuT [Dan24]. This doctoral work addresses two of these challenges, which will be discussed below: (1) state selection, and (2) state machine inference.

2.5.1 State Selection

Assuming that a fuzzer is aware of the statefulness of the SuT and has access to a representation of the SuT's states, the fuzzer needs to decide which states to focus on during fuzzing. This decision can affect a fuzzer's performance, as bugs and new coverage might be unevenly distributed across different states. Consequently, it is advantageous for the fuzzer to prioritize states that are

more likely to reveal bugs and new coverage. However, since the fuzzer does not have prior knowledge of which states will be most profitable, it must make this decision dynamically during fuzzing. This challenge is known as the state selection problem in stateful fuzzing [Liu22].

Typically, the fuzzer selects a state of the SuT and runs a predetermined number of fuzzing cycles in that state before switching to another state. We refer to the fuzzing cycles run before the next state selection as a fuzzing *round*.

We address the state selection problem in graybox fuzzing with `StateBandit` (see Section 7.6).

2.5.2 State Machine Inference

In most cases, we cannot assume access to an existing representation of the internal states of the SuT. Therefore, such a representation needs to be generated automatically during testing, for example by inferring a state machine based on the SuT's responses [Dou12]. The specific representation of the SuT's states depends on both the SuT and in which manner the fuzzer aims to utilize this information [Dou12, Ba22] [*Gir20*].

As this doctoral work's contribution to state machine inference, `SWaTEval`, focuses on blackbox WA fuzzing, we explain the general approach to state machine inference in this domain in more detail. The state machine inference for WAs, as presented by Doupé et al., is based on the assumption that a WA is deterministic and will generally send the same response to a given request if its internal state has not changed [Dou12]. Suppose a fuzzer sends a request A_1 to a WA and receives the response X. Then, the fuzzer sends different requests and receives corresponding responses from the WA. Later, the fuzzer sends the request A_2, which has the same content as A_1, and receives the response Y. If $X \neq Y$, the fuzzer received a different response to the same request and thus can infer that a state change has happened between A_1 and A_2. Based on this observation, the fuzzer can build and update an approximation of the underlying state machine during the fuzzing process.

The procedure for automatically inferring the state machine of a WA by monitoring the requests and responses during interactions with a fuzzer, as presented by Doupé et al., consists of the following steps. For detailed information on these steps, refer to the original publication [Dou12].

1 Run a crawler and a fuzzer against a WA and monitor the resulting request-response pairs to identify those pairs that have identical requests but different responses. Assume that a state change occurred between every two of these pairs.

2 Determine the state-changing request for every two pairs by using a heuristic. This heuristic prioritizes HTTP POST requests over HTTP GET requests, and favors requests that previously changed the state over those that have not.

3 Add a new state to the state machine for each state-changing request and collapse the state machine to avoid state explosion.

4 Continue with running the crawler and the fuzzer (step 1).

We utilize this approach to state machine inference to build the novel evaluation framework for stateful WA testing, `SWaTEval` (see Section 8.3).

2.6 Machine Learning

Several approaches presented in this doctoral work apply *Machine Learning (ML)* methods to security testing. The following sections provide background information on the methodologies and models used. In this doctoral work, we utilize methods from supervised ML, unsupervised ML, and Reinforcement Learning (RL).

In supervised learning, the learning process is based on training data consisting of pairs of data points and labels. The objective is to approximate a function that maps the data points to their corresponding labels. For the approaches presented in this work, specifically `Smevolution` and `NeDaP`, we employ a

Support Vector Machine (SVM), a Decision Tree (DT), a Neural Network (NN), an Autoencoder (AE), and a Convolutional Neural Network (CNN). These methods are detailed in Sections 2.6.1 to 2.6.3.

In contrast, unsupervised learning relies solely on the data points, as no labels are available. From this domain of ML, we utilize a Principal Component Analysis (PCA) in `NeDaP` and an HMM in `Palpebratum`. The foundational concepts of these methods are presented in Sections 2.6.4 and 2.6.5.

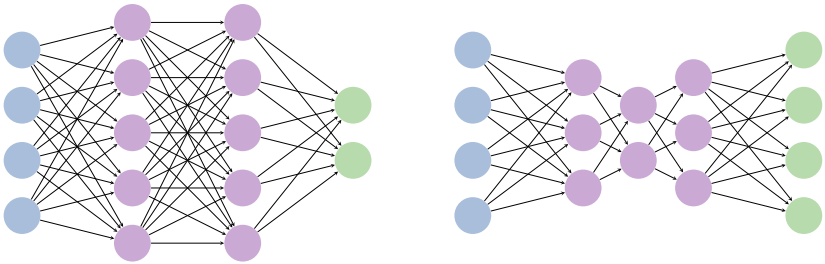
In RL, an agent interacts with an environment with the goal of maximizing the cumulative reward [Mah20]. For `StateBandit`, we utilize a specific RL problem known as the MaB problem. Since we adapt the general MaB problem to address the state selection problem in stateful network fuzzing, understanding the specifics of the MaB problem is crucial for comprehending `StateBandit`. Therefore, we present the foundational concepts of the MaB problem alongside the descriptions of `StateBandit` in Section 7.6.1.1.

As we apply ML methods to new use cases rather than presenting novel approaches to ML itself, the descriptions in the following sections focus on the characteristics of the different ML methods that are important for their application. Additionally, we provide information on further reading for each presented ML method.

2.6.1 Support Vector Machine

In their basic form, SVMs are classifiers that can discriminate two classes, but they can also be extended to support multi-class classification [Sut16]. A *linear* SVM separates the two classes using a hyperplane, making it suitable only for classes that are linearly separable. In contrast, a *non-linear* SVM first maps the data into a space where the classes become linearly separable, and then performs the linear separation in this new space.

According to Cervantes et al., SVMs are amongst the most commonly used classification methods [Cer20]. This popularity is largely due to their strong generalization capabilities, particularly with small training data sets. However, SVMs are based on complex algorithms that can increase training time.



(a) Neural Network **(b)** Autoencoder

Figure 2.2: Exemplary structures of an NN and an AE. A data point is fed into the input layer on the far left (blue), passed through the hidden layers represented in purple, and the output is given in the green output layer.

2.6.2 Decision Tree

DTs classify data points using a tree structure where each internal node represents a decision based on a specific attribute of the data point, and each leaf node is associated with a class label [Mit97]. To classify a data point, fist, the attribute at the root node of the tree is evaluated for the given data point. Based on this evaluation, the corresponding branch of the tree is followed. This process is repeated recursively for the subtree rooted at the subsequent node, until a leaf node is reached. At this point, the associated class label is assigned to the data point.

The tree-based structure allows for an interpretation and explanation of the decisions of a DT [Mah21]. Specifically, it is possible to analyze the path through the tree that was followed to classify a particular data point. By examining this path, one can determine which attributes of the data point need to be modified in order to alter its classification [App18].

2.6.3 Neural Network

In contrast to SVMs and DTs, NNs approximate a function that maps data points to class labels that allows for efficient gradient calculations [She19]. In their basic form, NNs consist of a set of connected units, where each unit takes an input and produces an output based on the input and an activation

function [Mit97]. These units are typically arranged in a layered structure, as illustrated in Figure 2.2a. To classify a data point, it is fed into the *input layer* of the NN. The units are then progressively evaluated based on the input they receive and their activation function. The predicted class label can be obtained from the *output layer* of the NN.

NNs are particularly well-suited for tasks where the training data may contain errors. They enable fast classification of data points, though they require relatively long training times. In addition, the decisions of an NN are not easily interpretable by humans [Mit97].

2.6.3.1 Autoencoder

AEs are an unsupervised variant of NNs where the input and output layer have the same dimensionality, while the hidden layers have a smaller dimension [Gér22]. This structure is illustrated in Figure 2.2b. During training, the AE is tasked to reconstruct the input values at the output layer. To achieve this, the hidden layers force the AE to learn a compact representation of the input, known as the *latent* representation. AEs are commonly used for tasks such as dimensionality reduction and data generation.

2.6.3.2 Convolutional Neural Network

CNNs are designed specifically for image recognition and are based on the concept of restricting the area of the input image a unit in the NN is connected to [Gér22]. This is achieved through *convolutional* layers, where each unit is connected to a subset of units in the previous layer. This localization allows the units to focus on specific regions of the image or data. Additionally, *pooling* layers aggregate information from the preceding layer, also focusing on a localized area. For instance, in a *max* pooling layer, each unit outputs the maximum value from the units it is connected to.

This locality-focused structure enables CNNs to efficiently process large images and is particularly effective for pattern recognition tasks [Wu17]. Moreover, CNNs are also used in other domains, such as network traffic classification [Chi20].

Convolutional Autoencoders are constructed similarly to general AEs, incorporating convolutional and pooling layers to leverage the benefits of CNNs [Mao16].

2.6.4 Principal Component Analysis

PCA is a widely used technique for dimensionality reduction [Gér22]. It operates by identifying a hyperplane close to the data and then projects the data onto this hyperplane. The objective is to choose the hyperplane in such a way that it preserves the maximum amount of variance from the original data, thereby minimizing information loss during the projection. Since PCA relies solely on deterministic calculations, it does not require a training phase or involve any randomness.

2.6.5 Hidden Markov Model

HMMs are used to represent dynamic processes over time where the current state is not directly observable [Suc15]. Instead, only the observation symbols emitted by the states of the HMM are observable, resulting in an *observation sequence*. Thus, HMMs account for two key aspects of a process's uncertainty: (1) the probability of the next state S_{t+1} given the current state S_t, denoted as $P(S_{t+1} \mid S_t)$, and (2) the probability of an observation O_t given a state S_t, denoted as $P(O_t \mid S_t)$.

In an HMM, the *hidden*, non-observable states of are modeled as a Markov chain, which adheres to the Markov property. According to this property, the next state S_{t+1} depends only on the current state S_t and not on any preceding states:

$$P(S_{t+1} \mid S_t, S_{t-1}, \dots) = P(S_{t+1} \mid S_t). \tag{2.1}$$

Formally, an HMM is defined by the following parameters [Rab86, Suc15], where N is the number of states, M is the number of observation symbols, and S_t represents the state of the HMM at time t.

- States

  $Q = \{q_i\}, i = 1, \dots, N$

- Observations symbols

  $V = \{v_k\}, k = 1, \dots, M$

- Initial probabilities

  $\Pi = \{\pi_i\}$, where $\pi_i = P(S_0 = q_i), i = 1, \dots, N$

- Transition probabilities

  $A = \{a_{ij}\}$, where $a_{ij} = P(S_{t+1} = q_j \mid S_t = q_i), i,j = 1, \dots, N$

- Observation probabilities

  $B = \{b_{ik}\}$, where $b_{ik} = P(O_t = v_k \mid S_t = q_i), i = 1, \dots, N, k = 1, \dots, M$

Tasks

Given an HMM, we are usually interested the following three tasks [Rab86, Suc15]. For each task, we assume a given observation sequence O and an HMM λ.

Evaluation Compute the probability of the observation sequence: $P(O \mid \lambda)$. This task is addressed using the Forward algorithm.

Decoding Compute the most probable state sequence that leads to O. This problem can be solved using the Viterbi algorithm.

Learning Adjust the HMM parameters to maximize the probability of O: $P(O \mid \lambda)$. This corresponds to training the HMM and is typically approached with the Baum-Welch algorithm.

Multivariate HMMs

In domains such as speech recognition, univariate HMMs are used, which emit scalar values. In contrast, *multivariate* HMMs are designed to handle multidimensional observations [Liu10]. This allows them to model more complex and richer patterns in the data.

Second-order HMMs

Typically, an HMM is based on a first-order Markov chain, where the state transition probability depends only on the current state and not on previous states (as described in Equation (2.1)). Second-order Markov chains extend this concept by making the state transition probability dependent on both the current state and the preceding state [Mar96]. This extension allows the model to represent more complex relationships between the states. However, a second-order Markov chain M_2 with N states can also be modeled by a first-order Markov chain M_1 with N^2 states [Rus16]. To model M_2, each state of M_1 is used to represents a pair of states of M_2.

3 Accessible Information

Although blackbox testing assumes no access to the internal details of the System under Test (SuT), a blackbox Test Tool (TT) may still have access to certain external information. In the following, we analyze the types of information that blackbox TTs have utilized, as documented in the literature, and classify them to provide an overview of the available and commonly used information in blackbox testing. Furthermore, we classify the approaches presented in this doctoral work to contextualize them within the broader field.

In their basic form, blackbox TTs do not consider the behavior of the SuT for test case generation. Instead, they solely monitor the SuT's behavior to identify test cases that result in crashes or anomalies in the SuT [God20]. This is demonstrated by TTs such as `radamsa` [Hel24] and the core version of ISuTest® [Pfr17].

However, other blackbox TTs utilize the accessible information to guide their testing process and test case generation. Table 3.1 provides an overview of these approaches and the information they leverage. We also include the general approach of (web) vulnerability scanners as presented in Section 2.3.

We classify the information used by blackbox TTs along two dimensions. First, we distinguish between information accessible prior to testing a specific SuT and information accessible only during testing. For instance, information on vulnerabilities in network stacks, such as those presented by Kohl et al. [Koh20a], are publicly accessible before testing a specific SuT. In contrast, information on crashes encountered by a specific SuT can only be obtained during or after testing. Second, we differentiate between static and dynamic information. Static information refers to information that typically remains

Table 3.1: Blackbox information used by blackbox TTs for test case generation or test guidance in the literature and in this doctoral work. We classify the used information based on the time of accessibility (before testing vs. during testing), and whether the information changes over the course of the test (static vs. dynamic). Next to specific approaches from literature, we also include the general approach of vulnerability scanning as presented in Section 2.3.

	Before Testing	
Static	Vulnerabilities	`ClusterCrash` [*Bor22*], vulnerability scanners
	Heuristics	[Pfr17], vulnerability scanners
	Network traffic	[Gas15], `Palpebratum` (Section 7.5)
	During Testing	
Static	Endpoints	[Esp18, Ren19], `HitM` [*Bor20*], ISuTest®
	Software versions	Vulnerability scanners
Dynamic	Crashes	[Hou12, Gop20, Fer22], `Smevolution` [*Bor23b*]
	Full response	[Dou12, Duc14, Esp18, Ren19, Gop20, Aic21, Sha21, Dra23b, Yin23, Liu24a], `SWaTEval` [*Bor23c*], `HitM` [*Bor20*]
	Response code	[Lin21, Sas21, Kim20]
	Response time	[Kim20]
	Firewall bypass	[App18]
	Network traffic	[Gas15], `Palpebratum` (Section 7.5)
	Side channels	[Spe19], [*Su24*]

unchanged throughout a test, such as the endpoints exposed by a Web Application (WA) or the open ports of an Operational Technology (OT) component. Conversely, *dynamic* information changes during testing, such as the network traffic generated by the SuT.

In Sections 3.1 and 3.2, we provide an overview of the types of information utilized during and before blackbox testing and how this information is leveraged. Details on how the newly presented approaches of this doctoral work utilize the different information is detailed in Section 3.3

3.1 Information Accessible before Testing

Existing blackbox TTs make use of three types of information accessible before testing: (1) published vulnerabilities, (2) heuristics, and (3) network traffic.

First, information on specific vulnerabilities potentially related to the SuT is utilized. The general approach of vulnerability scanners uses information on known vulnerabilities to verify whether the software presumed to be running on the SuT contains any of those vulnerabilities (see Section 2.3).

Other TTs leverage expert knowledge through heuristics that are used for test case generation. For example, this approach is employed by ISuTest® [Pfr17] and by the general approach of vulnerability scanning.

PULSAR [Gas15] utilizes network traffic generated by the SuT prior to the test to guide the testing process. Specifically, PULSAR analyzes a sample of the SuT's network traffic to infer a Markov chain that represents the state machine of the communication protocol in use. The network traffic is leveraged to define templates and rules that represent the format of the network packets. These models are then employed (1) to generate traffic for the specific network protocol, and (2) to guide the testing process.

3.2 Information Accessible during Testing

During testing, additional SuT-specific information becomes available that can guide the testing process. On the one hand, static information on the SuT can be utilized. This includes information on the communication endpoints exposed by the SuT. For example, Esposito et al. and Rennhard et al. [Esp18, Ren19] utilize the known Uniform Resource Locator (URL) endpoints of a WA, while ISuTest® utilizes information on known communication endpoints of an OT component. The approaches targeting WAs utilize the information on available endpoints to implicitly provide it to Web Vulnerability Scanners (WVSs), for them to be able to test all available endpoints [Esp18, Ren19]. Conversely, ISuTest® uses the information on the open ports of an OT component to

determine which test configurations and which test cases to apply. For instance, Simple Network Management Protocol (SNMP) test cases are only sent if the SuT provides an SNMP endpoint, or if they are explicitly requested. While we classify this information as *static*, it is worth noting that the open ports of a SuT may change during testing. For example, one OT component tested during this doctoral work closed a port after encountering three unsuccessful communication attempts on a different port. However, this information is usually static and is thus classified as such.

On the other hand, a SuT exposes dynamic information. For instance, monitoring the SuT for crashes provides valuable information for guiding future test case generation. Householder et al. present an approach to utilize the crash information to guide seed selection and mutation parametrization [Hou12]. Fernandez et al. propose a framework for a grammar-based fuzzer which utilizes crash information to guide the grammar-based test case generation [Fer22]. Similarly, Gopinath et al. utilize crash information together with information on which position of the test case lead to the crash to generate new test cases [Gop20]. Shang et al. utilize information on which test cases were successful in triggering anomalies to guide the mutation of test cases [Sha21].

Several approaches leverage the entire response from the SuT, but they utilize this information in different ways. Doupé et al., Aichernig et al., and Drakonakis et al. [Dou12, Aic21, Dra23b] use the full response of the SuT to automatically derive a state machine, which is then employed for test case generation. In contrast, JARVIS uses the responses to modify test cases of existing WVSs to allow for authenticated requests, amongst others [Esp18, Ren19]. Duchene et al. and Liu et al. use the full response of the SuT to guide the test case generation as follows. Duchene et al. focus on reflected Cross-Site Scripting vulnerabilities in WAs and analyze whether the response of the SuT reflects the full test case or parts of it [Duc14]. Liu et al. also consider WAs and approximate the code coverage based on the strings that are included in the SuT's responses [Liu24a].

Kim et al. utilize the response in combination with the response time to determine whether a test case is deemed interesting [Kim20]. ICPFuzzer and R0fuzz also use the responses to guide the test case generation, but focus exclusively on using the response code rather than the entire response [Lin21, Sas21].

Appelt et al. propose an evolutionary approach for testing WA firewalls, using binary information on whether a test case successfully bypassed the firewall [App18].

As previously mentioned, PULSAR utilizes network traffic in its testing approach [Gas15]. PULSAR leverages network traffic generated before the test to train its model, but it also uses the network traffic generated during testing to query this model. During the test, PULSAR analyzes network traffic to determine the current state of the SuT and thus to decide which message should be sent next.

Additionally, all approaches discussed and categorized above make use of information on the test cases that lead to specific behavior of the SuT. This information is crucial for generating, selecting, or mutating future test cases based on previous observations, or for deriving a state machine of the SuT. However, some approaches, such as scanner++ [Yin23] and the approach by Salem et al. [Sal21], exclusively utilize information on the test cases. These approaches were excluded from our categorization because they do not incorporate additional information available in blackbox testing or consider SuT-specific information.

Additionally, some publications utilize side-channel information for test case generation. Sperl et al. use the power consumption of the hardware running the SuT to assess the interestingness of a test case [Spe19]. FUZZ-E, proposed during the course of this doctoral work, focuses on fuzzing of Field Programmable Gate Arrays (FPGAs) and utilizes voltage measurements to approximate the SuT's behavior [*Su24*].

3.3 Utilized Information Sources

In the following, we detail which information sources are used by the approaches presented in this doctoral work.

`HitM`, a proxy-based approach to transparently improve existing WVSs, utilizes the static information on existing endpoints and the dynamic information on the responses of the SuT to the WVSs' requests. The information on the endpoints is used to explicitly provide it to the WVSs using the URL injection add-on of `HitM`. With this, it influences the test case generation of the WVSs. The responses of the SuT are used, amongst others, to identify whether the requests of the WVSs are authenticated requests. If the requests are not authenticated, the User Login add-on provides the necessary authentication details. With this additional information, the test cases generated by the WVSs are improved, and the future test case generation is influenced. See Chapter 4 and [*Bor20*] for more details.

`ClusterCrash` generalizes information on vulnerabilities revealed by past whitebox and graybox tests, and subsequently defines Vulnerability Anti-Patterns (VAPs). Based on these VAPs, we derive blackbox tests which test for the underlying causes of the past vulnerabilities. Thus, it allows for more targeted blackbox tests. See Chapter 5 or [*Bor22*] for more information.

`Smevolution` proposes an approach to utilize the information on crashed services of the SuT to guide an evolutionary test case generation. With this, it utilizes the dynamic information on which services of the SuT crash for a certain test case during testing. See Section 7.4 or [*Bor23b*] for more details.

`Palpebratum` presents an approach to modeling the behavior of the SuT by using a Hidden Markov Model (HMM). To train the HMM, `Palpebratum` utilizes network traffic recorded before testing. During testing, `Palpebratum` utilizes the dynamic network traffic generated by the SuT to query the HMM. This query is then used to assess the interestingness of a test case. More details on `Palpebratum` are given in Section 7.5.

SWaTEval, the novel framework for Stateful Web Application Testing (SWAT) evaluations, utilizes the responses of the SuT to automatically infer a state machine of the SuT. This state machine is used to guide the fuzzing process. See Section 8.3 or [*Bor23c*] for more details.

3.4 Analysis Conclusions

Our analysis of the information used in blackbox testing suggests the following conclusions:

1. A variety of static and dynamic information, accessible either before or during testing, can be utilized during the blackbox testing process.
2. The novel approaches proposed in this doctoral work do not consider new information sources, but utilize the information in new ways.

4 Industrial Web Security

This chapter presents `Helper-in-the-Middle` (`HitM`), a proxy-based solution aimed at transparently improving the applicability of Web Vulnerability Scanners (WVSs) within the domain of industrial security. We successfully design, implement, and evaluate `HitM`, revealing several previously unknown vulnerabilities in Web Applications (WAs) of Operational Technology (OT) components. Notably, one of these vulnerabilities was assigned a Common Vulnerabilities and Exposures (CVE) identifier [CVE18] with a high severity (Common Vulnerability Scoring System (CVSS) score of 7.5). Moreover, our evaluation shows that `HitM` increases the number of true positive reports, and improves the Uniform Resource Locator (URL) coverage for most of the Systems under Test (SuTs).

Testing the WAs of OT components presents several domain specific challenges to WVSs, such as handling crashes of the SuTs, and monitoring diverse interfaces of the SuT during testing (see also Section 4.1). Moreover, our analysis of prior studies highlights that a considerable portion of the identified limitations of WVSs are concerned with their practical applicability (Section 4.3.2). Thus, we introduce our new approach, `HitM`, designed to address several of these limitations while focusing on an industrial test setting. `HitM` adopts a proxy-based approach, facilitating a transparent enhancement of the WVSs. Since this enhancement is transparent for the WVSs, the WVSs themselves do not need to be altered (see Section 4.3.3). This is achieved by augmenting the functionality of the proxy used within `HitM` through a combination with a containerization of the WVSs and the security testing framework ISuTest® (see Section 2.2.3). Through this integrated approach, `HitM` successfully addresses several applicability challenges of WVSs.

Our evaluation of HitM encompasses both qualitative and quantitative analyses, involving testing five OT components with six WVSs integrated into HitM (Section 4.3.4). The results of this evaluation illustrate that HitM addresses nine of the previously identified limitations. Moreover, HitM improves the number of true positive reports of the WVSs, and also increases their URL coverage. However, consistent with existing literature, HitM also increases the false positive rate for most SuTs. Furthermore, our evaluation reveals several previously unknown vulnerabilities of WVSs within OT components.

4.1 Problem Statement

In addition to features that exclusively support the functionality of an OT component, many OT components offer supplementary features. These features are designed, for instance, to enhance user experience or to offer additional information to users of the OT component. An example of this additional functionality is WAs that run locally on an OT component. At the very least, such a WA usually offers information on the current status of the OT component. However, numerous WAs offer additional functionalities such as means for configuring the OT component or updating its firmware, allowing for more advanced user interaction [*Pfr19b*]. Considering that all interfaces of an OT component serve as potential entry point for attackers, a locally running WA is no exception. This is particularly true when the WA offers advanced features such as configuration or file upload. With this, attackers could potentially exploit the WA to attack the OT component and the physical process that is connected with the OT component. Thus, a WA provided by an OT component needs to be tested for vulnerabilities and these vulnerabilities need to be fixed.

Out of the diverse means to test a WA for vulnerabilities (see Section 2.3), this dissertation focuses on tests with blackbox WVSs. As detailed in Section 2.3, WVSs are automated Test Tools (TTs) which scan a WA for vulnerabilities by crawling the WA, finding input possibilities, and providing certain inputs to the WA [Dou10]. With most of the existing WVSs being designed to test general-purpose WAs, they show several limitations when used in industrial test settings [*Pfr19b*]. For example, typical WVSs often struggle to handle a

crash of the tested WA. In test scenarios focusing on Information Technology (IT), a tested WA usually has access to extensive resources that allow it to efficiently handle the high workloads generated by a TT. However, in industrial test settings, WAs are resource-constrained and thus the WA, and sometimes the entire OT component, may crash due to a test's load or specific test cases.

Based on a detailed analysis of the limitations of WVSs, especially with respect to their applicability in industrial test settings (see Section 4.3.2), we suggest a proxy-based solution which addresses these limitations while considering the challenges of industrial security testing as formulated in Section 1.3. Since our solution helps to improve the performance of WVSs transparently while also acting as a Machine-in-the-Middle (MitM), we call this approach *Helper-in-the-Middle* (`HitM`).

4.2 Contributions

This doctoral work makes the following two main contributions to the domain of blackbox testing of WAs on OT components.

Contribution 1. *Analysis of the limitations of blackbox WVSs with respect to their applicability to OT component security testing.*

Our analysis of prior studies reveals that a considerable portion of the identified limitations are concerned with the practical applicability of WVSs. Especially, it shows that aiming to use WVSs in industrial test settings adds additional challenges in applicability.

We perform this analysis of the current state-of-the-art in two steps. On the one hand, we analyze literature to identify general limitations of WVSs that have already been mentioned by previous studies, including a preliminary study conducted during the course of this doctoral work [*Pfr19b*]. We collect these limitations and cluster them. On the other hand, we analyze whether and how existing approaches from literature address these limitations. With this, we identify the current limitations of WVSs, especially regarding their

application in industrial test settings. Preliminary analyses and test results were published in the journal *at-Automatisierungstechnik* [*Pfr19b*], and the literature review and analysis were presented at *SECRYPT 2020* [*Bor20*]. The results of the analysis are presented in Section 4.3.2, which sets a special focus on setting the results into the context of this doctoral work.

Contribution 2. *Implementation and evaluation of the proxy-based solution* `HitM` *which transparently facilitates the use of blackbox WVSs in industrial test settings.*

`HitM` consists of three parts: (1) a proxy which is located in between the WVS and the SuT, (2) the security testing framework ISuTest® (see Section 2.2.3), and (3) a containerization environment for the WVSs. We evaluate `HitM` in a qualitative and a quantitative evaluation. The qualitative evaluation demonstrates that `HitM` addresses three limitations with respect to the methodology of WVSs, and six limitations that are concerned with the applicability of WVSs (Section 4.3.4.2). The quantitative evaluation is conducted by integrating six WVSs into `HitM`, and running each of them against five OT components, while varying the level of support provided by `HitM`. It shows that an increased support by `HitM` leads to more true positive findings of the WVSs, and to an increased URL coverage (Section 4.3.4.3).

`HitM` considers several of the challenges of industrial security testing that are formulated in Section 1.3. Especially, it considers the blackbox use case (Challenge 1 (Blackbox Testing)), and monitors all communication interfaces of the SuT (Challenge 3 (Insufficient Observations)). Furthermore, `HitM` is transparent for the WVSs (Challenge 5 (Choice of Testing Tool)), and injects additional information to the traffic of the WVS (Challenge 2 (Missing Information)).

First steps towards `HitM` were conducted during the Master's thesis of the author of this doctoral work [*Bor18*], which were subsequently published in the journal at-Automatisierung [*Pfr19b*]. The approach of `HitM` as well as the corresponding evaluations are based on the Master's thesis by Albrecht Weiche that was supervised during the course of this doctoral work [*Wei19*],

and have been published at *SECRYPT 2020* [*Bor20*]. During the work on HitM, several previously unknown vulnerabilities were revealed, including a Denial of Service (DoS) in BC_{ex} which was assigned a CVE identifier [CVE18].

The implementation of HitM's proxy is based on mitmproxy[1]. In order to provide the additional functionality required by HitM, several new addons were implemented and the corresponding source code was published[2][3].

4.3 Helper-in-the-Middle

This section details the work with respect to the transparent improvement of WVSs in the context of industrial security testing conducted during the course of this doctoral work. First, the methodology of HitM is detailed in Section 4.3.1. Then, the analysis is presented (Section 4.3.2), and the approach of HitM is described (Section 4.3.3). The qualitative and quantitative evaluation of HitM is presented in Section 4.3.4, while Section 4.4 gives additional insights into testing BC_{ex}. The content of this section is based on the respective publications produced during the course of these doctoral work [*Pfr19b, Bor20*], as well as the Master's thesis by Albrecht Weiche which was supervised during this doctoral work [*Wei19*].

4.3.1 Methodology

In order to approach the objective to transparently improve the performance of WVSs, we perform the following steps.

1 Conduct a literature review to identify the current limitations of WVSs in general, and with respect to OT components (Section 4.3.2).

2 Cluster these limitations and analyze which of these limitations are addressed by the literature (Section 4.3.2).

[1] https://mitmproxy.org/
[2] https://github.com/mitmproxy/mitmproxy/pull/3961
[3] https://github.com/mitmproxy/mitmproxy/pull/3962

3 Design and implement HitM, which addresses several of the limitations that are not yet addressed by literature (Section 4.3.3).

4 Integrate six WVSs into HitM.

5 Evaluate HitM qualitatively based on our experiments with five OT components (Section 4.3.4.2), focussing on the limitations of WVSs that are addressed by HitM.

6 Evaluate HitM quantitatively by using the integrated WVSs to test four OT components and measuring number of true positive reports of the WVSs, the false positive rate, and the total runtime of the tests.

Consistent with the approach of all evaluations carried out in this doctoral work involving OT components, we follow a responsible disclosure policy for the revealed vulnerabilities (see Section 2.2.2.2).

4.3.2 Analysis

We conduct a literature review with the objective to analyze which limitations of WVSs have already identified by literature and which of those are addressed by literature [*Bor20*]. This review includes a paper, published as part of this doctoral work, which discusses of WVSs with respect to their applicability in industrial test settings [*Pfr19b*]. Note that this analysis was conducted prior to the work on HitM, which is why it only includes publications from before 2019. Section 4.5 discusses recent publications related to HitM. The analysis presented in this section is based on 15 papers which compare the performance of several blackbox WVSs [Fon07b, McA08, Bau10, Dou10, Kho11, Fer11, Dou12, Ala12, Sut13, Mak15, Idr17, Veg17, Dee18, Esp18, Pfr19b]. In total, these papers evaluated 25 WVSs using 25 SuTs.

As a first step, we analyze which WVSs and which SuTs were used by the papers. Figure 4.1 illustrates the relationship between the utilized WVSs and the number of papers in which they are referenced. The diameter of the circles corresponds to the number of SuTs the respective WVS has been evaluated against in total. Additionally, the color of the circles represents the licence under which the respective WVS is published. For example, the open source

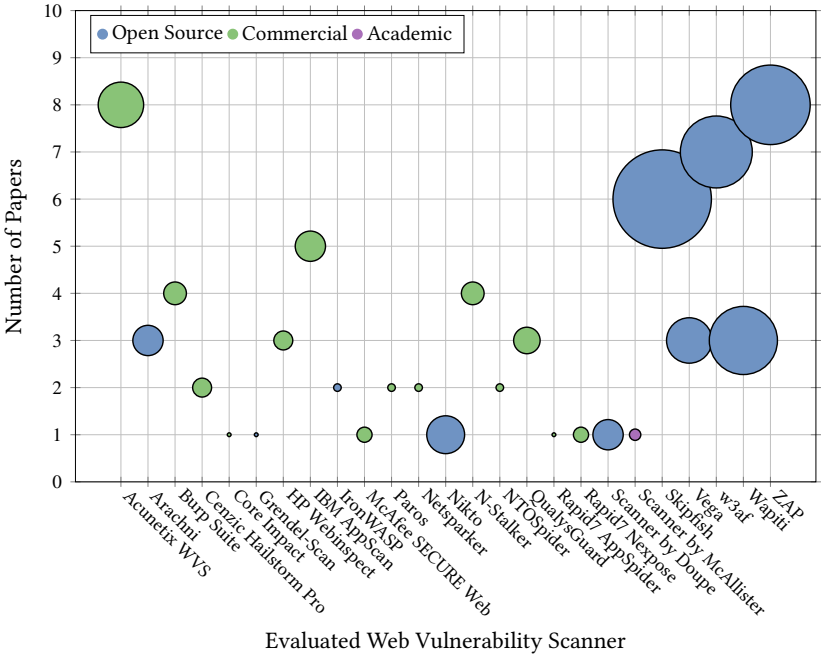


Figure 4.1: Frequency of WVSs that have been evaluated by the reviewed literature, and the license they were published under. For each WVS, the diameter of the respective circle corresponds to the number of SuTs the WVS was evaluated against. Various WVS are considered by literature, while open source scanners are evaluated more often.

WVS Arachni is represented by a blue circle at $y = 3$, showing that this WVS was used in three of the considered papers. The diameter of the circle corresponds to the number eight, which is the number of SuTs Arachni was tested against in the considered publications in total. Note that the evaluation of Fonseca et al. is not integrated into this figure since the authors do not reveal which scanners they evaluated [Fon07b].

The plot shows that open source scanners have been tested more thoroughly than commercial scanners. We believe this discrepancy is mainly due to financial reasons. Nevertheless, the choice of a WVS for productive testing is also driven by financial considerations. Another point revealed by the

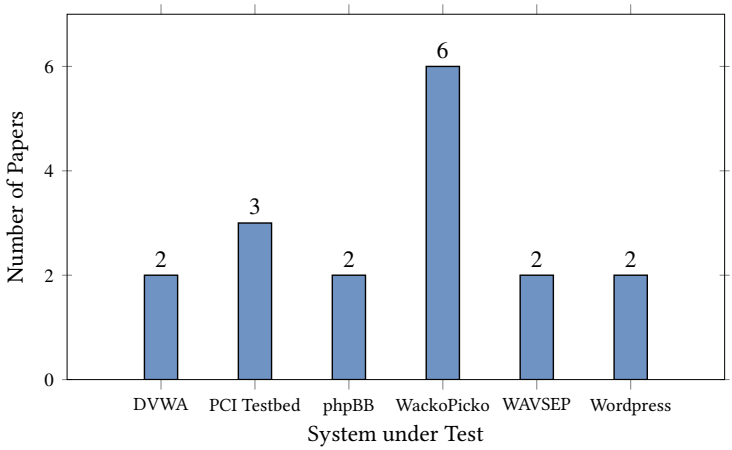


Figure 4.2: SuTs that have been used more than once for the evaluations of WVSs. WackoPicko by Doupé et al. [Dou10] is used the most.

review is the large amount of WVSs that are known enough to be considered in such an evaluation. This point is also supported by the extensive lists of WVSs that is presented by the Open Worldwide Application Security Project (OWASP) [OWA20]. This highlights the complexity of selecting an appropriate WVS for testing (Challenge 5 (Choice of Testing Tool)).

The SuTs used for the evaluations are also manifold. In total, 25 SuTs were used, of which those SuTs that were used by more than one paper are displayed in Figure 4.2. For this figure, papers using the same SuT with different versions have been aggregated. Striking is the frequent use of WackoPicko, as six of the considered papers use WackoPicko for their evaluation. WackoPicko is a WA that has been presented by Doupé et al. as a test bed for WVSs [Dou10]. Except for Wordpress and phpBB, all the SuTs are willful vulnerable WAs.

4.3.2.1 Identified Limitations

As a basis for a deeper understanding of known limitations of WVSs, we collect and classify the limitations that were identified by literature. Table 4.1 shows the results of this collection and our classification. We classify the

Table 4.1: Classification of the limitations of blackbox WVSs identified by the literature as well as publications that address the respective limitation (based on [*Bor20*]). The fourth column identifies the limitations that are addressed by HitM.

Limitations	Identified	Addressed	HitM
Vulnerability Classes			
Stored XSS Injection	[Bau10, Dou10, Ala12]	[Li16, Gal10]	
Persistent SQL Injection	[Bau10, Dou10, Kho11]	-	-
Remote File Inclusion	[Mak15, Dou10]	-	-
Local File Inclusion	[Idr17]	-	-
Path Disclosure	[Dou10]	-	-
Methodology			
Crawling	[Esp18, Dou10, Idr17, Fer11, Kho11]	[McA08, Esp18, Dou12]	✓
Fuzzing	[Idr17]	[McA08, Dou12]	-
Periodically Changing Content	[*Pfr19b*]	-	✓
Attack Code Selection	[Fer11, Kho11]	-	-
User Login	[Dou10, Kho11],[*Pfr19b*]	[McA08, Esp18]	✓
Second Order Vulnerabilities	[Bau10, Dou10]	-	-
Application Logic	[Dou10]	[Dee18]	-
JavaScript / Flash / HTML5	[Dou10, Idr17, Dou12]	-	-
Categorization of Findings	[Kho11]	-	-
Applicability			
Parameter Understandability	[Esp18]	[Esp18]	-
Reproducibility	[*Pfr19b*]	-	✓
Conduction of Single Tests	[*Pfr19b*]	-	✓
Load Reduction	[*Pfr19b*]	-	✓
Pause and Resume a Scan	[*Pfr19b*]	-	✓
Behavior in Case of an Error	[*Pfr19b*]	-	✓
Different Scanners Necessary	[Esp18, Idr17],[*Pfr19b*]	[Esp18]	✓
Runtime	[Dou10, Sut13],[*Pfr19b*]	-	-

limitations that were identified by literature into the following four clusters: Vulnerability Classes, Methodology, Applicability, and SUT. The following explains the nature of these clusters as well some of the limitations categorized within this cluster.

Vulnerability Classes This cluster encapsulates specific vulnerability classes WVSs have demonstrated difficulty in addressing. For instance, multiple studies have highlighted the challenges WVSs face in effectively identifying Stored XSS and Stored SQL vulnerabilities [Bau10, Dou10, Kho11, Ala12].

Methodology In this cluster, we group those limitations that pertain the general approach of WVSs. The cluster addresses various aspects of WVSs, such as *Crawling* and *Fuzzing* capabilities, handling of *Periodically Changing Content*, and *Application Logic* limitations. Application Logic limitations encompass the identification and detection of application-specific vulnerabilities.

The limitation concerning *Periodically Changing Content* pertains to WAs featuring regularly updated content, such as the current time. This dynamic content can pose challenges to WVSs during probing for injections. To probe for injections, a two-step process is typically employed. First, the WVS attempts to inject code into the WA. Second, the WVS assesses whether there are changes in the WA's content subsequent to the injection attempt. If the WA's content exhibits alterations independent from the WVS, the WVS regularly assumes that its code injection was successful. However, in reality, only the periodically changing content, such as the current time, may have changed.

Applicability This cluster includes limitations of WVSs concerning their practical applicability, especially when using them to test OT components. *Load Reduction* refers to challenges in reducing the load that is sent to the SuTs. Particularly when testing OT components, there is an increased risk of SuT crashes due to high load. *Behavior in Case of an Error* is closely linked to the previous limitation. If the SuT crashes and the WVS fails to detect this, the WVS will continue sending its probes fruitlessly. Furthermore, literature suggests that employing different scanners is necessary to achieve comprehensive

coverage. Additionally, the *Runtime* of a WVS significantly influences the frequency of SuT testing. Note that especially the limitations in this cluster are more prominent when testing an OT component instead of a classic WA.

SUT This cluster encompasses the limitations of the SuTs utilized during the evaluations. In particular, it encompasses two limitations: (1) the possibility of unindented vulnerabilities in the SuTs [Mak15], and (2) the possibility of outdated vulnerabilities in the SuTs [Sut13]. Although these limitations do not pertain directly the WVSs, they are frequently mentioned in literature. Consequently, we included these limitations in our categorization. However, we will refrain from delving further into this cluster and instead concentrate our analysis on the limitations directly associated with the WVSs.

4.3.2.2 Achieved Improvements

Building upon the identified limitations, researchers have endeavored to mitigate these limitations through various approaches. The following descriptions highlight publications in this domain and correlates them with the limitations described in the previous section (see Table 4.1).

McAllister et al. propose a WVS enhanced by techniques aimed at detecting more entry points for scanning [McA08]. This includes strategies such as recording and replaying user interactions, simultaneously fuzzing various input forms to broaden test coverage, and employing stateful fuzzing. Note that this stateful fuzzing assumes that the fuzzer is able to control the WA. With this approach, McAllister et al. address the limitations regarding *Crawling*, *Fuzzing*, and *User Login*. The experiments conducted by McAllister et al. suggest that their solution identifies more entry points and bugs compared to other evaluated WVSs.

Doupé et al. introduce a state-aware WVS that automatically builds a model of the internal states of a WA by sending packets and interpreting the corresponding responses from the SuT [Dou12]. This model is then utilized to derive and test new entry points as well as to generate new payloads for the included fuzzer (see also Section 2.5). The newly proposed WVS addresses

the limitations of *Crawling* and *Fuzzing*. The evaluation of Doupé et al. indicates that their approach improves code coverage and the effectiveness of vulnerability tests.

Deepa et al. present a WVS that leverages the data and control flow of a WA to construct a model of its behavior [Dee18]. This model targets logic vulnerabilities such as parameter manipulation, access control, and workflow bypass vulnerabilities. With this, the authors address the limitation regarding *Application Logic*. Deepa et al. demonstrate that their approach achieves high precision and a high true positive rate.

In contrast to the aforementioned solutions, Esposito et al. introduce JARVIS, a proxy-based solution aimed at enhancing the capabilities of existing WVSs rather than creating a new one [Esp18]. The proxy enhances crawling capabilities and authentication of the WVS against the target. Crawling is improved by injecting known URLs into locations where the WVS is expected to crawl, and the proxy authenticates initially unauthenticated requests sent by the WVS. Their evaluation illustrates that JARVIS increases the number of detected vulnerabilities while simultaneously increasing the false positive rate for some of the WVSs. Refer to Section 4.5 for a comparison of HitM and JARVIS.

Furthermore, several approaches addressing specific vulnerability classes have been presented, and corresponding tools have been published. For example, the publications by Li [Li16] and Galán et al. [Gal10] as well as XSSer[1] address Stored XSS Injection vulnerabilities, while sqlmap[2] tests for Stored SQL Injections.

[1] https://github.com/epsylon/xsser
[2] https://sqlmap.org/

4.3.2.3 Analysis Conclusions

The previous section reported the results of a literature review which analyzed publications regarding blackbox WVSs before the publication of HitM. In summary, 15 publications have been analyzed, which evaluated 25 WVSs using 25 SuTs. The key takeaways of this analysis are as follows.

1 Various WVSs are relevant enough to be considered in several studies, while these studies also show that there is no WVS performing better than all other WVSs. This makes the choice of an appropriate WVS harder (see also Challenge 5 (Choice of Testing Tool)).

2 The analysis shows that the limitations of WVSs that have been identified by literature can be clustered in the following clusters: *Vulnerability Classes*, *Methodology*, *Applicability*, and *SUT*.

3 Most approaches from literature address limitations from the clusters *Vulnerability Classes* and *Methodology*, while there are only few publications concerned with the *Applicability* of WVSs. However, this cluster is of high importance regarding the adoption of blackbox WVSs. Furthermore, several of these challenges are especially prominent when testing OT components, such as the *Behavior in Case of an Error*.

4.3.3 Approach

The approach of HitM is to improve the performance of arbitrary blackbox WVSs by providing a transparent *Helper-in-the-Middle*. This section introduces the approach of HitM as well as the details on the extensions that we designed and implemented. HitM aims to address several limitations of WVSs with

respect to their *Applicability*, namely *Crawling*, *Periodically Changing Content*, *User Login*, *Reproducibility*, *Conduction of Single Tests*, *Load Reduction*, *Pause and Resume a Scan*, *Behavior in Case of an Error*, and *Different Scanners Necessary*.

4.3.3.1 Fundamental Approach

The fundamental approach of HitM is to combine the advantages of a proxy, ISuTest®, and containers to support WVSs during their testing. Figure 4.3 shows how these three parts of HitM are connected. Even though the three components of HitM are logically different, they run on the same computer, the Test Device (TD).

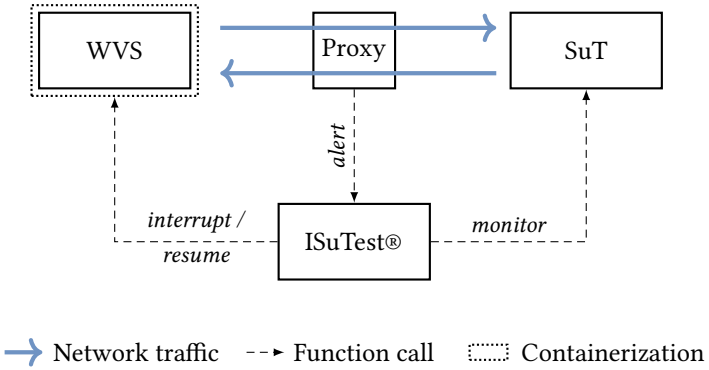


Figure 4.3: Overview of HitM, based on [*Bor20*]. It consists of the WVSs running in containers, ISuTest®, and a proxy which is located between the WVSs and the SuT. The proxy analyses and manipulates the traffic between the WVSs and the SuT.

Proxy The proxy resides between the WVSs and the SuT, enabling it to intercept, interpret, and modify the packets exchanged between these two. The proxy's extensions are implemented as add-ons, which will be elaborated upon later in this section (Section 4.3.3.2).

Containerization The WVSs are run within containers, offering two primary advantages: Firstly, it facilitates the pausing and resuming of WVSs' activities. Secondly, scans can be executed in a more reproducible manner, as the WVSs can be reset to a known state by resetting the container.

ISuTest® The third component of HitM, ISuTest® (see Section 2.2.3), is utilized for orchestrating the WVSs and monitoring the SuT. To allow for the orchestration of the WVSs, we implement the capability to integrate external tools into ISuTest® [*Pfr19b*]. Moreover, ISuTest® aids in monitoring, supported by the watchdog add-on of the proxy (see Section 4.3.3.2). In short, this watchdog alerts ISuTest® if the SuT appears to be unresponsive to HTTP requests, prompting ISuTest® to initiate a monitoring cycle to verify the SuT's status.

In summary, we present a comprehensive integration of blackbox WVSs into an industrial security testing system without requiring modifications to the WVSs themselves (Challenge 1 (Blackbox Testing), Challenge 2 (Missing Information), Challenge 3 (Insufficient Observations), Challenge 5 (Choice of Testing Tool)). This integration encompasses an extension of ISuTest®, deploying the WVSs in containers, and implementing new proxy add-ons. We have made these add-ons publicly available for further research[1]. It is worth noting that the proxy can also function independently, allowing many of the enhancements to be utilized simply by installing and configuring the proxy, without the need to setup ISuTest® or a containerization.

4.3.3.2 Features

The following details the features of HitM. These features address several of the limitations of WVSs.

Containerization and Transparent Proxy Mode We opted to utilize LXD[2] for building the containers. Our decision was primarily influenced by the ease of configuration and manipulation of packet forwarding for virtual network

[1] https://github.com/mitmproxy/mitmproxy/pull/3961
https://github.com/mitmproxy/mitmproxy/pull/3962
[2] https://linuxcontainers.org/

bridges provided by LXD. This capability is essential for enabling the transparent proxy mode, which ensures that HitM can be used both with proxy-aware and non-proxy-aware WVSs. The challenge in implementing the transparent proxy mode lies in distinguishing between packets that should be routed to the proxy and those that should not. Leveraging the containers in which the WVSs operate, we introduced a virtual network bridge on the TD to direct all container traffic to the proxy. This enables HitM to be utilized even with non-proxy-aware WVSs.

Proxy As foundation for the proxy integrated into HitM, we utilize mitmproxy[1]. We selected this proxy due to its extensibility, its open source license, and its support for HTTP and HTTPS. Furthermore, mitmproxy is fairly commonly used in research (see e.g. [Yin23, Dra23b]), which simplifies adoption and comparison.

The following features are implemented as add-ons for the proxy. Figure 4.4 shows a graphical representation of the add-ons, their interaction with the network traffic, and the information they require. In the figure, each add-on is represented by a rectangle including the name of the corresponding add-on. Each add-on covers either one or both directions of the network traffic that is flowing trough the proxy. For example, the watchdog analyzes both directions to search for indicators of a crash of the SuT or the WVS. In contrast, the mapping add-on has the task to replace parts of the SuT's responses and thus only interacts with the traffic coming from the SuT. Some add-ons need additional information, which is represented by the rounded rectangles in the figure. For example, the mapping add-on needs information on which parts of the SuT's responses should be replaced. Note that the logging add-on is excluded from the figure to simplify the visualization.

Watchdog The watchdog add-on addresses the limitation of handling a SuT's crashes during testing (*Behavior in Case of an Error*). It achieves this by monitoring the SuT's communication with the WVS for errors, and alerting ISuTest® in the case of an error. The sequence of actions performed when the watchdog detects an error is shown in Figure 4.5 and described in the

[1] https://mitmproxy.org/

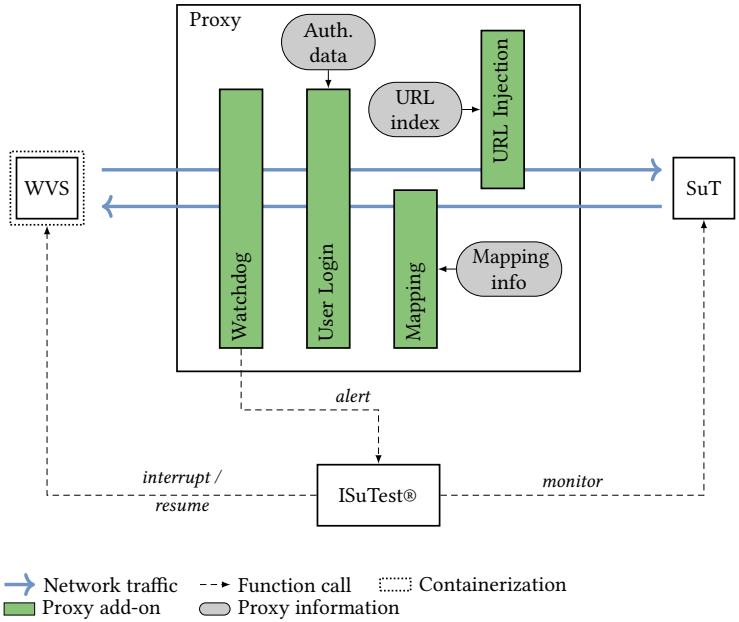


Figure 4.4: Detailed view of HitM, focusing on the add-ons of the proxy: the watchdog, user login, mapping, and URL injection. Each add-on is represented by one rectangle which interacts with one or both directions of the network traffic flowing through the proxy. Some of the add-ons need additional information, represented by the rounded rectangles.

following. The watchdog add-on is implemented as an add-on of `mitmproxy` and uses the error method provided by `mitmproxy`. This method is invoked if an error occurs during the transmission or reception of an HTTP packet. Acting as a bridge between `mitmproxy` and ISuTest®, the watchdog alerts ISuTest® of any detected errors. Upon receiving an alert, ISuTest® interrupts the WVS, which is possible due to the containerization of the WVS. ISuTest® then initiates a monitoring cycle, which involves checking whether the SuT responds to requests from different protocols, such as Internet Control Message Protocol (ICMP) ping packets. If the SuT fails to respond, ISuTest® restarts the SuT and verifies its status again. Subsequently, the WVS is resumed, and the test continues. Note that this entire process remains transparent to the WVS.

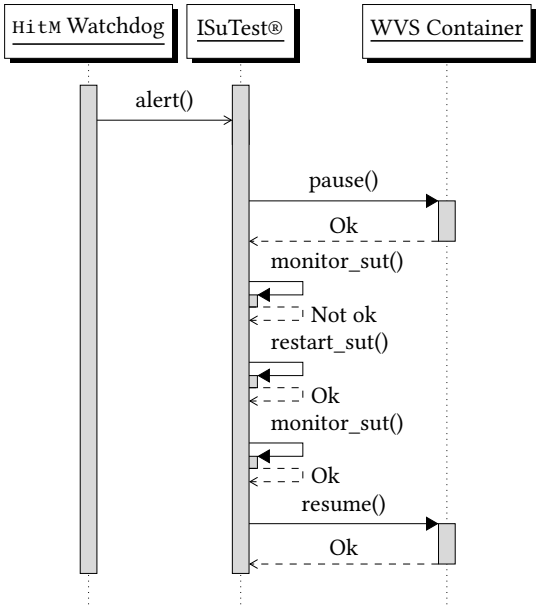


Figure 4.5: Sequence diagram showing the functionality of the watchdog add-on of the proxy within HitM. The watchdog alerts ISuTest® as soon as a communication error is detected. ISuTest® then pauses the WVS and runs a monitoring cycle, checking the responsiveness of the SuT. Note that this monitoring is run via the network. In the scenario shown, the SuT does not respond to the monitoring and is thus restarted by ISuTest® through a power cycle. After the restart, the SuT becomes responsive again and the test can continue.

Mapping The mapping add-on aims to mitigate false positives resulting from changing content (*Periodically Changing Content*). It offers the capability to replace or delete content on a web page, such as the current time. For this, content identified by a CSS selector is replaced with fixed HTML code.

User Login We propose two distinct approaches to address the limitation regarding user login and authentication (*User Login*).

The first approach involves implementing various authentication schemes and executing the authentication process when necessary. This enables transparent authentication of the WVS's requests. However, one drawback of this approach is its lack of access to a browser instance, preventing a full rendering and execution of the web page.

To overcome this limitation, the second approach incorporates Selenium[1]. Selenium provides a browser instance, allowing for simulating user login actions. Subsequently, the login information, such as cookies or session IDs, can be transferred to the WVS's requests.

Both approaches share the common requirement of first detecting whether a request is authenticated or not. This automated detection is conduced for both WVS requests and SuT responses.

URL Injection To enhance the *Crawling* capabilities of the WVSs, we introduce the URL injection add-on. This add-on injects URLs into the responses sent by the SuT. As demonstrated by Esposito et al., this injection can improve the WVS's ability to discover more links and pages within the WA significantly [Esp18]. Esposito et al. suggest four places to inject the additional URL information into:

1 robots.txt

2 sitemap.xml

3 landing page of the WA

4 index page of the WA

We adopt these suggestions and inject the URLs in these places. Nevertheless, we choose a more thorough approach to URL extraction than Esposito et al. [Esp18]. While Esposito et al. propose to either use whitebox endpoint extraction, if applicable, or to use the results of the best crawler (Arachni in their case), we use the aggregated crawling results of all used WVSs. These crawling results can be collected automatically by using `HitM`. We initiated

[1] https://selenium.dev/

automated scans of all WVSs against a SuT and recorded all requests and responses made. Subsequently, we extracted all requests to which the SuT responded with a non-error code, thereby constructing the URL index automatically. This index is then utilized for all WVSs, increasing the number of possibly detected URL endpoints.

Logging In order to effectively verify and analyse a vulnerability, it is important to have a comprehensive record of the requests and responses exchanged between the WVS and the SuT. The logging add-on fulfills this requirement by logging every response and request observed by the proxy.

4.3.3.3 Approach Conclusions

This section has introduced the approach of `HitM`. Its main insights and key takeaways are as follows:

1 `HitM` is a proxy-based approach that combines a proxy in between the WVSs and the SuT, a containerization of the WVSs, and ISuTest®.

2 By leveraging the advantages of these three approaches, `HitM` addresses several challenges that WVSs face when they are examining OT components.

3 With its proxy-based structure, `HitM` is transparent for the WVS.

4.3.4 Evaluation

To assess the impact of HitM, we conduct both qualitative and quantitative evaluations. The following outlines the evaluation setting in Section 4.3.4.1, presents and discusses the outcomes of the qualitative evaluation in Section 4.3.4.2, and presents and discusses the findings of the quantitative evaluation in Section 4.3.4.3.

In summary, the qualitative evaluation indicates that HitM effectively addresses several limitations of WVSs, including *Crawling*, *Periodically Changing Content*, *User Login*, *Reproducibility*, *Pause and Resume a Scan*, *Conduction of Single Tests*, *Load Reduction*, *Behavior in Case of an Error*, and *Different Scanners Necessary*. Meanwhile, the quantitative evaluation demonstrates an increase of true positive reports of the WVSs and improved URL coverage. Notably, for one of the SuTs, conducting a full scan was only feasible with the aid of HitM, which successfully handled a reproducible crash of the SuT during the scan.

4.3.4.1 Evaluation Setting

The evaluation of HitM is based on six WVSs and five OT components. As our evaluation focuses on evaluating the impact of HitM rather than the resilience of specific OT components, we do not disclose the manufacturers of the used OT components. Instead, we describe the device class and relevant properties of each of the OT components to give an impression of the used OT components. Nevertheless, we disclosed the vulnerabilities found during the evaluation to the respective manufacturers. The following paragraphs give an overview of the WVSs, OT components, and configurations of HitM used during the evaluation.

Web Vulnerability Scanners

In order to give insights on HitM's impact on WVSs, we select six open source WVSs to be integrated into HitM and to be used during the evaluation. These six WVSs are selected based on their popularity observed in evaluations from

Table 4.2: WVSs that are integrated into HitM and are used for the evaluation.

WVS	Version	Reference
Arachni	1.5.1	https://arachni-scanner.com/
Nikto	2.1	https://github.com/sullo/nikto
Skipfish	2.10	https://github.com/spinkham/skipfish
Vega	1.0	https://subgraph.com/vega/
Wapiti	commit 0bf7g7	https://git.code.sf.net/p/wapiti/git
ZAP	2.8.0	https://www.zaproxy.org/

Table 4.3: SuTs used for the evaluation, including the special challenges they pose for WVSs.

SuT	Device Type	Challenge
SuT1	PROFINET bus coupler	Vulnerability of the WA that crashes the SuT
SuT2	OPC UA Gateway	Form login with changing form field identifiers
SuT3	Firewall	Only allows HTTPS and login only requires password (no username)
SuT4	PROFINET bus coupler	One page, dynamically loaded WA
SuT5	Thermometer	WA displays current measurements which change regularly

previous studies (see Figure 4.1). Table 4.2 gives an overview of the selected WVSs as well as the specific versions of the WVSs used for this evaluation. A detailed description of the WVSs can be found in the publications this section is based on [*Pfr19b*, *Bor20*].

Systems under Test

We choose five OT components as SuTs for the evaluation. With this, we can directly analyze how HitM impacts the performance of the WVSs when targeting OT components. In the following, the SuTs are presented, while Table 4.3 gives an overview of the used SuTs as well as the specific challenges they pose for WVSs.

SuT1 (PROFINET Bus Coupler) SuT1 is a PROFINET bus coupler, just like BC_{ex}, and such has the task to translate commands between PROFINET and digital I/O (see Section 1.6). Moreover, it features a WA which allows for a

configuration of SuT1 and uses HTTP Basic Authentication. This WA includes a vulnerability that, when triggered, leads to a crash of SuT1, requiring a restart of SuT1 to be fully functional again. With this, SuT1 poses a challenge for the WVSs, as they usually do not detect such a crash and cannot automatically restart the SuT after a crash.

SuT2 (OPC UA Gateway) SuT2 serves as a gateway, connecting various communication protocols to OPC Unified Architecture (OPC UA). Its WA implements a form-based login with changing identifiers for the login input fields. This complicates access to the WA for WVSs that expect static identifiers for the login fields.

SuT3 (Firewall) SuT3 is an industrial firewall and thus incorporates the functionality to restrict traffic between two networks. It also features a WA, which permits only HTTPS connections and requires solely a password for user authentication, instead of a username and a password. This special login requirement can prevent WVSs from accessing the WA, as they typically expect both a username and a password field.

SuT4 (PROFINET Bus Coupler) SuT4 is, similar to SuT1 and BC_{ex}, a PROFINET bus coupler. The provided WA is constructed in a different way than the WAs of the other SuTs. It initially serves a JavaScript program which then loads the actual user interface. This process complicates the utilization of a proxy to analyze and modify the network packets effectively.

SuT5 (Thermometer) SuT5 is a sensor measuring the current temperature. These measurements are then shared over various communication protocols such as Message Queuing Telemetry Transport (MQTT), Simple Network Management Protocol (SNMP), File Transfer Protocol (FTP), and mail. Moreover, the measurements are displayed on the local WA trough a dynamically updated graph and a feed. Such periodically changing content can lead to misunderstandings by WVSs since the WVSs might assume that they successfully changed the content of a WA, while in reality only the current temperature was updated in the background (see Section 4.3.2.1).

HitM Configurations

As presented in Section 4.3.3, HitM is designed and implemented based on various add-ons. In order to evaluate the impact of these add-ons, we define seven configurations of HitM that include different subsets of the add-ons (see Table 4.4). These configurations include the following two base cases: (1) running the WVSs without HitM (denoted as *bare*), and (2) *virtualized*, where the WVSs are containerized and configured by ISuTest®, without using the proxy and its add-ons. The remaining configurations represent a full application of HitM including an increasing subset of add-ons.

4.3.4.2 Qualitative Evaluation

HitM enhances the capabilities of WVSs transparently through three key mechanisms: proxy add-ons, containerization, and ISuTest®. Below, we detail and discuss qualitatively how these mechanisms help to address a subset of the limitations identified above (Section 4.3.2). In summary, we show how HitM addresses the limitations of *Crawling*, *Periodically Changing Content*, *User*

Table 4.4: Configurations of HitM used for the evaluation. *bare* corresponds to an execution of the WVSs without HitM, *virtualized* to WVSs running in a container, and all other configurations to an utilization of HitM, on using an increasing subset of add-ons.

Configuration	Containerization	Proxy	Logging	Watchdog	User Login	URL Injection
bare	-	-	-	-	-	-
virtualized	✓	-	-	-	-	-
proxy	✓	✓	✓	-	-	-
HitM_W	✓	✓	✓	✓	-	-
HitM_WL	✓	✓	✓	✓	✓	-
HitM_WI	✓	✓	✓	✓	-	✓
HitM_WLI	✓	✓	✓	✓	✓	✓

Login, *Reproducibility*, *Pause and Resume a Scan*, *Conduction of Single Tests*, *Load Reduction*, *Behavior in Case of an Error*, and *Different Scanners Necessary*(see Section 4.3.2 for explanations on these limitations).

Crawling The main mechanism that addresses the limitations with respect to crawling is the URL injection add-on of the proxy (see Section 4.3.3.2). To create the URL index, we conduct scans with all used WVSs and save each URL that leads to a response of the SuT that does not include an error code. With this approach, we share the knowledge and approaches of the different WVSs between them. However, in certain scenarios, this approach my lead to unnecessary URLs. For instance, if a WVS randomly modifies a URL parameter, and the WA consistently responds with a non-error response to these requests, each of these URL variations will be saved and injected later. Nevertheless, our quantitative evaluation shows that the injection of crawled URLs increases the URL the WVSs access during their test (see Section 4.3.4.3).

Periodically Changing Content This limitation is addressed by the Mapping add-on of the proxy, which facilitates the replacement or removal of periodically changing content that could potentially mislead a WVS. The current implementation requires manual user interaction to identify this periodically changing content. A user needs to identify the periodically changing content and needs to specify its CSS selector. The efficacy of the Mapping add-on was validated through an evaluation using SuT5 and ZAP. As stated, the WA of SuT5 includes an XML feed showing the measured temperatures (see Section 4.3.4.1). Once a minute, this feed is updated and ZAP occasionally interprets the changes of the WA's content incorrectly as a successful injection attempt. However, when using the Mapping add-on of `HitM`, these false positive findings are fully eliminated.

User Login Some SuTs grant logged-in users access to more features than unauthenticated users. Despite most WVSs supporting various authentication mechanisms, they may struggle with authentication against certain OT components. Amongst others, this is based on the observation that many OT components use specialized authentication mechanisms which WVSs usually do not encounter in IT WAs. To aid WVSs during scanning, two Login add-ons have been developed for `HitM`. One of these add-ons, the Selenium add-on,

is of special help with the OT components. First, a user records a manual authentication process with Selenium. This process is required once for each SuT. Afterwards, this recording is loaded into the add-on and can then be used to enhance the scanning capabilities of all used WVSs.

Our evaluation shows that using the Selenium add-on enables WVSs to scan WAs requiring only a password (such as SuT3) or those with dynamically changing session IDs (such as SuT5). However, the add-on encounters difficulties with WAs utilizing background logic relying on regular calls to check the authentication status (such as SuT2 and SuT4). To overcome this issue, one would need to implement SuT-specific add-ons. Nevertheless, due to the proxy-based approach of HitM, one would only need to implement such a solution once for each SuT.

Reproducibility HitM addresses the limitation of non-reproducible scans by running the WVSs in containers and taking snapshots of these containers. As a result, the containers can be reset to a certain snapshot and the scan can be run again from the same starting point. However, some WVSs generate random data for their scans on the fly, potentially leading to different test outcomes even if a run was started from the same snapshot. This challenge can only be addressed by the WVSs directly and thus is out of scope for HitM.

Pause and Resume a Scan Pausing a scan, particularly when targeting OT components, may be necessary to allow for time to restart the SuT. Leveraging containers with snapshots enables HitM to pause the WVSs and resume them once all necessary steps have been completed.

Conduction of Single Tests The containerization of WVSs also allows for the execution of subsets of security tests, enhancing the efficiency of WVSs lacking native support for running individual tests.

Load Reduction High network traffic loads can overwhelm certain SuTs. The containerization implemented by HitM enables stopping and resuming a scan as needed. For example, this helps in situations in which the SuT only allows for a limited number of Transmission Control Protocol (TCP) connections. If the proxy of HitM identifies a situation in which too many concurrent TCP requests are sent by a WVS, it can pause the WVS. This allows the SuT to

process the TCP requests, thereby freeing up resources. Then, the SuT once again has enough resources to respond to new requests from the WVS after the WVS is resumed by HitM.

Behavior in Case of an Error In the event of a SuT crash, WVSs may become stuck, hindering further vulnerability testing. HitM addresses this limitation by leveraging the combination of ISuTest® and the containerization of the WVSs. As stated in Section 2.2.3, ISuTest® conducts regular monitoring cycles to check whether one or more of the SuT's services have crashed. During these monitoring cycles, the WVS is paused. If ISuTest® detects that one or more services of the SuT have indeed crashed, it restarts the SuT by performing a power cycle. Subsequently, ISuTest® checks whether all services of the SuT are responsive again. If this is the case, the containerized WVS is resumed and can continue its tests. The crash of the SuT is noted by ISuTest® and reported as a finding.

In the specific example of SuT1, using HitM offers two key advantages: Firstly, a crash of SuT1 is automatically detected and SuT1 is restarted afterwards. With this, the scans of the WVSs can be fully completed. Secondly, HitM logs detailed information about the packages sent during the test and generates a report about the crash, using features of ISuTest®. This data enables a security tester to understand and further analyze the reported findings.

Different Scanners Necessary HitM supports the use of several WVSs by making use of the corresponding new feature of ISuTest® [*Pfr19b*]. With this, it is possible to schedule multiple scans using different WVSs. Each of them is transparently supported by the features of HitM.

4.3.4.3 Quantitative Evaluation

Following the qualitative evaluation with respect to the limitations of WVSs that are addressed by HitM, we present the results of our quantitative evaluation that analyzes the actual performance of the WVSs when used with OT components as targets. For this evaluation, we execute each of the considered WVSs against SuT1, SuT2, SuT3, and SuT4, using each of the configurations of

HitM presented in Section 4.3.4.1. We measure and analyze the true positive and false positive reports of the WVSs, the final URL coverage, the performance of the watchdog, and HitM's impact on the runtime of the scans of the WVSs. In summary, the evaluation reveals the following key insights.

1 The utilization of HitM and its add-ons leads to a higher number of true positive reports.

2 However, for most SuTs, the false positive rate of reports also increases with the utilization of HitM and its add-ons, which supports the findings by Esposito et al. [Esp18].

3 For most SuTs, the final URL coverage is increased by HitM and its add-ons.

4 The proxy itself is the primary factor impacting runtime, suggesting potential for runtime improvements through proxy enhancements.

Reports

The first part of the evaluation focuses on the analysis of the reports that the WVSs generate as a result of their scans. We evaluate the true positive reports and the false positive rate.

True Positive Reports Figure 4.6 displays the cumulative number of true positive reports generated by the WVSs for each SuT. It is important to note that the displayed number excludes false positives, but still includes duplicates. Due to our blackbox approach, we are unable to ascertain the exact number of distinct vulnerabilities that lead to the observed anomalies that then lead to the WVSs' reports. It might be that a single part of vulnerable code leads to observable issues in several web pages of the WA. Nevertheless, we manually analyzed and reproduced the reports of the WVSs to find the reports that are false positives.

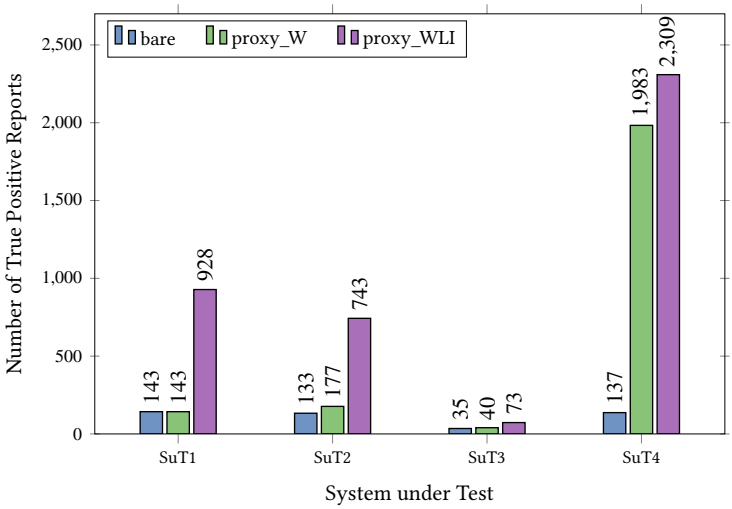


Figure 4.6: Number of the cumulative true positive reports generated by the six WVSs for the four SuTs. For each SuT, the number of true positive reports increases with the inclusion of more add-ons of HitM.

Figure 4.6 shows the results of the following configuration of HitM: (1) *bare*, where the WVSs are run without HitM, (2) *proxy_W*, where we use HitM including the logging and the watchdog add-on, and (3) *proxy_WLI*, where we use *proxy_W* and the login and the URL injection add-on (see also Table 4.4). For SuT2, SuT3, and SuT4, the number of true positive reports steadily increases with the inclusion of more of the add-ons of HitM. For example, the number of true positive reports for SuT3 are at 35 for the *bare* configuration, increase to 40 with *proxy_W*, and to 73 using *proxy_WLI*. For SuT1, the number of reports remains constant between the *bare* configuration and *proxy_W* (both at 143), but rises to 928 with the introduction of the user login and URL injection add-ons (*proxy_WLI*).

False Positive Reports In addition to the number of true positive reports, we also report the false positive rate of the reports generated by the WVSs. Figure 4.7 presents the cumulative false positive rate of the WVSs for each SuT. It shows that using HitM increases the false positive rate in comparison to the *bare* configuration for SuT1 and SuT4. For SuT3, using HitM in the *proxy_W*

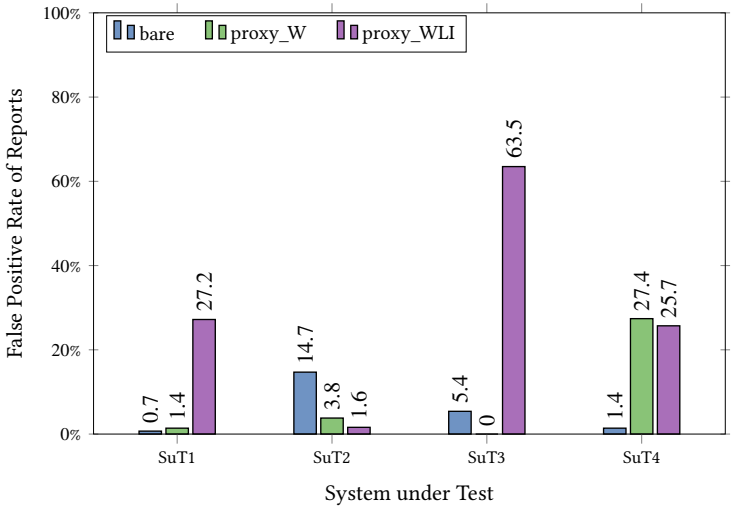


Figure 4.7: Cumulative false positive rate of the reports generated by the six WVSs for the four SuTs. Utilization of the user login and URL injection add-ons (*proxy_WLI*) increases the false positive rate for most of the SuTs.

configuration decreases the false positive rate from 5.4 to 0, but *proxy_WLI* increases the false positive rate again to 63.5. For SuT2, both configurations using `HitM` decrease the false positive rate in comparison to *bare*.

Discussion The analysis shows that the utilization of `HitM` increases the number of true positive reports given by the WVSs. Even though Figure 4.6 includes duplicates, we analyzed the reported vulnerabilities and showed that the increased number of true positive reports also includes reports that point to vulnerabilities that were not reported previously. That means that the usage of `HitM` actually leads to improved testing results.

Nevertheless, the false positive rate is also an important metric to evaluate the performance of a WVS. If a WVS reports many false positives, it is harder for a tester to identify the true positive reports which are worthwhile to analyze further. For most SuTs, the false positive rate increases with the usage of more add-ons, especially with the usage of the URL injection and user login

add-on. These observations align with the findings of Esposito et al., who similarly observed that integrating URL injection and user login increases the false positive rate [Esp18].

Furthermore, our analyses show that the proxy itself has an impact on the false positive rate. In the case that the SuT sends no response or an response that cannot be parsed correctly, the proxy sends a response to the WVS with the status code 502 (Bad Gateway). With this, the WVS usually assumes that it somehow triggered an server error. Even though this actually reveals an error in the SuT, the WVS reports all consecutive requests as an error and thus generates many false positives.

Notably, for SuT2, the false positive rate was decreased by the usage of HitM. This is mainly caused by the increased number of true positive findings, while the number of false positives stays roughly the same. SuT2 is not as affected by the increased false negative rate caused by the proxy as the other SuTs since it reliably sends responses to the requests. Thus, the watchdog and ISuTest® are never triggered during the tests and false positives based on proxy's responses as described above do not occur.

URL Coverage

An important metric to analyse the performance of a security test is the coverage that the test achieved [Li21]. In the case of blackbox WA scanning, we measure the URL coverage by counting the number of distinct URLs that were covered during a test. The intuition behind this metric is as follows. If a WVS fails to call a certain URL, it inherently cannot identify vulnerabilities present on this particular web page. Therefore, a broader scope of visited URLs generally corresponds to a higher likelihood of uncovering vulnerabilities [Dou12].

In the following, we assess the percentage of URLs on a SuT that were visited by at least one WVS, and compare the URLs present on a SuT with those visited by the WVSs. We define the set of URLs that are present on a SuT as the set of URLs in the URL index generated by the Logging add-on of the proxy (see Section 4.3.3.2). In the case of the *proxy_WLI* configuration, in which the

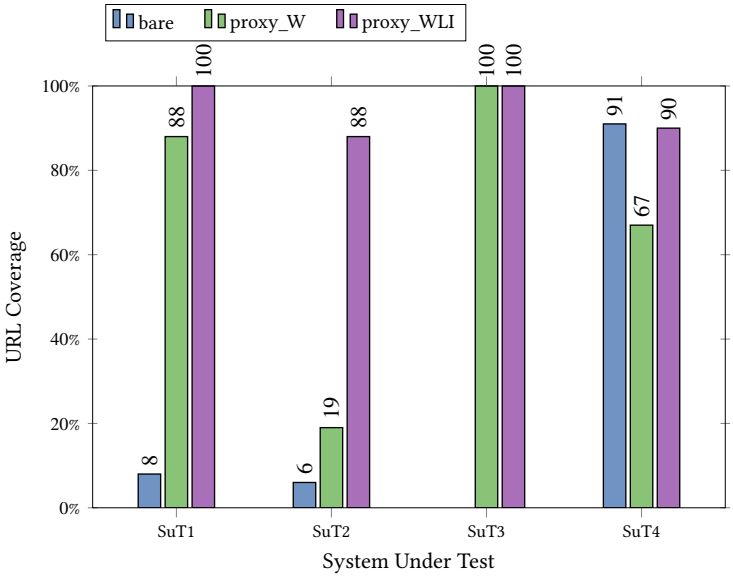


Figure 4.8: Cumulative URL coverage of the scans performed by the WVSs for each SuT. Generally, the URL injection add-on introduced with *proxy_WLI* leads to a higher URL coverage.

URL injection add-on is used and thus the URLs of the URL index are injected to the traffic, this results in an evaluation of how many of the injected URLs are actually visited by the WVSs.

Results Figure 4.8 shows the cumulative URL coverage of the WVSs for each SuT. For example, for SuT1, the URL coverage using the *bare* configuration is at 8%. Utilizing `HitM` with the watchdog add-on already increases the URL coverage to 88%, and the introduction of the URL injection add-on (*proxy_WLI*) increases it to 100%. Note that for SuT3 no value is displayed for the *bare* configuration. This is due to SuT3 using encrypted HTTPS traffic which we cannot analyze without the use of a proxy which acts as a Man-in-the-Middle and enables packet analysis for encrypted packets. Since the proxy of `HitM` supports HTTPS packet analysis, we can analyse the URL coverage of the *proxy_W* and *proxy_WLI* configuration.

For SuT1 and SuT2, the utilization of the URL injection add-on increases the final URL coverage. For SuT3, the proxy itself allows for a URL coverage of 100%, and the activation of the URL injection add-on does not change the final URL coverage. Interestingly, for SuT4, the application of the proxy decreases the URL coverage, but the utilization of the URL injection add-on increases the URL coverage again to a value close to the results of the *bare* configuration (91% and 90%, respectively).

Discussion The results for SuT1 and SuT2 show that the utilization of the proxy with only the watchdog add-on activated already leads to an increased URL coverage. There are several reasons for this, but for SuT1 the most apparent reason is the watchdog add-on which enables full scans of the SuT which were not possible in the *bare* configuration due to the SuT's crashes. For SuT4, the evaluation shows that the proxy with the watchdog add-on decreases the URL coverage, and the URL coverage achieved with the URL injection add-on does not exceed the URL coverage achieved with the *bare* configuration. These results highlight the challenge of automatically generating a URL index using non-deterministic WVSs. Especially due to Skipfish, the URL index generated for SuT4 contains some URLs with randomly generated parameter values that the WVSs visited during the generation of the index, but not during the evaluation. See also Section 4.6 for a discussion on the non-determinism of WVSs. Nevertheless, the results of this evaluation show that the utilization of the proxy and its add-ons increase the final URL coverage in most cases.

Runtime

In addition to `HitM`'s impact on the performance of the WVSs, we also evaluate its impact on their runtime. Since `HitM` utilizes a proxy which intercepts and processes each packet sent between the WVS and the SuT, it is expected to increase the overall time that is needed for a test. To quantify this hypothesis, we run Nikto with each configuration of `HitM` against SuT2. We choose Nikto for this evaluation since it generates its test deterministically and thus

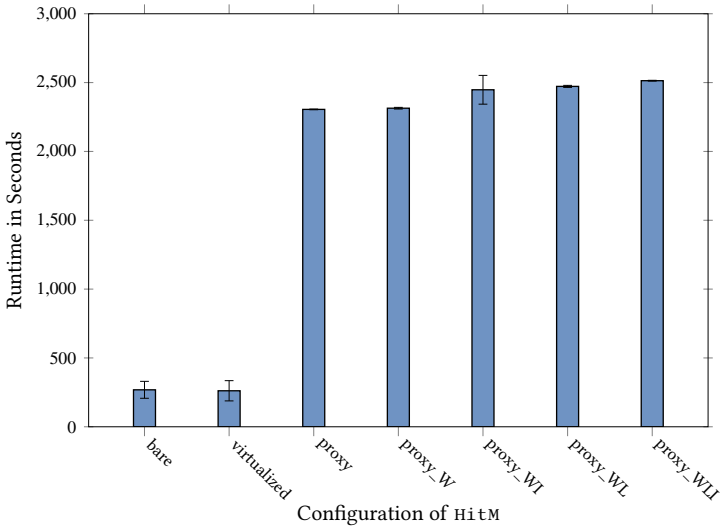


Figure 4.9: Mean runtime of ten runs of Nikto against SuT2 to measure the impact of the different configurations of `HitM`. The proxy itself introduces the highest runtime increase.

the runtime of Nikto itself is relatively stable. This stability allows for an assessment of the impact of `HitM` on the runtime. Moreover, we repeat each run of Nikto ten times to achieve reliable results.

Results Figure 4.9 displays the median value, along with the minimum and maximum runtimes of the ten runs of Nikto for each configuration. Firstly, it shows that the virtualization of Nikto has minimal impact on its runtime. Furthermore, the evaluation reveals that the additional add-ons for the proxy do not significantly affect runtime. However, as expected, the introduction of the proxy itself increases the runtime of the scan. For Nikto, it results in an increase from 261.5 seconds to 2,305.3 seconds (factor of 9).

Discussion For Nikto, the results reveal an increase in runtime, while the proxy itself has the highest impact on the runtime. This suggests potential for runtime improvement through proxy enhancements or a substitution of the proxy implementation.

However, for some combinations of WVSs and SuTs, HitM was able to even reduce the runtime. For instance, this is the case for the combination of Arachni and SuT4. Using the *bare* configuration, Arachni reaches the number of maximum allowed simultaneous TCP connections of SuT4 within roughly 31 seconds. Subsequently, Arachni fruitlessly attempts to establish new connections until the test's maximum runtime is reached and the test is interrupted. Leveraging HitM, the watchdog of HitM detects this issue, prompting ISuTest® to restart SuT4. Consequently, SuT4 can once again accept new TCP connections, enabling Arachni to complete its tests successfully. In this scenario, the runtime was reduced by a factor of 68 when comparing the *bare* configuration to *proxy_W*.

4.4 Testing BC_{ex}

The security of the WA provided by BC_{ex} is evaluated in the preliminary work on HitM [*Pfr19b*, *Bor18*], there denoted as *BK1* and *PHOENIX Buskoppler*, respectively. In these publications, the WVSs are combined with ISuTest® without including the containerization and the proxy proposed and implemented with HitM. Nevertheless, the combination of the WVSs and ISuTest® already improves the WVSs' performance. Furthermore, a manual analysis and verification of the findings of the WVSs is presented these publications, including an analysis of BC_{ex}.

For BC_{ex}, the WVSs report that several security-related HTTP headers are not set, i.e., the X-Frame Options Header, the X-Content-Type Options Header, and the Anti-XSS Header [*Bor18*]. Additionally, the integration of WVSs with ISuTest® uncovers three DoS vulnerabilities within BC_{ex}. When a WVS floods BC_{ex} with TCP packets with a high throughput, the WA becomes unresponsive under the high load (CVSS 5.3). Ceasing the high load restores the WA's availability. However, if the WVS opens around 400 TCP connections simultaneously without closing them, the WA of BC_{ex} crashes (CVSS 7.5). BC_{ex} needs to be restarted in order to make the WA available again.

For the third DoS vulnerability, first, one of the diagnostic pages of the WA needs to be accessed in a browser. These pages have the property of repeatedly sending new HTTP GET requests in the background to keep the diagnostic data up to date. When these requests are running, one needs to send a specific HTTP POST request to BC_{ex}. As a result, the WA initially reports that the Ethernet connection to BC_{ex} has been interrupted. Subsequently, this message disappears, and the WA can be used as usual. However, the digital I/O interface of BC_{ex} visually reports an error, using the physical LEDs, and does no longer send the signals that are expected. Once BC_{ex} is restarted, all interfaces work again as expected. This DoS vulnerability could only be revealed due to the consideration and monitoring of all communication interfaces of BC_{ex} (Challenge 4 (Insufficient Protocol Support)). All of these vulnerabilities have been reported to the manufacturer. It showed that the third DoS affected several devices by two vendors, and it was assigned a CVE identifier [CVE18].

Since the WA of BC_{ex} itself does not require authentication or includes any input fields, the advantages of the proxy of HitM do not come into play, and the utilization of HitM did not reveal any additional vulnerabilities of BC_{ex}.

4.5 Related Work

HitM addresses several limitations of the applicability of WVSs while also considering the challenges of industrial security testing. To this end, it implements a proxy-based approach which can be used with several WVSs. This section presents publications which are related to this objective and approach. None of these publications is concerned with the specific use case of industrial security testing. Note that some of the publications presented in this section have been published after the publication of HitM [*Bor20*]. To provide a full picture of the related work and state of the art, they are nevertheless included in this literature review.

Transparent Improvement of Web Vulnerability Scanners

One of the main objectives of HitM is to improve the performance of WVSs in a way that is transparent to the WVSs. Three approaches from literature also aim to transparently improve the performance of blackbox WVSs, two of which were published after the publication of HitM in 2020.

JARVIS Esposito et al. present JARVIS, a solution that uses a proxy to improve the WVSs' performance, similar to HitM. While JARVIS was originally published in 2018 [Esp18], the authors published an extended version in 2019 [Ren19].

The authors utilize the proxy to inject known URLs of the WA to the content of other pages of the WA to improve the WVSs' crawling capabilities. Furthermore, they use the proxy to authenticate to the WA, helping those WVSs which are not supporting the WA's authentication scheme. In their evaluation, Esposito et al. show that JARVIS increases the number of detected vulnerabilities while also increasing the number of false positives for some WVSs.

While the objective and basic approach of JARVIS and HitM are the same, the scope of HitM is different. On the one hand, HitM especially targets the domain of industrial security testing and considers the specific challenges of this domain. For example, this includes the need to monitor the responsiveness of the WA to detect crashes, and the need to stop and resume the WVSs to handle those crashes. On the other hand, HitM incorporates additional functionality into the proxy. For example, HitM allows to improve the determinism of the tested WA by mapping periodically changing content to fixed values. While working on HitM, we reached out to the authors of JARVIS in order to obtain the source code of JARVIS for conducting a thorough comparison. However, due to intellectual property concerns, the authors were unable to provide access to the source code.

In summary, JARVIS adopts a similar idea as HitM, but it diverges in focus and the features it encompasses from those proposed and implemented by HitM.

Scanner++ Yin et al. also propose a proxy-based solution, Scanner++, which was published after the publication of HitM [Yin23]. Scanner++ uses a proxy-based architecture with the objective to combine and thus enhance the capabilities of the used WVSs. To this end, Scanner++ first extracts the crawling and attack input generated by the WVSs when run in their basic configuration by observing the resulting network traffic. Then, the WVSs are run again while their detection is enhanced with the previously collected information by injecting the additional information to the network traffic.

For their evaluation, the authors integrate four WVSs, and use ten benchmark WAs and three real-world WAs as targets. This evaluation shows that Scanner++ improves the performance of the used WVSs in terms of the final coverage, the number of unique packets generated by the WVSs, and the number of bugs found. In particular, combining the capabilities of the different WVSs results in a greater number of discovered bugs compared to simply summing up the bugs found by the individual WVSs.

With this approach, Yin et al. pursue a similar goal as HitM, and improve upon one of the features proposed by HitM. HitM synchronizes the crawling inputs of the WVSs, while Scanner++, advances this synchronization by additionally combining attack inputs generated by the WVSs. As Scanner++ does not focus on the domain of industrial web security testing, other features of HitM, like monitoring of the WA or pausing and resuming a scan, are not implemented in Scanner++.

ReScan Drakonakis et al. also pursue the goal to transparently improve the performance of blackbox WVSs [Dra23b]. The authors also choose to build their approach, ReScan, based on a proxy. In contrast to the previously presented publications, ReScan incorporates a fully-fledged browser. As a result, dynamic content and client-side events can be evaluated and analyzed as well. Furthermore, ReScan constructs an explicit navigation model of the WA, and considers the different endpoints of a WA in their common context. For their evaluation, Drakonakis et al. use four different WVSs against ten WAs. They show that ReScan improves the performance of the WVSs in terms of code coverage and detected vulnerabilities. Despite the similar objective

of transparently improving blackbox WVSs, ReScan chooses a different focus than HitM. Thus, both approaches are complementary and could be used in combination.

Although Yin et al. [Yin23] and Drakonakis et al. [Dra23b] share the general objective with HitM [*Bor20*] and JARVIS [Esp18], neither of them cites the latter two publications.

Utilizing Additional Information in Blackbox Web Application Testing

The overall objective of this doctoral work and especially of HitM is to use the information that is accessible during a blackbox test to improve the general test performance. Trickel et al. follow a similar approach by providing graybox information to blackbox WA fuzzers [Tri23]. The difference to the approach of HitM is that the approach by Trickel et al., Witcher, collects actual graybox coverage information to be used by the fuzzers. Thus, they are enriching the information of the blackbox fuzzers by actual graybox information and Witcher is to be classified as a graybox approach, which is also stated by the authors. Nevertheless, the approach of Witcher does not need the source code of the WA that should be tested but is based on changes to the respective language's interpreter. Furthermore, Witcher's improvements are not transparent to the fuzzer.

Web Application Scanning for OT Components

Despite various sources indicating that the WA of an OT component frequently serves as an entry points for attackers (e.g. [Gon19, Dra23a]), research on the effective utilization of WVSs for OT components remains limited.

The Master's thesis preceding this doctoral work [*Bor18*] as well as the corresponding paper [*Pfr19b*] presents the integration of WVSs into the industrial security testing framework ISuTest® as well as an extensive evaluation on the WVSs' performance when targeting OT components. This evaluation shows that WVSs targeting WAs on OT components show a similar vulnerability detection rate as when targeting IT WAs, but with the drawback of a higher

false positive rate. In addition, several challenges that arise when running WVSs against OT components are identified, such as the necessity of pausing and resuming a scan of a WVS if the tested OT component crashes during a test. HitM builds upon these findings.

Considering a related domain, the security of Internet of Things (IoT), Jeannotte et al. present a security testing framework, which also includes two WVSs [Jea19]. However, this work does neither explicitly state the specific challenges of the IoT domain, nor do provides specific solutions to this.

4.6 Discussion

Acknowledging that HitM is to be seen withing the broader context of OT component security, the following discussion contextualizes the results and explains their implications in Section 4.6.1. Furthermore, this section discusses the limitations of HitM and the associated evaluation in Section 4.6.2, while also outlining possible directions for future work in Section 4.6.3.

4.6.1 Implications

Our analysis of known limitations of WVSs shows that WVSs especially struggle in applicability. This includes, for example, the behavior of the WVSs if the SuT is no longer available due to a crash or the load that is given on the SuT. These limitations are especially relevant for testing OT components, since these are resource-restricted and less resilient, and thus require more monitoring and fine grained tests. In summary, our analysis suggests that TTs need to acknowledge the special requirements of OT components to effectively reveal their vulnerabilities. This underscores the broader notion that the special requirements of OT components needs to be taken into account when applying general security measures and approaches. Several publications come to a similar conclusion, including publications focussing on OT

security in general [Idr19, Idr21] as well as publications focussing on specific parts of OT security such as honeypots [Mae22], Security by Design [Flu22], and security testbeds [Ani21].

HitM addresses several of the limitations that were identified during our analysis. With this, HitM supports the application of WVSs to OT components. Especially, HitM uses a transparent, proxy-based approach, which allows for arbitrary WVSs to be integrated into HitM. With this, HitM is not restricted to a specific set of WVSs, but can improve the performance of arbitrary WVSs, especially those which are to be built in the future,. We made sure to make the add-ons to `mitmproxy` developed during this work publicly available (see Section 4.2). With this, future research and security analyses can built upon the results of this work and improve upon the current state of HitM.

While primarily focussing on evaluating HitM, the evaluation conducted in this work also revealed several vulnerabilities in the tested SuTs. All of these vulnerabilities were responsibly disclosed to the respective manufacturers. One of the DoS vulnerabilities with a high severity present in BC_{ex} was assigned a CVE identifier [CVE18]. Although other vulnerabilities were confirmed by the manufacturers, they received no CVE identifiers. In most cases, this can be attributed to the internal vulnerability management processes of the manufacturers.

The primary focus of this doctoral work lies within the domain of testing OT components. However, the applicability of HitM extends beyond this specific domain. Notably, HitM could be used to improve testing of embedded systems in general. Such systems share similar requirements and show behavior similar to OT components during testing [Yun22]. For instance, embedded systems are also likely to crash as a whole if the WA encounters an error. This behavior can be detected and handled by HitM.

4.6.2 Limitations

While HitM addresses several of the identified limitations, it still requires some manual actions of the tester. For example, for uncommon or SuT-specific authentication approaches, the current implementation of the authentication add-on cannot derive the necessary steps for the authentication automatically. Thus, it is necessary that a tester manually performs the authentication while recording the steps using Selenium. This recording is then used by the Login add-on to perform the authentication automatically during testing.

Some of the used WVSs use non-deterministic randomness for their tests, which results in non-deterministic test processes and results. Thus, replaying the test from a given snapshot of one of these WVSs will not necessarily lead to the same results each time, restricting the fulfilment of the requirement of reproducible tests. However, this problem cannot be solved without changing the specific WVS and as such is out of scope for HitM.

Nevertheless, the non-determinism might also have an influence on the results of the quantitative evaluation. While this is not common practice in WVS evaluations, several runs of each configuration could have been run to receive more reliable results. This is already best practice for related research domains such as blackbox fuzzing [Kle18]. During the course of the evaluation of HitM, this was not possible due to time restrictions.

A prevalent challenge in blackbox evaluations is selecting a metric to analyze the performance of the evaluated TTs [Li21]. For the evaluation of HitM, we employed the following metrics: (1) number of false positive reports, including duplicates, (2) false negative rate, and (3) URL coverage. These metrics allow for a relative evaluation of the WVSs. For an absolute performance evaluation, establishing a ground truth for the number of vulnerabilities and the URLs of the WAs would have been necessary. However, the nature of blackbox evaluations using OT components prevented the creation of such a ground truth, since the internal structure and the vulnerabilities of the SuTs are not known.

4.6.3 Future Work

Future work comprises the improvement of automated URL index generation, which would help the WVSs to improve their crawling capabilities. With respect to the current approach of URL index generation, this could include a reduction and clustering of the currently collected URLs. With this, the number of URLs in the URL index could be reduced and thus the tests of the WVSs would be more focussed.

The current implementation of HitM mainly focuses on rather simple WAs, which do not make use of modern web approaches such as dynamically generated DOM content and asynchronous requests. One possibility to achieve this would be to combine HitM with the approach presented by Drakonakis et al. [Dra23b]. However, up until now, only few WAs of OT components include such modern features, but it is to be expected that they will be included in more WAs in the future.

In order to improve the runtime of HitM, one could investigate whether a different proxy implementation could be used as a basis for HitM. Furthermore, the evaluation and analysis of HitM could be extended by integrating more WVSs and by testing more OT components.

4.7 Summary

With HitM, we proposed, implemented, and evaluated an approach that transparently improves the applicability of WVSs in the domain of industrial security. During our evaluation, we successfully utilized HitM to test OT components, revealing several previously unknown vulnerabilities of the WAs of the OT components. We disclosed these vulnerabilities to the respective manufacturer and thus contributed to the overall security of OT components.

As a first step, we conducted a literature review to analyze which limitations of WVSs have already been identified by literature. This literature review identified 22 limitations of WVSs, which we then clustered into four clusters. Subsequently, we analyzed which of these limitations have been addresses by

literature. It shows that only few approaches address limitations with respect to the applicability of WVSs. These limitations are specially critical when testing OT components, since those are less resilient. Thus, WVSs need, for example, to be able to handle crashes of SuTs.

Based on these insights, we propose HitM, a proxy-based approach that transparently addresses the limitations of WVSs, while focussing on limitations with respect to applicability. HitM consists of a proxy, located in between the WVS and the SuT, the containerization of the used WVS, and an integration of the industrial security testing framework ISuTest®. Furthermore, HitM comprises several proxy add-ons, such as a watchdog add-on which concurrently monitors the SuT, and a Mapping add-on, dealing with periodically changing content.

Our evaluation, which was based on six WVSs and five OT components, shows that HitM addresses limitations with respect to a WVS's methodology (*Crawling*, *Periodically Changing Content*, *User Login*), and limitations with respect to a WVS's applicability (*Reproducibility*, *Conduction of Single Tests*, *Load Reduction*, *Pause and Resume a Scan*, *Behavior in Case of an Error*, *Different Scanners Necessary*). Moreover, it shows that the application of HitM increases the number of true positive reports and the final URL coverage. However, the false positive rate also increases, which supports results from literature [Esp18].

5 Blackbox Network Fuzzing

This chapter covers the subtopic of utilizing information from previously performed whitebox and graybox tests to improve future blackbox fuzzing tests of network stacks used in Operational Technology (OT) components. We base our analysis and approach on three vulnerability groups concerned with open source and commercial network stacks for Transmission Control Protocol (TCP)/Internet Protocol (IP), namely `Ripple20` [Koh20a, Koh20b], `Amnesia:33` [San21c], and `Urgent/11` [Ser19] (Section 5.1). Based on the knowledge from these vulnerability groups, we formulate Vulnerability Anti-Patterns (VAPs) (Section 5.4.2) and, subsequently, derive blackbox tests from these VAPs (Section 5.4.3). Our evaluation of these tests using eight OT components reveals several previously unknown vulnerabilities. Three of these vulnerabilities were assigned a Common Vulnerabilities and Exposures (CVE) identifier [CVE21a, CVE21b, CVE21c]. Moreover, our evaluation shows that implementations of the VAPs that we formulated spread over different implementations of the same protocol as well as over different protocols and different device classes (Section 5.4.4).

5.1 Problem Statement

OT components support several communication protocols, including industrial protocols as well as standard internet protocols, and thus incorporate several network stacks [Mar19]. These network stacks potentially introduce vulnerabilities to the OT component that could be exploited by attackers. To avoid exploitation, identifying and fixing these vulnerabilities is crucial. However, for a tester of an OT component, these network stacks are oftentimes blackboxes, since the network stacks are often purchased from third-party

vendors [Mar19, Dou23]. This is why blackbox tests and especially blackbox fuzzing tests of network stacks in OT components are necessary to improve the overall security of OT components. Nevertheless, there are use cases in which the source code of network stacks is available, either since the stack is open source or since the testing is performed in cooperation with the developers of the network stack. In this case, whitebox and graybox testing can be conducted. In the approach presented in this chapter, we aim to combine both cases and leverage insights from whitebox and graybox tests to improve future blackbox fuzzing tests of network stacks.

5.2 Contributions

This doctoral work makes the following two main contributions to the domain of industrial blackbox fuzzing.

Contribution 3. *Analysis and clustering of published TCP/IP stack vulnerabilities, and derivation of VAPs which formalize the underlying causes of these vulnerabilities.*

We analyze the vulnerabilities published with `Ripple20`, `Amnesia:33`, and `Urgent/11`, and cluster and thus structure the vulnerabilities as well as the root causes for these vulnerabilities. First, we classify the vulnerabilities based on the packet field they are concerned with, in order to understand the packet field types that are commonly vulnerable. Second, we cluster the vulnerabilities based on their root causes to understand which common root causes are responsible for the vulnerabilities. Third, we select the most common packet field types and formulate corresponding VAPs in order to make the knowledge of the root causes available in a structured format.

We show that more than 50% of the vulnerabilities of `Ripple20`, `Amnesia:33`, and `Urgent/11` are concerned with either (1) a length or offset field, or (2) the domain name. Additionally, we show that the vulnerabilities show similarities

within different implementations of the same protocol as well as within different protocols. A tabular representation of the derived six VAPs is available on GitHub [*Bor21*].

Contribution 4. *Utilization of the newly developed VAPs to implement new tests, and an evaluation of those tests which revealed three confirmed previously unknown vulnerabilities of OT components.*

To allow for an evaluation of the practical applicability of the structured knowledge derived from whitebox and graybox tests, we implement blackbox fuzzing test scripts which test for these VAPs. Our evaluation reveals eleven findings, including anomalies in the behavior of the Systems under Test (SuTs) and crashes. We disclose all findings to the manufacturer of the affected SuT, of which three findings were assigned a CVE identifier [CVE21a, CVE21b, CVE21c]. Moreover, our evaluation shows that similar vulnerabilities occur in different implementations of the same protocol as well as in different protocols, and also in different device classes.

We achieve this by integrating the fuzzing test scripts into the security testing framework ISuTest®, allowing for an orchestrated execution of the test scripts and monitoring of the SuTs. Based on this implementation, we run the test scripts against eight OT components from five device classes. The main objectives of this evaluation are as follows. With this, we first evaluate whether the test scripts based on the VAPs are able to identify vulnerabilities in the network stacks of the OT components. Second, we analyze the distribution of the findings, in order to understand whether similar vulnerabilities are found in (1) different implementations of the same protocol, (2) different protocols, and in (3) different device classes.

The approach presented in this chapter shows how information on network stack vulnerabilities accessible prior to a blackbox test can be leveraged to improve blackbox testing (Challenge 1 (Blackbox Testing), Challenge 2 (Missing Information)). Furthermore, the resulting test procedures were incorporated to ISuTest® (see Section 2.2.3) and thus all communication interfaces of the SuT can be observed during testing (Challenge 3 (Insufficient Observations)).

The content of this section is based on the corresponding publication produced during this doctoral work together with Philipp Takacs [*Bor22*]. The publication focuses on the technical details of the vulnerabilities and the implementation, while the content of this section focuses on the contextualization of the topic as well as a description of the VAPs.

Based on the crashes that have been identified with the approach presented in this chapter, we call it `ClusterCrash`.

5.3 Related Work

`ClusterCrash` is related to the domains of VAPs, vulnerability scanning, and industrial blackbox fuzzing. The following gives an overview of related work in these domains.

Anti-Patterns Anti-Patterns serve as descriptors for common errors encountered during software design or development [Tum19, Hec15], management practices [Jul13], code review [Cho21], and identifying vulnerabilities in design and development [Naf18]. Examples of such VAPs include the use of deprecated software or lack of proper authentication. Moreover, there are *performance* Anti-Patterns used in areas such as communication [Wer14, Tru18], cyber-physical systems [Smi20], and simulated models [Arr18]. Literature on Anti-Patterns published before the publication of `ClusterCrash` does not specifically address security or vulnerabilities in network protocols, industrial devices, or cyber-physical systems.

A research report by Forescout and JSOF, released concurrently with `ClusterCrash` and titled `NAME:WRECK`, focuses on DNS vulnerabilities and formulates six DNS Anti-Patterns [San21b]. The focus of `NAME:WRECK` differs from the focus of `ClusterCrash` since it analyzes DNS vulnerabilities and formulates DNS Anti-Patterns, while `ClusterCrash` chooses a more general approach and considers all the protocols analyzed by `Ripple20`, `Amnesia:33`, and `Urgent/11`. Since this also includes DNS, there are some similarities in the Anti-Patterns of `NAME:WRECK` and `ClusterCrash`. The Anti-Patterns AP#3 and AP#6 of `NAME:WRECK` correspond to VAP4 and VAP5 of `ClusterCrash`,

respectively, while AP#5 is partly covered by the test scripts generated based on VAP3. AP#1 of NAME:WRECK covers predictable random functions within DNS, which is a vulnerability that is usually not detected by fuzzing, since it has no direct impact on the behavior of the SuT, but rather allows for easier DNS spoofing attacks [San21b]. AP#2 is concerned with the character set of domain names, which is, as the authors state, not a direct vulnerability, but can make crafting new DNS packets easier for attackers. Again, this is not a vulnerability that is usually found with fuzzing. AP#4 describes the lack of validation regarding the NULL-termination of domain names. This is closely related to VAP4, but covers a more specific case.

In summary, NAME:WRECK follows a similar goal as `ClusterCrash`, but focuses on DNS, while `ClusterCrash` includes more protocols in its analysis. Note that both research works where conducted in parallel, and thus the results by NAME:WRECK could not be included in `ClusterCrash`. However, it would be interesting to generate new test scripts based on the Anti-Patterns presented by NAME:WRECK, and to analyze which new vulnerabilities could be found using these test scripts.

Vulnerability Scanning One approach to finding known vulnerabilities is vulnerability scanning. It involves scanning the SuT to find vulnerabilities by identifying the software versions running on the SuT (see also Section 2.3). Especially, scanners to identify the network stacks associated with the vulnerability groups considered in `ClusterCrash` have been developed. JSOF developed a blackbox scanner for `Ripple20`, available upon request, while Forescout and ArmisSecurity have published their blackbox scanners for `Amnesia:33`[1] and `Urgent/11`[2]. These scanners actively profile the SuT to determine if it includes any of the network stacks that are known to be vulnerable with respect to the respective vulnerability group. To this end, the scanners need to solve the following two main challenges. On the one hand, it is challenging to create and maintain a complete list of network stacks that are affected by a vulnerability group. This is based on the observation

[1] https://github.com/Forescout/project-memoriadetector
[2] https://github.com/armissecurity/urgent11-detector

that the affected components are used in various network stacks in different versions. On the other hand, fingerprinting the communication stack that is used on an OT component is especially hard [Cas13, Biß23]. Nevertheless, the scanner give a good indication on whether a OT component might be vulnerable. Thus, we use these scanners in `ClusterCrash` to ensure that none of the used SuTs is known to be vulnerable to the respective vulnerability group. For `NAME:WRECK`, a whitebox scanner was developed, which requires access to the source code of the SuT [Wan21d].

Industrial Blackbox Fuzzing Fuzzing is a testing technique that is increasingly used in the domain of industrial security. Lan et al. and Wei et al. both present an extensive overview of fuzzing in the industrial domain [Lan21, Wei24]. The following focuses on those fuzzers closely related to `ClusterCrash`.

Lan et al. show that most fuzzers focusing on the industrial domain base their test case generation either on heuristics or on grammars. These heuristics are usually built based on the general knowledge of the developers, but are not formally founded. One example for such a fuzzer is ISuTest®, presented by Pfrang et al. [Pfr18], which is also discussed in Section 2.2.3. To generate new test cases, ISuTest® employs static heuristics to select values for packet fields based on their data type. These heuristics are based on insights from previous projects conducted by the authors. Our approach, `ClusterCrash`, adapts a broader scope by identifying VAPs and using these to pinpoint promising field types and values for testing. Nevertheless, we implement the test scripts that we derive from these VAPs in ISuTest®, to make them available for the users of this framework.

Sasi et al. present `R0fuzz`, a blackbox fuzzer for the industrial communication protocols Modbus and DNP3 [Sas21]. `R0fuzz` incorporates three approaches for test case generation: random, generational, and mutational. The mutational approach utilizes the response code sent by the SuT to determine the interestingness of a test case. In contrast to `ClusterCrash`, this approach does not incorporate information sources available prior to testing.

Zhao et al. address a different problem in industrial network fuzzing by proposing an approach to automatically infer the structures of a proprietary industrial network protocol which applies techniques from Machine Learning (ML) [Zha19]. Lv et al. train ML models on network traffic data which are then used to generate test cases for the seen network protocols [Lv21]. Similar to `ClusterCrash`, these approaches leverage information sources accessible prior to testing, but they focus on different information sources and utilize the information to achieve different goals.

5.4 Cluster Crash

This section describes the work leveraging information from graybox and whitebox tests to be used in blackbox fuzzing tests of OT components. Section 5.4.1 details our analysis of published vulnerabilities, while Section 5.4.2 describes the derived VAPs. Information on the newly implemented blackbox fuzzing test scripts based on these VAPs is provided in Section 5.4.3. The evaluation of `ClusterCrash` is detailed in Section 5.4.4.

5.4.1 Analysis

The goal of this analysis is to understand the root causes and the similarities between the considered vulnerabilities, with the underlying objective of formulating VAPs on the basis of this analysis. We base our analysis on the vulnerabilities published with `Ripple20`, `Amnesia:33`, and `Urgent/11`. In total, these three vulnerability groups include 63 vulnerabilities and affect nine network stacks. `Ripple20` is concerned with the closed source Treck TCP/IP stack [Koh20a, Koh20b], `Urgent/11` affects the closed source Interpeak IPNet stack [Ser19], and `Amnesia:33` shows vulnerabilities of nine open source network stacks [San21c]. Of the 63 vulnerabilities, one vulnerability is not concerned with a specific protocol, but represents a general vulnerability of the Treck stack (CVE-2020-13987). This is why we exclude this vulnerability from

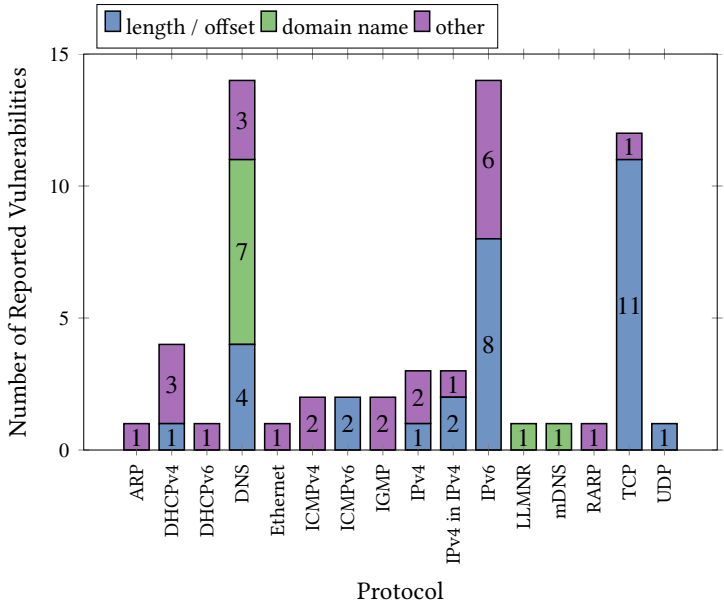


Figure 5.1: Classification of the considered vulnerabilities based on the affected protocol and packet field type. More than 50% of the vulnerabilities are related to a length or offset field, or the domain name.

our analysis. The vulnerability CVE-2020-13987, however, affects both TCP and UDP, which is why we divide it into two vulnerabilities for our analysis. In the end, we base our analysis on 63 vulnerabilities.

First, we classify the vulnerabilities based on the protocol that they affect as well as the packet field type they are based on. For our classification, we consider length and offset fields to be the same packet field type, since the use cases and error cases are similar. An aggregated representation of our classification is shown in Figure 5.1. It shows, for example, that the vulnerability groups included 14 vulnerabilities in DNS, of which 4 are based on a length or offset field, while 7 are based on the domain name. The remaining 3 vulnerabilities are concerned with other packet field types.

Our analysis reveals that out of the 63 vulnerabilities, 29 (46.03%) are based on a length or offset field, while 9 (14.29%) are based on the domain name field. After these two field types, the next most occurring packet field type, the ID field, is the base for only two vulnerabilities. Moreover, it shows that vulnerabilities concerning length or offset fields spread over almost all the considered protocols. Issues with the domain name, however, only occur in DNS, mDNS, and LLMR, even though the protocols DHCP, DHCPv6, and ICMPv6 also include domain name fields.

It is apparent that both length or offset fields, and domain name fields pose high challenges to the network stack that needs to parse these packets. One reason for this might be that most network stacks are implemented in C, which does not enforce boundary checks. This leads to less robust code with respect to length and offset calculations and accesses [Fet02]. The domain name complicates the challenge of correct offset and length calculations with a feature called *compression* [Moc87a, Moc87b]. This feature allows for a compressed representation of a domain name by conceptually using pointers that reference parts of the domain name that has been seen before. With this, it shortens the representation of the domain name, but adds additional complexity to the parser. For more details on the vulnerabilities and underlying mechanics of the protocols, please refer to the respective publication [*Bor22*] or the relevant Requests for Comments [Moc87a, Moc87b].

In summary, our analysis of the vulnerabilities reveals that the vulnerabilities show similarities across different implementations of the same protocol and also across different protocols. For instance, CVE-2020-11912, part of `Ripple20`, enables an out-of-bounds read using TCP. This out-of-bounds read is possible due to the absence of validation concerning the TCP `option length` field, thus failing to account for the actual length of the TCP options. Similarly, CVE-2020-17441 allows for an out-of-bounds read caused by missing validation regarding the IPv6 `payload length`, not considering the actual length of the IPv6 payload.

> Our analysis comes to the following conclusions:
>
> 1 More than 50% of the vulnerabilities are based on length or offset fields, or domain name fields.
>
> 2 Similar vulnerabilities affect different implementations of the same protocol as well as different protocols.

5.4.2 Vulnerability Anti-Patterns

Based on our analysis, we derive six VAPs for the most vulnerable field types, namely length or offset fields, and domain name fields. These VAPs represent the underlying common causes for those vulnerabilities from `Ripple20`, `Amnesia:33`, and `Urgent/11` that are concerned with these two field types. Refer to We present the VAPs in a tabular form which is based on the template introduced by Nafees et al. [Naf18]. In the following, one of the VAPs is presented in more detail to give insights on how the VAP was created and what the different parts of the VAP represent. The tabular representation of the VAPs was published upon publication of the corresponding paper [*Bor22*], and can be accessed on GitHub [*Bor21*].

In general, the tabular representation of the VAPs is divided into the three parts *General Information*, *Anti-Pattern*, and *Known Exploitation* [Naf18]. The first part includes information on the name of the VAP, an alternative name, a mapping to a Common Weakness Enumeration (CWE) ID, and examples of vulnerabilities that are related to this VAP. CWEs are commonly used to identify software and hardware weakness types[1]. The second part is concerned with the actual content of the VAP, thus detailing the development practices that are described by this VAP. This includes an example of an exploitable scenario as well as typical causes for this vulnerability. The third part gives information on the exploitability of the VAP by giving examples of attacks.

[1] https://cwe.mitre.org/

Table 5.1: Tabular representation of VAP1: Assume validity of length / offset field. Representation and content based on [*Bor21*].

Category	Description
General Information	
Anti-Pattern Name	Assume validity of length / offset field
Also Known As	Improper validation of length and offset field(s)
CWE Mapping	CWE-130: Improper Handling of Length Parameter Inconsistency
CVE Examples	CVE-2020-11912, CVE-2020-17441
Anti-Pattern	
Anti-Pattern Example	Consider the example code in Listing 1. In this case, the parser trusts the supplied length input without checking it against the real length of the given input. If an attacker sends a packet which contains a length field with the value of 255 and has option A set, but only contains 64 bytes of data, an out-of-bounds read will happen in line 8.
Typical Causes	Lack of input validation, and missing range checks
Known Exploitation	
Attack Patterns	CAPEC-540

These attacks are referenced to with their Common Attack Pattern Enumeration and Classification (CAPEC) ID. These CAPEC IDs are commonly used to uniquely identify and structure known attack patterns which have been used by attackers [1]. With the usage of commonly used identification numbers, such as CVE, CWE, and CAPEC, the VAPs developed in this doctoral work provide links to existing knowledge bases and thus are contextualized.

One of the VAPs, VAP1, is concerned with the two vulnerabilities that have been discussed briefly in Section 5.4.1: CVE-2020-11912 and CVE-2020-17441. Both are based on improper validation of a length field in the corresponding protocol. Thus, this VAP describes the Anti-Pattern of assuming the validity of length or offset fields. Table 5.1 shows the tabular form of VAP1. As

[1] https://capec.mitre.org/

Listing 5.1: Example code snippet to illustrate VAP1. Presentation and content based on [*Bor21*].

```
1  while (*pkg) {
2      len = *(pkg + 1);
3      switch (*pkg) {
4          case NOOP:
5              len = 1;
6              break;
7          case DATA:
8              memcpy(data, pkg + 2, len - 2);
9              break;
10         default:
11             break;
12     }
13     pkg += len;
14 }
```

can be seen in the first section of VAP1, we mapped this VAP to the corresponding CWE, CWE-130, which is concerned with the improper handling of length parameters [MIT24]. Out of the considered vulnerabilities, CVE-2020-11912 (`Ripple20`) and CVE-2020-17441 (`Amnesia:33`) are to be contributed to this VAP.

As suggested by Nafees et al. [Naf18], we include an example of a vulnerability that could be produced by this VAP to clarify the nature and practical impact of the VAP. This example is based on the C code snippet shown in Listing 5.1. This code snippet shows a simplified packet parsing and processing routine that continuously expects new network packets to be provided in the buffer `pkg`. First, in Line 2, the parser reads the length of the packet from the second byte of `pkg`. Then, it reads the type of the packet from the first byte of `pkg`, and parses the packet based on this type (Line 3). If the type indicates that the packet is of type `DATA`, the parser reads `len` bytes, starting from the third byte of the buffer `pkg`, and writes it into the `data` buffer (Line 8). Note that the parser does not check whether the length of the currently parsed packet

is equal to or greater than the number of bytes that is read in Line 8. Thus, it would read too much data if the length given in the second byte of `pkg` were greater than the actual data, resulting in an out-of-bounds read. While this is a simplified example, it illustrates the problem that VAP1 addresses to and the two related vulnerabilities are based on.

Moreover, the tabular representation of VAP1 includes information on known exploitations by referring to CAPEC-540 [MIT18]. This attack pattern is concerned with the general approach of out-of-bound buffer reads.

Similar to VAP1, VAP2 through VAP6 represent VAPs that are common in the considered vulnerabilities from the vulnerability groups `Ripple20`, `Amnesia:33`, and `Urgent/11`. For the details on VAP1 through VAP6, refer to the tabular representation of these VAPs on GitHub [*Bor21*].

5.4.3 Test Scripts

The main objective of `ClusterCrash` is to utilize information and knowledge from previous whitebox and graybox analyses for blackbox fuzzing tests of OT components. To this end, we use our analysis results and VAPs to implement 15 blackbox fuzzing test scripts. These test scripts are implemented and run in a true blackbox manner, meaning that we have no access to the SuTs' source code during development and the test scripts interact solely via the Ethernet interface with the SuTs. The primary goal of these test scripts is to induce anomalies or crashes in the tested OT components.

To ensure the usability of these test scripts, we design them to integrate with the security testing framework ISuTest® (see Section 2.2.3 and [Pfr18]). While standalone implementations require manual analysis of the SuT's behavior during the test, integrating the test scripts into ISuTest® allows for automated monitoring of the SuT.

Our test scripts target the communication protocols DHCPv4, DNS, IPv4, TCP, and UDP. We select these protocols based on the number of vulnerabilities within the considered vulnerability groups that concern the respective protocol, and their practical relevance for OT components. Initially, we identify the

Table 5.2: Packet fields of the different protocols that have been considered in the test scripts, including their types (length/offset (l/o) or domain name (dn)).

Protocol	Type	Target
IPv4	l/o	length
	l/o	internal header length (ihl)
	l/o	option length
TCP	l/o	data offset
	l/o	option length
	l/o	urgent pointer
UDP	l/o	length
DHCP	l/o	option length
	dn	search option
	l/o	option payload termination
	l/o	zero length option payload
DNS	dn	compression pointer
	dn	label length
	l/o	qdcount
	l/o	rdlength

protocols with the highest vulnerability counts, including DNS and IPv6 (14 vulnerabilities each), TCP (12 vulnerabilities), DHCPv4 (4 vulnerabilities), IPv4 and IPv4 in IPv4 (3 vulnerabilities each), and ICMPv4, ICMPv6 and IGMP (2 vulnerabilities each). Additionally, considering the widespread use of UDP and its vulnerability related to length fields, we also include it in our test scripts.

From this initial set of protocols, we narrow down our selection to those commonly found in OT components, aligning with the focus of our research. As a result, we select DHCPv4, DNS, IPv4, TCP, ICMP, and UDP. Each of these protocols, except ICMP, features testable length or offset fields and domain names, making them suitable for our test scripts. Hence, we exclude ICMP from our selection and our final set of protocols comprises DHCPv4, DNS, IPv4, TCP, and UDP.

Table 5.2 outlines the packet fields that test scripts target. These packet fields include the length or offset fields and domain name fields of the considered protocols, as our analysis shows that these packet field types are most vulnerable. For instance, we include three test scripts for the TCP protocol. The first one fuzzes the `data offset`, the second one fuzzes the `option length`, and the third one fuzzes the `urgent pointer`. Moreover, we include two additional test scripts for DHCP, which test for vulnerabilities in the packet field length parsing. The first additional test script, targeting `option payload` termination, adds a termination character at the middle of the option payload string. This enables us to assess the SuT's response to such unexpected terminations. The second additional test script, targeting zero length `option payload`, sends packets with an `option payload` with a length of zero. With this, we examine the SuT's capability to handle unexpected payload lengths.

We implement the fuzzing test scripts based on Scapy[1]. Details on the implementation of the test scripts as well as workarounds needed to set all necessary packet fields are provided in the publication on `ClusterCrash` [*Bor22*].

5.4.4 Evaluation

The objective of our evaluation is to analyze whether the test scripts resulting from our analysis and VAPs can be used to identify previously unknown vulnerabilities in OT components. Moreover, we aim to analyze whether similar vulnerabilities spread over different implementations, protocols, and device classes.

5.4.4.1 Evaluation Setting

We utilize our test scripts to assess eight OT components. Throughout this process, the OT components remain blackboxes, allowing us to realistically evaluate the efficiency of our methods and test scripts within a blackbox setting. Note that the OT components that we use as SuTs for our evaluation are not known

[1] https://scapy.net

to include any of the vulnerabilities published by `Ripple20`, `Amnesia:33`, or `Urgent/11`. This supports our objective of determining whether the VAPs and the derived test scripts can be used to find new vulnerabilities in other network stacks and OT components.

Hypotheses

Our evaluation of `ClusterCrash` is driven by the following hypotheses.

- H1 Blackbox test scripts derived from our VAPs help to identify previously unknown vulnerabilities.
- H2 Implementations of VAPs do not only spread over implementations of the same protocol but also over different protocols.
- H3 Implementations of VAPs spread over different device classes.

Systems under Test

For our evaluation, we use eight OT components from five different device classes. This ensures that our evaluation results have the potential to provide insights with respect to our hypotheses. Moreover, we perform a fingerprinting analysis revealing that each SuT includes a unique TCP/IP stack, further enhancing variability and coverage of the chosen SuTs. Below, we provide a brief overview of the functions and characteristics of the device classes considered in this evaluation. Consistent with the remainder of this doctoral work, we refrain from disclosing manufacturer details.

Firewall (FW) Acting as a barrier between network segments, a firewall like the SuT *FW* controls traffic flow. In addition, *FW* provides an HTTPS Web Application (WA) for configuration purposes.

Controller (Ctl) Industrial controllers monitor and control industrial processes. We select two controllers for our evaluation, namely *Ctl1* and *Ctl2*. *Ctl1* is a safety controller and thus meets higher redundancy and reliability standards.

Gateway (GW) *GW1* and *GW2* serve as OPC Unified Architecture (OPC UA) gateways, linking various communication protocols to OPC UA. With this, they facilitate combining and analyzing data from various sources.

I/O Device (IO) Industrial I/O devices serve to bridge analog and digital actuators and sensors with controllers. *IO1* allows for a coupling of PROFINET, IO-Link, and digital and analog signals, while *IO2* translates a controller's PROFINET communication to digital or analog signals, similar to BC_{ex}.

Sensor (Sen) The sensor used for our evaluation, *Sen*, functions as a temperature sensor. It provides its measurements through various communication protocols such as File Transfer Protocol (FTP), Simple Network Management Protocol (SNMP) and Message Queuing Telemetry Transport (MQTT). Additionally, *Sen* offers a WA displaying current measurements and allowing for configuration.

5.4.4.2 Results

This section presents the results of our evaluation, while Section 5.4.4.3 discusses the presented results. During our evaluation, we encountered crashes of the SuTs, as well as anomalies in their behavior. In the following, we use the term *finding* as an umbrella term for crashes and anomalies.

Findings

We identify a total of 11 findings. Figure 5.2 displays these findings by the affected protocol and the affected field type (length or offset, or domain name). It shows, for example, that our evaluation reveals 3 anomalies in DNS. Of these anomalies, 2 are concerned with a length or offset field, while 1 is concerned with the domain name. Notably, our test scripts reveal findings across all considered protocols except UDP, with the majority of findings affecting IPv4.

A more comprehensive breakdown of our results, as detailed in Table 5.3, reveals anomalies in IPv4 and TCP, as well as crashes caused by testing DHCPv4 and DNS. In Table 5.3, anomalies are denoted by **A**, while crashes are indicated

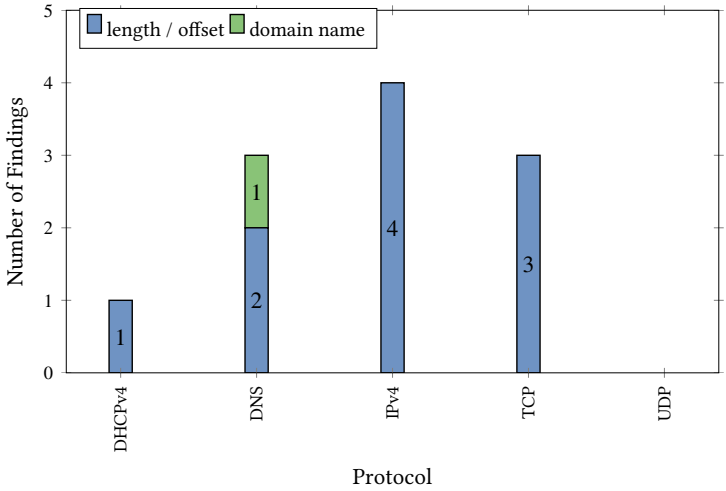


Figure 5.2: Findings of our test scripts, grouped by the affected protocol and field type. Except for UDP, the test scripts reveal findings in each protocol.

by **C**. Runs of our test scripts in which no findings occur are marked with dashes (-), while empty cells signify untested combinations. The untested combinations are due to lack of support of the respective protocol by the SuT. With respect to the previously mentioned findings in DNS, Table 5.3 shows that all of the three findings are crashes. One crash occurred in *Sen* while testing the `rdlength`, and the other two in *GW2*. Of those two, one also occurred while testing the `rdlength`, while the other occurred during the test of the `compression pointer`.

Crashes Two of the DNS-related crashes are triggered by an `rdlength` value exceeding the available data. For *Sen* as well as for *GW2*, an unexpected value for `rdlength` results in the SuT stopping to send DNS requests. This behavior indicates that the DNS resolver of the SuT crashed. In both cases, the expected `rdlength` values was 4. However, the concrete value needed to trigger the crash differs between the two SuTs. The DNS resolver of *Sen* crashes for a `rdlength` of `0x084a`, while the DNS resolver of *GW2* crashes for `0xfd8c`.

Table 5.3: Crashes (**C**) and anomalies (**A**) of the SuTs as revealed by the blackbox test scripts. Dashes (-) represent runs that do not reveal an anomaly or a crash, and empty cells represent combinations where the SuT does not support the corresponding protocol.

		Sen	*FW*	*Ctl1*	*Ctl2*	*IO1*	*IO2*	*GW1*	*GW2*
IPv4	len	-	-	-	-	-	-	-	-
	ihl	-	-	-	-	-	-	-	-
	optlen	**A**	**A**	-	**A**	-	-	-	**A**
TCP	len	-	-	-	-	-	-	-	-
	optlen	-	-	-	-	-	-	-	-
	urgent	-	-	-	**A**	-	-	**A**	**A**
UDP	len	-	-	-	-	-	-	-	-
DHCP	optlen	**C**	-		-			-	-
	search	-	-		-			-	-
	term	-	-		-			-	-
	zero	-	-		-			-	-
DNS	cptr		-		-				**C**
	labellen	-	-		-			-	
	qdcount	-	-		-				-
	rdlen	**C**	-		-				**C**

The third crash was caused by the test script targeting the `compression pointer` in DNS. *GW2* stops sending DNS requests after receiving two DNS responses with a `compression pointer` of `0x05` and `0x06`, respectively. However, the SuT is still up and running, and the web interface can still be used to trigger a new DNS request. Our further analysis suggests that the `NTPd` process, which is responsible to regularly generate and send DNS requests, has crashed. A reboot restores normal functionality.

The DHCPv4 test script targeting the `option length` results in a full crash of *Sen*. If the DHCP ACK packet lacks expected values, the DHCP client as well as all other services of *Sen* crash. Following a reboot, *Sen* resumes normal operation.

Anomalies Moreover, several anomalies are revealed by running the test scripts against the SuTs. The test script targeting the TCP `urgent pointer` reveals anomalies in three of the SuTs, namely *Ctl2*, *GW1*, and *GW2*. Based on

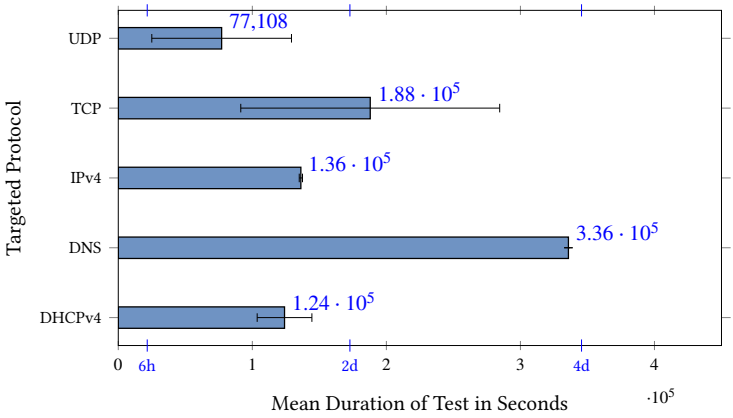


Figure 5.3: Run time of the test scripts, aggregated by protocol and averaged over the SuTs. The DNS test scripts take longer since the DNS test scripts need to wait for the SuT to establish a connection before being able to start the actual tests.

the usual use cases of the TCP `urgent pointer`, setting the `urgent pointer` to zero is unexpected, since the `urgent data` needs to include at least one byte and thus the pointer should be equal or greater than one [Ser19]. Thus, one would expect that an OT component treats a packet with the `urgent flag` set and the `urgent pointer` set to zero the same as a packet in which the `urgent flag` is not set and for which, thus, the `urgent pointer` is not parsed. However, our experiments show that the responses of the SuTs differ. For example, *GW1* always responds with an HTTP status code `200 Ok` to requests without the `urgent flag`, but sometimes responds with an HTTP status code of `400 Bad Request` to HTTP requests having the `urgent flag` set and a value of zero for the `urgent pointer`. With this, we show that the `urgent pointer` influences the HTTP response code, which is not to be expected.

Regarding IP, anomalies emerge in four industrial devices when the `option length` is set to an unexpected value. Even if the length being either too long or too short, the affected SuTs still reflect the content of the options back to the sender. This is true even if the options include non-parseable content. However, our manual analyses suggest that this anomaly cannot be exploited directly.

Duration

To provide insight into the duration required for the tests of each protocol, we report the timing of the test scripts in Figure 5.3. In this figure, each bar represents the duration of test script runs for each protocol, averaged over the runs against the eight SuTs. It shows, for example, that the tests of DNS took $3.36 \cdot 10^5$ seconds (almost four days).

Notably, the test scripts for DNS exhibit significantly longer run times compared to the other protocols. This is attributed to the difference in connection establishment that is necessary for DNS. Since we are testing the DNS client of the SuTs, the test script needs to act as the DNS server. Thus, the SuT needs to initiate the DNS connection. Depending on the SuT, this takes some additional time (see the corresponding publication for more information [*Bor22*]).

5.4.4.3 Discussion of Results

Our evaluation provides valuable insights into the hypotheses outlined in Section 5.4.4.1. In the following, we describe the implications of our findings presented in Section 5.4.4.2 concerning these hypotheses. To improve readability, the hypotheses are repeated above the corresponding discussion.

H1: Blackbox test scripts derived from our VAPs help to identify previously unknown vulnerabilities.

Our evaluation supports H1 since we developed the test scripts using a true blackbox approach, and our evaluation reveals several crashes of the OT components during testing. However, our test scripts did not lead to findings in the UDP network stack implementations. A possible explanation for this is that UDP is the least complex protocol we tested, and therefore less likely to be vulnerable. This is also reflected in the vulnerability groups on which we based our analysis, as only one UDP vulnerability was reported by the three vulnerability groups (see Section 5.4.1).

H2: Implementations of VAPs do not only spread over implementations of the same protocol but also over different protocols.

Santos et al. showed in their work that similar vulnerabilities occur in different implementations of the same stack [San21c]. Our evaluation supports this observation. For example, *Sen*, *FW*, *Ctl2*, and *GW2* all show the same anomalous behavior for the IPv4 protocol when testing the `option length`. Going one step further, our evaluation suggests that implementations of one VAP also spread over different protocols, thus supporting H2. For instance, our evaluation demonstrates this for VAP1 (see Section 5.4.2). On the one hand, one of the crashes in DNS is caused by an `rdlength` value exceeding the length of the available data. On the other hand, anomalies in IPv4 stem from an `option length` exceeding the length of the available data.

H3: Implementations of VAPs spread over different device classes.

Our evaluation supports this hypothesis since our results show implementations of VAPs over several device classes. For instance, this holds for the anomalous behavior triggered by the test script targeting the IPv4 `option length`. These anomalies can be observed for *Sen*, *FW*, *Ctl2*, and *GW2*. Each of these SuTs is from a different device class.

5.5 Testing BC_{ex}

In addition to the previously described evaluation, we also tested BC_{ex}, the bus coupler used as a running example for this doctoral work (see Section 1.6). As BC_{ex} does not support DHCP or DNS, we only run the test scripts regarding IPv4, TCP, and UDP. We find that our IPv4 test scripts lead to a crash of BC_{ex}. This crash happens when we send an ICMP echo request to BC_{ex}, where the `ipv4_len` field is set to zero. Interestingly, if we send a TCP packet, also with `ipv4_len` set to zero, BC_{ex} does not crash. Apparently, the crash is dependent on the payload of the IPv4 packet.

5.6 Discussion

This section puts the results of `ClusterCrash` into context and shows the implications of the findings revealed by our evaluation in Section 5.6.1. Moreover, this section discusses the limitations of `ClusterCrash` in Section 5.6.2, and outlines possible directions for future research in Section 5.6.3.

5.6.1 Implications

Following the previously described evaluation and the evaluation of BC_{ex}, we used the test scripts and approaches to test other OT components. Especially, we analyzed an industrial network switch regarding its behavior concerning the vulnerabilities published in the considered vulnerability groups by using our test scripts. Moreover, we included other known vulnerabilities, such as CVE-2014-4727[1], in our analysis. Based on our test scripts and additional manual analysis based on the aforementioned vulnerability, we found three previously unknown vulnerabilities of the switch. These vulnerabilities are based on TCP fragmentation [*CVE21a*], the `urgent pointer` [*CVE21c*], and Cross-site Scripting (XSS) via the LLDP protocol [*CVE21b*].

As with the other evaluations and findings of this work, we responsibly disclosed our findings to the respective manufacturers (see Section 2.2.2.2). With this, we ensure that the manufacturers are aware of the findings and can react accordingly. The findings that we revealed during the work on `ClusterCrash` affected the OT components of five manufacturers.

We contacted two of the manufacturers directly, and one of them acknowledged the vulnerabilities, published the vulnerabilities accordingly, and assigned them a CVE identifier [*CVE21a*, *CVE21b*, *CVE21c*]. The other manufacturer also acknowledged the vulnerabilities and fixed them, but was neither willing to publish the vulnerabilities, nor to assign a CVE identifier. The remaining three manufacturers were contacted via CERT@VDE, the Computer Emergency

[1] https://www.cve.org/CVERecord?id=CVE-2014-4727

Response Team (CERT) of the German Verband der Elektrotechnik, Elektronik und Informationstechnik (VDE). This CERT handles and coordinates vulnerability reports in the industrial domain, especially if they concern one or more of their members. We have discussed the anomalies with one of the developers and as a result of this discussion we have found that for some anomalies the source of the anomalous behavior is not the respective SuT, but the network stack of the Test Device (TD). Therefore, we do not consider these anomalies in our evaluation, but mention them in this discussion as it illustrates the challenges of blackbox testing.

In summary, `ClusterCrash` shows how knowledge from whitebox and graybox analyses can be generalized and be used to perform blackbox testing of OT components. This builds a solid base to implement new blackbox test scripts and to improve blackbox network fuzzing. Note that these test scripts are not limited to the domain of industrial security, but could be used to perform blackbox tests against arbitrary network stack implementations.

Moreover, we run the newly developed test scripts against OT components and revealed six vulnerabilities that have been acknowledged by the respective manufacturer. However, we encountered some difficulties during reporting these vulnerabilities caused by different maturity levels of the vulnerability handling processes by the manufacturers. As a result, only three of the acknowledged vulnerabilities were assigned a CVE identifier.

5.6.2 Limitations

One of the assumptions of `ClusterCrash` is the adoption of a realistic setting in which we perceive the SuTs as blackboxes. This perspective allows us to analyze and understand the requirements needed for blackbox tests of OT components. Nevertheless, this assumption also imposes certain limitations on our analysis and evaluation of vulnerabilities of the OT components.

Firstly, we are unable to ascertain the specific network stacks utilized by the SuTs. To address this limitation, we leverage fingerprinting techniques by employing tools like nmap[1] to identify the network stacks used by the SuTs. With this, we are able to demonstrate that all SuTs used for our evaluation use different network stacks.

Secondly, our ability to analyze the identified vulnerabilities is limited. Without access to the source code, it is challenging to definitely attribute each finding to its VAP and identify the underlying cause of an anomaly or crash. Nevertheless, the observable behavior already provides valuable insights into the relationship of findings and VAPs. To address this limitation, we report our findings to the corresponding manufacturers and discuss the implications of these findings with the developers (see Section 5.6.1).

5.6.3 Future Work

Future work comprises the extension of `ClusterCrash` into other domains and the inclusion of additional vulnerability groups. For instance, it would be interesting to explore the applicability of our VAPs and test scripts to Internet of Things (IoT) and Industrial Internet of Things (IIoT) devices. Since our VAPs and test scripts make no domain-specific assumptions about the SuTs, such an evaluation would be possible without the need to make significant changes to the VAPs and test scripts. In addition, the analysis that builds the base for the VAPs and test scripts could be extended by additional vulnerability groups such as INFRA:HALT [San21a] or NAME:WRECK [San21b] (see also Section 5.3). To further advance the automation of our test scripts, the state machine necessary for our DNS test scripts could be fully integrated into ISuTest®.

[1] https://nmap.org/

5.7 Summary

ClusterCrash structures general information on vulnerabilities in OT components by formulating VAPs. In addition, ClusterCrash provides test scripts that can be used to test OT components for implementations of the VAPs. Moreover, the evaluation of the test scripts of ClusterCrash reveals several vulnerabilities that now have been fixed by the manufacturers and thus improving the security of OT components and the production facilities they are used in.

To achieve this, ClusterCrash utilizes the knowledge of whitebox and graybox tests for blackbox fuzzing of OT components. More specifically, it analyzes and clusters the vulnerabilities published within the vulnerability groups Ripple20, Amnesia:33, and Urgent/11. It shows that more than 50% of the vulnerabilities are concerned with a length or offset field, or a domain name field. Based on this analysis, we derive six VAPs which summarize the underlying root causes for these vulnerabilities. These VAPs then are used to derive 15 blackbox fuzzing test scripts which test for implementations of these VAPs.

For our evaluation, we run these 15 test scripts against eight OT components and report our findings with respect to anomalies and crashes. In total, we report 11 findings affecting DHCPv4, DNS, IPv4, and TCP. We disclose the findings to the respective manufacturers. Six of the findings are acknowledged by the affected manufacturers, and three of them are assigned CVE identifiers. Moreover, our evaluation shows that implementations of VAPs spread over different implementations of the same protocol, different protocols, and different device classes.

6 Network Data Processing

This chapter covers the preprocessing of raw network traffic data for model-based blackbox fuzzing in the industrial domain. In this use case, the objective is to train Machine Learning (ML) models on network traffic generated during fuzzing, and to leverage these models to enhance the fuzzing process. To input the network traffic into ML models, the network traffic needs to be preprocessed first.

Our use case presents two specific requirements for this preprocessing. First, the preprocessing should be agnostic of the network protocols used, facilitating adaption to the diverse protocols used in the industrial domain, particularly proprietary protocols without accessible parsers. Second, although the preprocessing is intended for network traffic generated during fuzzing, it cannot be assumed that such data is always available for training purposes. Therefore, our goal is to develop preprocessing methods that can be trained on small datasets of user-generated network traffic and subsequently be applied to fuzzing network traffic.

The work presented in this chapter focuses on the preprocessing of network packets, with the downstream application being blackbox fuzzing using Hidden Markov Models (HMMs), as outlined in Section 7.5.

We analyze existing preprocessing approaches from literature in Section 6.4.1 and propose our novel preprocessing pipeline, including three options for dimensionality reduction, in Section 6.4.2. These options consist of a Principal Component Analysis (PCA) (denoted as PCA), an Autoencoder (AE) (denoted as AE), and the Convolutional Autoencoder (CAE) CAPC as proposed by Chiu

et al. [Chi20]. We evaluate the performance of this preprocessing pipeline with a special focus on the performance of the three dimensionality reduction approaches in Section 6.4.3.

Our experiments demonstrate that CAPC achieves the smallest reconstruction error when trained and validated on the same dataset. However, they also reveal that AE yields smaller reconstruction errors in a scenario where the dimensionality reduction approaches are trained on a small dataset with user-generated network traffic, and validated on a fuzzing dataset. This suggests that AE generalizes better. Moreover, our experiments confirm that the output dimension chosen for the dimensionality reduction affects the reconstruction error during training and validation, with the error decreasing as the output dimension increases. As such, there is a trade-off between having a low-dimensional encoding required for efficient data processing, and maintaining as much packet information as possible.

6.1 Problem Statement

One valuable data source that is available during blackbox tests, particularly during blackbox fuzzing, is the communication between the Test Device (TD) and the System under Test (SuT) as captured on the network. However, this network traffic must undergo preprocessing before it is usable by downstream ML models [Chi20]. Given the industrial focus of this doctoral work, it is crucial to recognize the unique demands of this domain. Specifically, the diverse, often proprietary communication protocols supported by the Operational Technology (OT) components pose significant challenges for leveraging network traffic efficiently (Challenge 4 (Insufficient Protocol Support), Challenge 3 (Insufficient Observations)).

Analyzing the specific contents of network packets, as achieved through deep packet inspection methods (see e.g. [Mag17, Ant12]), requires parsing each individual network protocol. In contrast, this doctoral work presents an approach that is protocol-agnostic, facilitating network traffic preprocessing without assuming parsers for each network protocol.

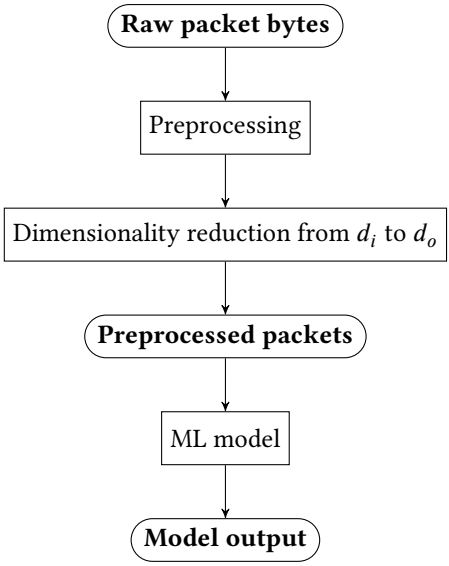


Figure 6.1: Overview of the general approach to preprocessing network packets for ML models.

Preprocessing only serves as the foundational step for subsequent data processing tasks, as presented in Figure 6.1. In the context of this doctoral work, the primary goal is to utilize preprocessed network data for model-based testing, with a particular focus on training multivariate HMMs (Section 7.5). The number of parameters of an HMM is directly linked to the dimensionality of its inputs, and minimizing the number of HMM parameters is essential for improving training efficiency [Cad22]. Therefore, one key objective of preprocessing the network data is to reduce the dimensionality of the data.

The approach presented in this chapter aims to preprocess network packets independently of specific communication protocols, ensuring suitability for ML models while prioritizing dimensionality reduction.

6.2 Contributions

This doctoral work makes the following contribution to the topic of network packet preprocessing. Based on the topic it contributes to, we call the newly presented approach NeDaP.

Contribution 5. *Analysis and evaluation of three approaches to network packet preprocessing with the goal of model-based security testing.*

We analyze three different approaches for network packet preprocessing and evaluate their performance. Specifically, we use a pipeline of preprocessing steps based on the work by Chiu et al. [Chi20] and Lotfollahi et al. [Lot20], and then reduce the dimensionality of the resulting data with three different means: (1) a PCA, (2) a basic AE, and (3) the CAE presented by Chiu et al. [Chi20], denoted as PCA, AE, and CAPC, respectively. We evaluate the performance of these approaches by measuring their reconstruction errors. In summary, our experiments show that CAPC outperforms PCA and AE if we train and evaluate the dimensionality reduction approaches on a single dataset, while AE outperforms the other two approaches if we train the approaches on user data but validate them on fuzzing data. The latter use case is the more realistic one, since we assume access to a small user data dataset, but the dimensionality reduction approaches are then used on fuzzing data.

NeDaP does not assume information on the used communication protocol and thus can be applied to blackbox testing as well as to industrial communication protocols (Challenge 1 (Blackbox Testing), Challenge 4 (Insufficient Protocol Support)).

6.3 Related Work

The work presented in this chapter, NeDaP, generally relates to the domain of traffic analysis and specifically network traffic classification. In this research domain, the objective is to classify the network traffic into certain categories, such as normal or abnormal traffic, or the application type [Pac18]. This

classification can, e.g., serve the tasks of intrusion detection [Zen19], cyber attack classification [Spe24] and quality of service management [Hir09]. For this classification, the network traffic needs to be preprocessed first, linking it to the work presented in this chapter.

In the past, traffic classification relied on port-based methods, identifying applications by their registered Transmission Control Protocol (TCP) ports [Pac18]. However, this approach becomes unreliable if applications use random or unregistered ports. Going one step further, *deep packet inspection* analyzes the payload of network packets [Mag17]. However, this approach is restricted to use cases in which a parser for the considered protocol is available and the payload is not encrypted. To address these challenges, ML has emerged as a viable solution, enabling not only traffic classification but also prediction and new knowledge discovery [Pac18]. The ML-based approaches can be divided into (1) flow-based approaches, and (2) packet-based approaches [Chi20]. Flow-based approaches rely their analyses on a packet flow, i.e. a set of related packets that is sent from a source to a destination [Pac18]. In contrast, packet-based approaches focus on the classification of single packets.

As the use case for the network packet preprocessing discussed in this chapter is packet-based model training for blackbox fuzzing, we focus on *packet-based* traffic classification in the following. Moreover, accounting for the protocol-agnostic approach of NeDaP, we focus on classification of *encrypted* network packets. Similar to the analysis of non-parseable proprietary industrial network protocols, the analysis of encrypted traffic also needs to rely on the general features of a network packet since it cannot decrypt and parse its actual payload.

Lotfollahi et al. and Chiu et al. present approaches for the analysis of encrypted traffic that include preprocessing of single network packets to be used for ML models [Lot20, Chi20], while Lim et al. choose a different preprocessing approach and transform application level data of network packets to images which are then fed into the ML models [Lim19]. Both of the first two approaches perform several steps for the preprocessing of network packets, including padding, normalization, and dimensionality reduction. As we also require the preprocessing of single network packets, we base our approach on both of

these works, and present a deeper analysis in Section 6.4.1. Both approaches use the preprocessing to prepare the network packets to be fed into a Neural Network (NN). This NN is then used to classify the packet, for example by the application that created that specific packet. However, since the focus of both publications is on the network traffic classification, their evaluation only includes information on the performance of the resulting NN. In contrast, our experiments presented in this chapter focus on the performance of the network data preprocessing itself (Section 6.4.3). We apply the preprocessing steps and evaluate the performance of the full application in Section 7.5.

As discussed in Chapter 3, other approaches in blackbox testing utilize network traffic as well, and thus need to process it as well. Zhao et al., Lin et al. and Lv et al. preprocess a network packet stream before feeding it into an Long Short-Term Memory Network (LSTM). Lin et al. and Lv et al. both parse the network traffic and extract the TCP payload from each packet [Lin21, Lv21], while Zhao et al. use the full packets [Zha19]. The bytes are then converted into a decimal representation and then directly fed into the LSTM. As an LSTM does not require fixed-sized inputs, the data does not need to be padded. Moreover, the authors do not require a dimensionality reduction approach as the parameters of the LSTM to not depend on the length of the sequences.

PULSAR represents the network protocols by the occurrence of certain tokens and n-grams within the network packet. For dimensionality reduction, PULSAR excludes volatile features and constant elements [Gas15].

6.4 Preprocessing of Network Packets

The objective of the work presented in this chapter, NeDaP, is to analyze the performance of various network packet preprocessing approaches, with a special focus on dimensionality reduction. This builds the base for model-based fuzzing with HMMs, as presented in Section 7.5. Initially, we review related preprocessing approaches (Section 6.4.1.1), as well as five datasets from the domain of industrial security testing (Section 6.4.1.2). We adapt the existing approaches to this domain and present a preprocessing pipeline including

three different dimensionality reduction approaches, a PCA, an AE, and a CAE, in Section 6.4.2. Our evaluation focuses on the performance of these three dimensionality reduction approaches with respect to in-domain and out-of-domain generalization. Moreover, we analyze the impact of the input dimension d_i and the output dimension d_o of the dimensionality reduction approaches.

6.4.1 Analysis of Existing Approaches

As a basis for our approach and experiments, we analyze related approaches from literature as well as datasets representing various use cases from the industrial domain. The two relevant approaches from literature, namely CAPC by Chiu et al. [Chi20] and Deep Packet by Lotfollahi et al. [Lot20], both use similar strategies for the preprocessing of network packets. However, they use protocol-specific knowledge for some of the preprocessing steps. In contrast, we aim for an approach that is independent of the concrete communication protocol, to be as flexible as possible (Challenge 4 (Insufficient Protocol Support)). Moreover, CAPC and Deep Packet focus their evaluations on the performance of the ML model using the preprocessed packets, while we focus on the performance evaluation of the actual preprocessing, with a special focus on the dimensionality reduction.

6.4.1.1 Preprocessing Steps

As shown in Section 6.3, network packet preprocessing is related to approaches concerning the analysis of encrypted network traffic using ML models. Thus, we analyze the approaches in this domain and build our work upon them. More specifically, we analyze the work published by Lotfollahi et al. [Lot20] and by Chiu et al. [Chi20], called Deep Packet and CAPC respectively. Table 6.1 presents an aggregated view on the preprocessing steps from the two publications as well as the preprocessing steps for NeDaP. This section gives details on the approaches from literature, while Section 6.4.2 focuses on presenting the new approach presented in this doctoral work.

Table 6.1: Steps for network packet preprocessing as presented by literature [Lot20, Chi20], and as proposed by this doctoral work. Both approaches from literature assume IP-based traffic and trim or mask the corresponding headers or address fields. Furthermore, both approaches fix the length of the input data to 1500. Since this work focuses on the performance analysis of the preprocessing and dimensionality reduction, we include three different approaches for the dimensionality reduction.

	Deep Packet [Lot20]	**CAPC** [Chi20]	**NeDaP**
Trim	Remove Ethernet header	-	-
Mask	Mask IP address	Mask IP address and MAC address	-
Fix length	Pad with zeros or cut to a length of 1500	Pad with zeros or cut to a length of 1500	Pad with zeros or cut to a length of $d_i \in \{304, 1504\}$
Normalize	Normalize bytes to [0,1]	Normalize bytes to [0,1]	Normalize bytes to [0,1]
Reduce dimensions	(1) Stacked AE, (2) 1D CNN	CAE	(1) PCA, (2) AE, (3) CAE

All approaches take raw packet data as input. CAPC as well as NeDaP explicitly use the packet bytes as input, while Deep Packet starts with the packet bits and transforms them to bytes later on. However, this does not introduce a conceptional difference between the approaches, since both approaches base their actual calculations on the byte representation of the packets.

Deep Packet and CAPC both trim and mask the network packet first, while assuming knowledge on the communication protocols used in this specific packet. More specifically, both approaches assume IP-based traffic. Deep Packet removes the Ethernet header of the packet, claiming that this header does not include information necessary for the two tasks of Deep Packet, namely application identification and traffic characterization. Moreover, Deep Packet masks the IP address of the packet with a fixed value to prevent that the trained model will draw conclusions from these addresses which will not be transferable to other datasets. Driven by the same goal, CAPC masks the MAC

address and the IP address in the network packet. Note that Deep Packet does not mask the MAC address since this address is located in the Ethernet header which is removed from the packet either way.

Since the ML models that the packets are preprocessed for expect a fixed input size, both approaches conduct steps to produce data that is of fixed length. This is achieved by zero padding those packets that are too short, and cutting those packets that are too long. Deep Packet and CAPC both fix the length of the packets to 1500 bytes. Note that the resulting data includes more IP payload data for Deep Packet, since the Ethernet header is removed from the packet before cutting. To ensure that each of the remaining bytes of the packet has the same weight, the bytes are normalized to the interval [0,1] by both approaches.

Subsequently, both approaches perform a dimensionality reduction to generate a more dense representation of the input data. Deep Packet uses two approaches for this, namely a Stacked Autoencoder (SAE) and a 1D CNN. A SAE is an architecture consisting of several AEs. These AEs are stacked in a way such that the output of one AE is used as the input of the next AE [Lot20]. With this, more complex relationships within the features can be represented by the AEs and a more efficient dimensionality reduction can be achieved [Vin10]. CAPC uses a CAE consisting of several convolutional and pooling layers. With this, the authors aim to efficiently extract the relevant information from the network packet [Chi20]. Since both approaches use convolutional parts in their dimensionality reduction step, they focus on local correlations instead of global correlations over the whole network packet [Pin21].

6.4.1.2 Network Data

As we locate our work in the domain of industrial security testing, and aim for a preprocessing pipeline to be used for model-based fuzzing, we choose corresponding datasets for our analysis and experiments. In the following, we first describe the datasets and then analyze those properties that are relevant for the preprocessing pipeline. Table 6.2 summarizes this information. Note that the datasets contain a different number of network packets, with the

maximum being 15,350. We aim to make our analysis as realistic as possible, and in usual industrial security testing scenarios, only small sets of network traffic data are available.

Industrial System This dataset is based on a testbed at Fraunhofer IOSB, representing a real production environment. It includes bus couplers, sensors and actors. We excluded the PROFINET realtime data from the original dataset, since this would introduce a high class imbalance due to the high throughput and thus high representation of realtime data in the dataset.

Vulnerability Scan In this dataset, we find a vulnerability scan conducted using the web vulnerability scanner ZAP[1]. More specifically, it shows a scan of the Web Application (WA) of an industrial controller. Thus, the dataset includes various HTTP requests and responses, as the scanner tries to analyze the WA.

Web Fuzzing This dataset also includes traffic generated by ZAP, but that of a fuzzing run. Again, this dataset includes mostly HTTP requests and responses, but these requests and responses are rather untypical since ZAP mutates these packets in various ways to reveal vulnerabilities in the scanned controller. Thus, one would expect that this dataset poses a greater challenge to the dimensionality reduction approaches than the previous dataset.

User Data This dataset shows traffic of a user interacting with a File Transfer Protocol (FTP) server, namely ProFTP. It is part of the datasets recorded for the subsequent model-based testing with HMMs. More details on the data collection can be found in Section 7.5.2.

AFLnwe Fuzzing Similar to *Web Fuzzing*, this dataset includes fuzzing traffic. In contrast, this dataset is based on FTP fuzzing conducted by the graybox network protocol fuzzer AFLnwe[2]. It targets the same FTP server that is used in the *User Data* dataset, ProFTP.

[1] https://www.zaproxy.org/
[2] https://github.com/thuanpv/aflnwe

Table 6.2: Properties of the considered datasets, including the number of total packets, the median packet lengths (PL), and the percentage of packet lengths that are smaller than the threshold of 304. Moreover, we show the number of Principal Components (PCs) that are needed to achieve ≥ 0.99 cumulative explained variance, giving a measure for the complexity of the network packets in the respective dataset.

Name	Total Packets	Median PL	PLs ≤ 304	PCs
Industrial System	15,100	60	99.7%	91
Vulnerability Scan	15,350	66	72.1%	108
Web Fuzzing	9,524	66	90.0%	30
User Data	1,127	72	98.7%	36
AFLnwe Fuzzing	15,100	74	99.7%	49
BC_{ex}	944	60	100.0%	104

BC_{ex} This dataset includes the traffic of a user interacting with the FTP server of the running example bus coupler BC_{ex}. The user performed the same actions as in the *User Data* dataset, but BC_{ex} does not support all of the used FTP commands. For example, it does not support creating new directories or renaming files.

Packet Lengths

The first property of the datasets that we analyze is the total length of the packets included in the datasets, including all headers. Similar to Lotfollahi et al. [Lot20], we use this information to choose possible values for the fixed packet length d_i for our experiments. Lotfollahi et al. show that 96% of packets in their dataset have a total length of less than 1500 bytes and thus decide to set the fixed length to 1500 during preprocessing.

As an example, Figure 6.2 shows an FTP packet taken from *User Data*, including the raw bytes on the left and the corresponding text on the right. This packet forms a request to get the contents of the folder `Testfolder1`. It includes 14 bytes of Ethernet header (green), 20 bytes of IP header (red), and 32 bytes of TCP header (blue). The FTP payload consists of the final 18 bytes requesting the contents of the test folder (orange). In total, this network packet has a length of 84.

```
02 42 ac 13 00 02 02 42  ab e7 0e 69 08 00 45 00   .B.....B ...i..E.
00 46 22 eb 40 00 40 06  bf 9d ac 13 00 01 ac 13   .F".@.@. ........
00 02 95 84 00 15 0d e2  1f f9 39 07 22 53 80 18   ........ ..9."S..
01 f6 58 62 00 00 01 01  08 0a f0 04 5f 61 fb 5e   ..Xb.... ...._a.^
31 e6 43 57 44 20 2f 54  65 73 74 46 6f 6c 64 65   1.CWD /T estFolde
72 31 0d 0a                                        r1..
```

Figure 6.2: Bytes representation of a FTP packet with a length of 84, showing the bytes on the left and the corresponding text representation on the right. Non-printable characters are represented as dots. The shown packet consists of an Ethernet header (green), an Internet Protocol (IP) header (red), a TCP header (blue), and a FTP payload (orange) requesting to change the working directory (CWD) to the folder *TestFolder1*.

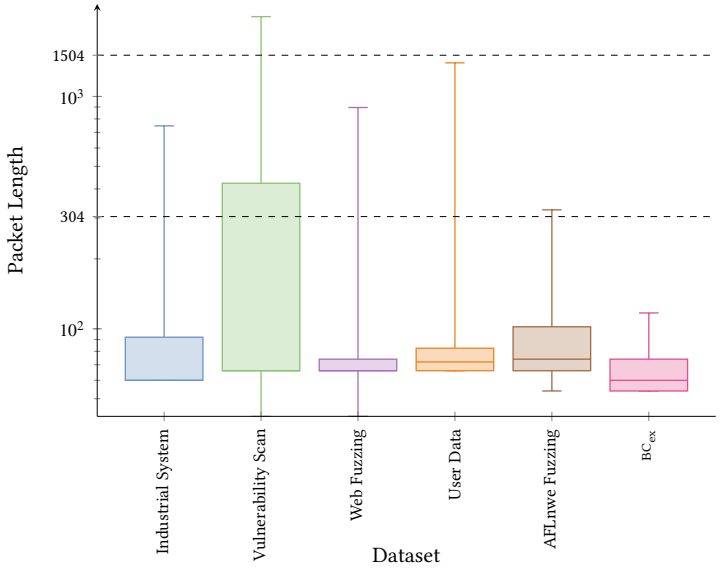


Figure 6.3: Box plot of the total packet lengths observed in the datasets, with the boxes showing the upper and lower quartile and the whiskers showing the minimum and maximum value. The two horizontal lines show the fixed packet lengths used for the experiments at 304 and 1504.

Figure 6.3 shows a box plot representing the packet lengths that we observe in our datasets. The boxes represent the upper and lower quartile, while the whiskers show the minimum and maximum values. For example, the

Vulnerability Scan dataset shows a maximum packet length of 2205 and a minimum packet length of 42. Note that the values on the y-axis are shown on a logarithmic scale.

As we aim to propose a preprocessing pipeline suitable for network fuzzing, we focus on the *User Data*, BC_{ex}, *Web Fuzzing*, and *AFLnwe Fuzzing* datasets to choose possible values for the input dimension d_i. Based on the packet length data, we first choose $d_i = 304$ for our experiments. With this number, 90.0% of the packets have fewer bytes than this threshold for *Web Fuzzing*, and 99.74% for *AFLnwe Fuzzing* (see also Table 6.2). For the dataset recorded with BC_{ex}, 100% of the network packets have a length smaller than 304. Within *User Data*, 15 packets are longer than 304 bytes. Each of these packets is an FTP data connection packet. The FTP data connection is an additional communication channel opened for data transfer within FTP [For10]. Within this additional channel, possibly large files are transferred and thus the FTP data connection packets tend to be larger in size. However, since most fuzzers only consider the control connection and not the data connection, these packets are unlikely to appear in usual fuzzing scenarios. With this, we should be able to cut the packets at a length of 304 bytes without changing the actual content of the packets much.

Moreover, 304 is divisible by 8, which helps with the implementation of the CAE used by CAPC that we will be using for our experiments, since it requires to divide the number of input dimensions into halves three times. Thus, if we choose a number divisible by 8, less rounding effects are to be expected. We choose $d_i = 1504$ as the second option for our experiments. On the one hand, we based this choice on the results from literature. On the other hand, Figure 6.3 shows that all packets in *Web Fuzzing* and *AFLnwe Fuzzing* are below this threshold and thus it serves as a good baseline for the other choice, $d_i = 304$.

Explained Variance

The second property of the datasets that we analyze is the number of PCs that are needed to explain the datasets. For this, we conduct multiple PCAs, each with a different number of PCs, and calculate the *cumulative explained variance*.

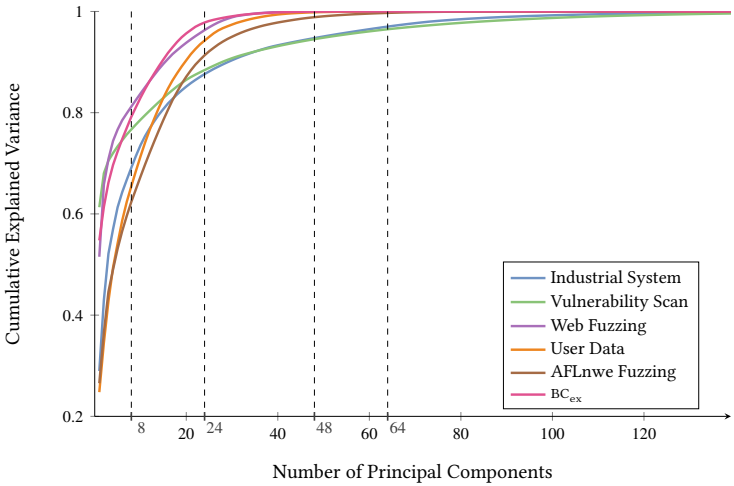


Figure 6.4: Cumulative explained variance for the considered datasets. *Industrial System* and *Vulnerability Scan* are the most complex datasets, while the two most relevant datasets, *User Data* and *AFLnwe Fuzzing*, need fewer PCs to be explained. We choose the output dimensions 8, 24, 48, and 64 for our evaluation and represent them as dashed lines in the figure.

The explained variance denotes how much of the total variance of the used dataset is explained by each PC [Gér22]. Therefore, the cumulative explained variance indicates how much of the total variance of the dataset is explained by all PCs of a PCA together. We leverage this metric to identify choices for the output dimension that is to be achieved by our dimensionality reduction. Our goal is to identify a dimension that sufficiently captures information while remaining manageable for the subsequent ML model utilizing the preprocessed data. Note that a PCA uses only linear mappings, while AEs can learn non-linear mappings. Nevertheless, the explained variance helps us to select a number of possible choices for the dimensions to be further evaluated during our experiments.

Figure 6.4 displays the results of this analysis, showing the number of PCs on the x-axis and the cumulative explained variance on the y-axis. It shows that the curves for *Industrial System* and *Vulnerability Scan* converge slower than the curves representing the other datasets, indicating that these two datasets

are more complex. For example, 91 PCs are needed to achieve a cumulative explained variance ≥ 0.99 for *Vulnerability Scan*, while only 30 PCs are needed for *Web Fuzzing* (see also Table 6.2). With respect to our use case of model-based fuzzing, *User Data* and *AFLnwe Fuzzing* are the most relevant datasets. For *User Data*, 36 PCAs are needed for a cumulative explained variance of ≥ 0.99, while 49 PCs are needed for *AFLnwe Fuzzing*.

Based on these observations, we choose 8, 24, 48, and 64 as output dimensions for the dimensionality reduction step in our experiments. We will evaluate the impact of the chosen output dimension in Section 6.4.3.

6.4.1.3 Analysis Conclusions

Our analysis leads to the following main conclusions:

1. The two related approaches from literature, CAPC and `Deep Packet` both use similar preprocessing steps, wile using protocol-specific knowledge for some of the steps.
2. In the two most relevant datasets, *User Data* and *AFLnwe Fuzzing*, more than 98.7% of the packets are shorter than 304. Thus, we choose 304 and the value from literature, 1504, as possible choices for the input dimension d_i for our experiments.
3. For *User Data* and *AFLnwe Fuzzing*, up to 49 PCs are needed to achieve a cumulative explained variance of ≥ 0.99. We choose 8, 24, 48, and 64 as possible choices for the output dimension d_o for our experiments.

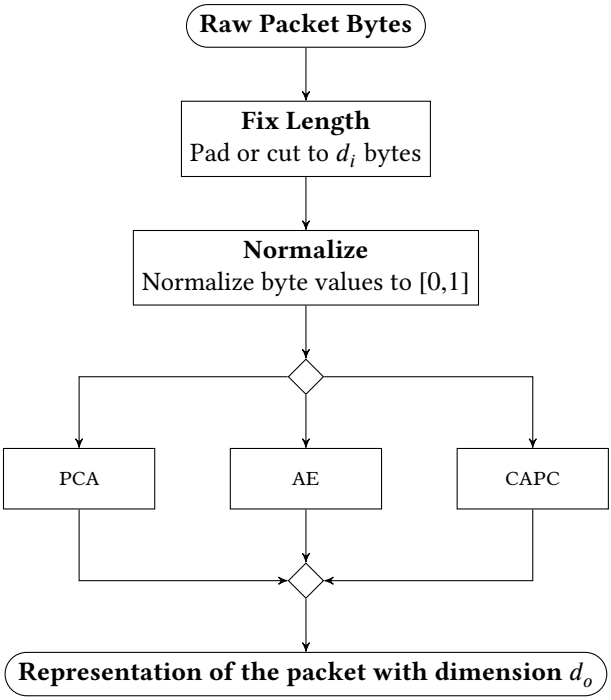


Figure 6.5: Preprocessing pipeline for NeDaP. The raw packet bytes are cut or padded to a fixed length of d_i, the values are normalized, and then the dimensions are reduced by either PCA, our AE, or CAPC. The input dimension d_i and the output dimension d_o are varied in our experiments ($d_i \in \{304, 1504\}, d_o \in \{8, 24, 48, 64\}$)

6.4.2 Approach

Our approach presented below, NeDaP, builds upon the aforementioned analysis and integrates multiple preprocessing steps to turn raw network packets into a representation suitable for utilization by ML models. The full preprocessing pipeline, including the different options for dimensionality reduction, is shown in Figure 6.5. In the figure, the rounded rectangles represent inputs or outputs of the process, while the rectangles show processing steps.

The input for the preprocessing pipeline are raw packet bytes as they are read from the network. In contrast to the approaches from literature, we do not alter concrete headers of the packet, since we do not assume any knowledge on the communication protocols used in the packet. Note however that this means that we are including the MAC and IP address of the communication partners in the input data for the preprocessing steps. In contrast to the two approaches from literature, we focus on a communication setting in which there are only two communication partners (TD and SuT), and do not focus on traffic from a whole network. Thus, the information on the MAC or IP address can only help to distinguish between the TD and the SuT, which does not leak unintended information to the preprocessing approach, as this information is already known in the fuzzing setting. In the domain of network traffic classification that is targeted by the approaches from, however, the MAC or IP address can leak unintended information. For example, if one IP address always only sends traffic coming from one application, the downstream ML model could learn to directly classify all traffic from this address to this specific application and thus overfit on the dataset.

The first step of preprocessing is to either cut or pad the packet to be of a fixed size of d_i bytes. In line with literature, we cut the packets that are too long starting from the end of the packet, and pad packets that are too short with zeros. Subsequently, we normalize the d_i bytes and feed the resulting data in three different algorithms for dimensionality reduction. We choose a PCA, a vanilla AE as presented in the following paragraph, and the CAE-based CAPC as presented by Chiu et al. [Chi20]. The output of the preprocessing is a representation of the initial packet with a reduced number of dimensions, d_o. In our experiments, we vary the input dimension d_i and the output dimension d_o to analyze the impact of this choice on the performance of the dimensionality reduction. We derive the possible choices for d_i and d_o from our analysis as presented in Section 6.4.1.

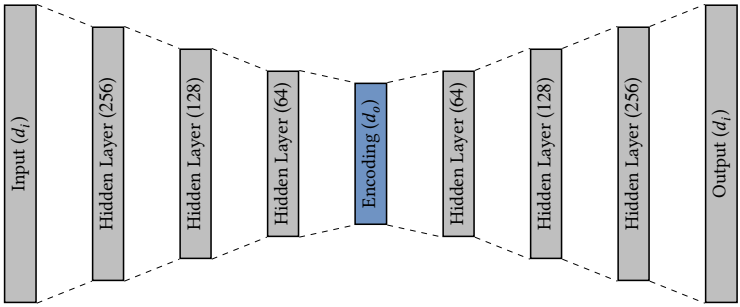


Figure 6.6: Visual representation of the architecture of the AE used as one approach for dimensionality reduction in NEDAP. The dimension of the encoding layer is equal to the output dimension of the dimensionality reduction, denoted as d_o.

Autoencoder

We choose a basic AE as one of the algorithms for dimensionality reduction, serving as a baseline for the more complex AE CAPC. Thus, we choose a basic encoder-decoder architecture for this AE, as presented in Figure 6.6. The encoder consists of three dense hidden layers, each using the ReLu activation function [Gér22]. We choose the following dimensions for the hidden layers: 256, 128, 64. The dimension of the encoding layer is equal to d_o, the output dimension of the preprocessing pipeline. Symmetric to the encoder, the decoder consists of three dense layers using the ReLu activation function.

Implementation

We implement the preprocessing pipeline using the popular Python ML libraries Tensorflow[1] 2.15.0, Keras[2] 2.15.0, and scikit-learn[3] 1.4.1. For PCA, we use the PCA implementation as provided by scikit-learn in the decomposition module (`sklearn.decomposition`). To construct AE, we use the corresponding modules provided by keras, namely the dense and the sequential layers

[1] https://www.tensorflow.org/
[2] https://keras.io
[3] https://scikit-learn.org

provided in `tensorflow.keras.layers`. For CAPC, we use the description provided in the corresponding paper [Chi20], and construct it similar to AE using the appropriate keras layers. To ensure reproducibility, we construct all of our models with fixed seeds for the used random functions. For each repetition of the experiments, we use a different but fixed seed to account for the influence of the seed.

6.4.3 Experiments

The objective of our experiments is to analyze the performance of the preprocessing pipeline, with a special focus on the performance of the dimensionality reduction approaches. For this, we define several configurations by varying the datasets, the input dimension d_i, and the output dimension d_o. As stated before, we use a PCA, an AE, and the CAE CAPC. Since the PCA does not include learning steps, we can use this as a baseline for the two learning-based approaches AE and CAPC.

We focus our experiments on the user data and fuzzing datasets as described above, which consist of FTP network traffic. We choose FTP since it is a plaintext network protocol used in office and industrial networks, and is often used as fuzzing target (see e.g. [Gas15, Nat21, Liu22, Nat22]). Thus, we can reuse existing frameworks to generate the necessary network traffic and generate comparable results, while still recognizing the industrial use case. Nevertheless, our approach NeDaP is not tailored to FTP but is agnostic of the used network protocol.

Our experiments demonstrate that CAPC outperforms PCA and AE if the training and evaluation of the dimensionality reduction approaches is conducted on the same dataset. However, if the approaches are trained on user data and are then validated on fuzzing data, AE outperforms the other two approaches. Moreover, we show that d_i has an impact on the training time of the dimensionality reduction approaches as well as the reproduction error. Our experiments also confirm that d_o impacts the reconstruction loss such that a higher output dimension d_o leads to a smaller reconstruction loss. This is the expected behavior and highlights the soundness of our experiments.

6.4.3.1 Experimental Setting

We run our experiments on an Ubuntu 23.10 server with an AMD Ryzen Threadripper PRO 5975WX CPU (32 physical cores) with 128 GB of RAM and an NVIDIA GeForce GTX 1060 6GB graphics card (Cuda version 12.3).

Research Questions

Our experiments presented in this section are driven by the following research questions.

RQ1 How do the dimensionality reduction approaches perform in terms of reconstruction error when applied to different network traffic datasets?

RQ2 How do dimensionality reduction approaches trained on FTP user data perform in terms of reconstruction error when applied to fuzzing data?

RQ3 What is the effect of the output dimension d_o on the reconstruction error?

RQ4 What is the effect of the input dimension d_i on the reconstruction error?

Methodology

We train and evaluate each dimensionality reduction approach (PCA, AE, and CAPC) on each of the datasets using a 10-fold cross validation with a $80:20$ split between train and test data. Both NN-based approaches are trained for a fixed number of 200 epochs. In order to achieve reproducible results, we use fixed seeds for the various randomness functions used during the construction and evaluation of the dimensionality reduction approaches. With the validation on the 20% test data, we measure the in-domain generalization of the dimensionality reduction approaches. Moreover, we validate the trained models on the remaining datasets, analyzing the approaches' out-of-domain

generalization capabilities. Especially, we are interested in how the dimensionality reduction approaches trained on user data generalize to fuzzing data. Table 6.3 shows the configurations used during the evaluation.

6.4.3.2 Results

This section presents the results of the experiments conducted as described above. These results are then discussed in Section 6.4.3.3.

Runtime

First, we report the runtime needed to train the different dimensionality reduction approaches. The timing measurements were conducted across 10 runs for each output dimension and preprocessing approach using the user data datasets for ProFTP and LightFTP. Table 6.4 presents the average of these 40 runs for each approach. Notably, CAPC stands out as the most time-consuming

Table 6.3: Dataset configurations used for the evaluation. *User Data* and *AFLnwe Fuzzing* are based on the FTP implementation of ProFTP. For comparison, we also use user data collected using the FTP implementation LightFTP. The goal of these configurations is either to analyze the in-domain performance of the dimensionality reduction approaches, or to analyze the out-of-domain performance.

ID	Cross validation	Validation	Goal
only-user	*User Data*	-	in-domain validation
only-fuzz	*AFLnwe Fuzzing*	-	in-domain validation
user-fuzz	*User Data*	*AFLnwe Fuzzing*	out-of-domain validation

Table 6.4: Training time of the dimensionality reduction approaches for the user data of ProFTP and LightFTP, mean over 40 runs (10 runs for each of the four output dimensions). Note that for PCA, no training is required.

	Training time in seconds			
	ProFTP		LightFTP	
	$d_i = 304$	$d_i = 1504$	$d_i = 304$	$d_i = 1504$
PCA	0.01 *s*	0.03 *s*	0.01 *s*	0.01 *s*
AE	19.99 *s*	19.56 *s*	8.90 *s*	8.96 *s*
CAPC	56.14 *s*	146.36 *s*	23.87 *s*	60.45 *s*

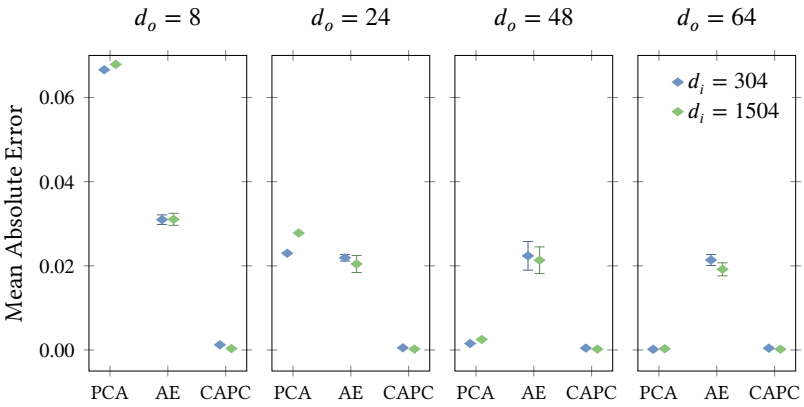


Figure 6.7: Mean absolute reconstruction error for dimensionality reduction approaches trained and validated on *User Data* (configuration *only-user*) for different choices of d_i and d_o (mean of 10 runs). The whiskers represent the upper and lower quartiles. The mean absolute error is calculated only on the original input bytes and not on the padding. Choosing $d_i = 304$ leads to smaller reconstruction errors for PCA, while choosing $d_i = 1504$ leads to slightly smaller reconstruction errors for AE and CAPC.

approach, with a training time of 56.14 seconds for the ProFTP dataset with $d_i = 304$. Conversely, PCA shows the shortest training time. This is expected, since for PCA, no NN needs to be trained. In comparison to the training times for $d_i = 304$, the training times for $d_i = 1504$ increase. Furthermore, the training times for the ProFTP dataset are higher than those for the LightFTP dataset, reflecting the difference in their sizes.

Input Dimension

Following the runtime, we present the results with respect to the two choices for the input dimension d_i, which defines the fixed length the network packets are cut or padded to. Subsequently, the network packets with fixed length are fed into the dimensionality reduction approaches. As discussed in Section 6.4.1.2, we use two choices for d_i during our experiments: $d_i \in \{304, 1504\}$.

Figure 6.7 displays the mean absolute reconstruction error of the three dimensionality reduction approaches trained and validated on *User Data*. For AE, we also add whiskers showing the upper and lower quartile respectively. We omit them for PCA and CAPC for visualization purposes, since their reconstruction errors have such a small variance that the whiskers would be smaller than the mark that represents the mean value. Each data point represents the mean of 10 runs of the specific configuration. The choices for d_i are represented by the different colors and marks, while the choice of d_o is reflected by the different graphs of the figure. For example, the graph on the far left shows the mean absolute reconstruction error for $d_o = 8$, and shows that PCA achieves a mean reconstruction error of 0.067 for $d_o = 304$, and of 0.068 for $d_o = 1504$.

Note that the different choices for d_i particularly imply different percentages of padded zeros in the input data for the dimensionality reduction approaches. Recall that more than 98% of the network packets of *User Data* are shorter than 304. Thus, for $d_o = 1504$, for most network packets, more than 80% of the input data consists of padded zeros. In order to achieve comparable reconstruction errors for $d_i = 304$ and $d_i = 1504$, we calculate the reconstruction error only on the original input length, omitting the padded parts of the input. For the remaining experiments, we calculate the error on the full input and output data of the dimensionality reduction approaches, and thus the absolute values of the error are not comparable to the other figures in this chapter.

Our experiments suggest that choosing $d_i = 304$ leads to a smaller reconstruction error for PCA. In contrast, $d_i = 1504$ leads to a slightly smaller reconstruction error for AE and CAPC. The relative performance of the dimensionality reduction approaches is not affected by d_i for $d_i \in \{304, 1504\}$.

Together with the insights given by the runtime analysis, these observations help to choose an appropriate input dimension for a given use case and given datasets. Since the following experiments focus on the relative performance of the dimensionality reduction approaches, which is not changed by d_i, and the observation that the training time is reduced for 304 (see Table 6.4), we only report the results for $d_i = 304$ in the following.

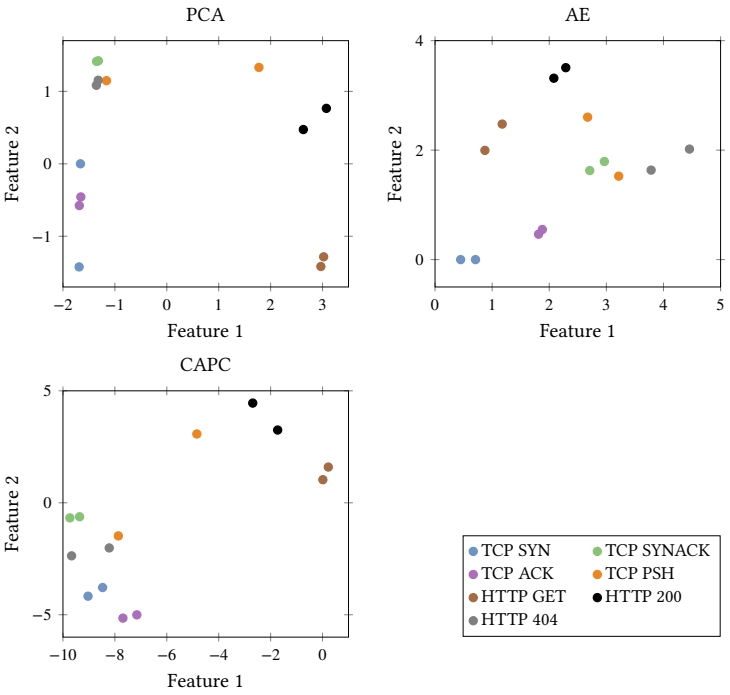


Figure 6.8: Visual representation of a two-dimensional encoding of selected packets as created by PCA, AE, and CAPC. The visual analysis shows that same packet types are generally located close to each other.

Packet Representation

To give an intuition of the performance of the dimensionality reduction approaches, we visually present the results of a dimensionality reduction to a output dimension of $d_o = 2$ in Figure 6.8. The choice for d_o in this case is based on the constraint of multidimensional visual representations not being feasible on paper. The x-axis and y-axis each represent one of the two dimensions of the encoding. Note that the absolute values of the resulting encodings are not comparable between the encodings, especially since the values of AE's and CAPC's representation do not have a frame of reference. The colors of the points correspond to the packet type of the encoded packet. For example, PCA

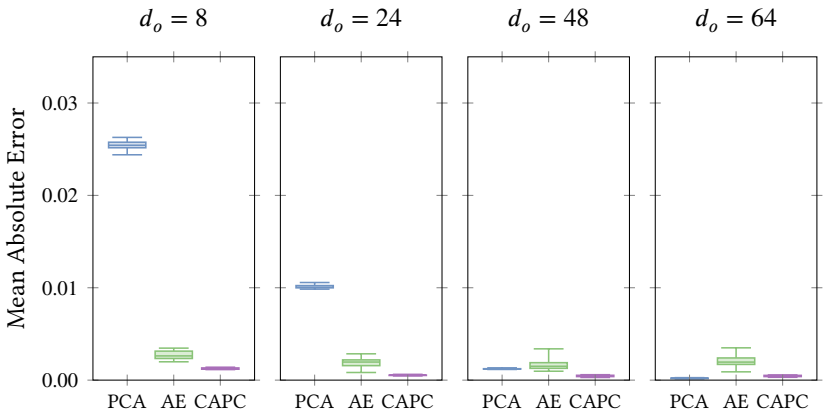
encodes the two instances of HTTP 404 packets, shown as gray dots, relatively close to each other at $(-1.36, 1.08)$ and $(-1.32, 1.15)$. AE, however, encodes them further away from each other at $(3.78, 1.64)$ and $(4.46, 2.02)$. Interestingly, all three approaches classify one of the TCP PSH packets (represented in orange) close to the HTTP 404 packets (represented in gray).

For all three dimensionality reduction approaches, it shows that packets of the same type are mostly located next to each other. From that observation, we can derive the intuition that the dimensionality reduction is able to create suitable encodings, even for an output dimension of $d_o = 2$. In the following, we analyze the performance of the dimensionality reduction approaches in a more data-driven way.

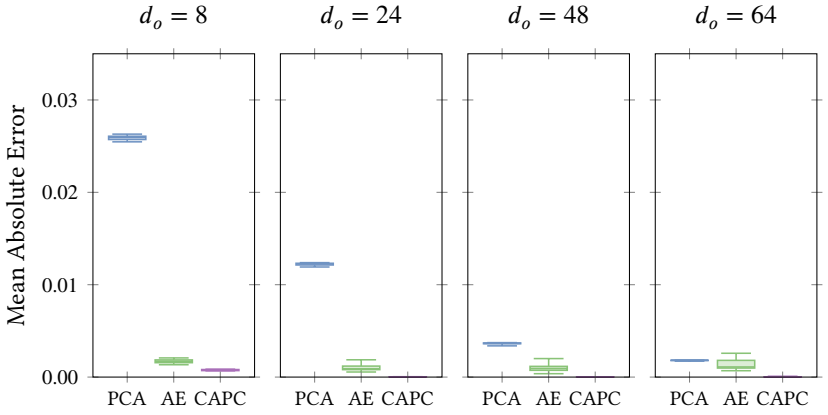
Reconstruction Error

The reconstruction errors of the three dimensionality reduction approaches for the datasets *User Data* and *AFLnwe Fuzzing* are shown in Figure 6.9. The y-axis shows the mean absolute error of the reconstruction of the input data based on the encoding of the dimensionality reduction approaches. Note that the mean absolute error is calculated on the full given input and the reconstructed values of the respective dimensionality reduction approach, both having the dimension $d_i = 304$. For the AE-based approaches, AE and CAPC, we use the respective decoder for the reconstruction. For the PCA, we use the `inverse_transform` functionality as provided by scikit-learn.

Each subplot in Figures 6.9a and 6.9b shows the reconstruction error of the three dimensionality reduction approaches for one of the output dimensions d_o. The reconstruction error is presented as a box plot, in which the box represents the area between the upper and the lower quartile, while the whiskers show the minimum and the maximum. This data is taken from ten runs of each dimensionality reduction approach for each configuration of d_i and the used dataset. Note that for this plot, we train the approaches on one dataset and then calculate the reconstruction error on the same dataset, representing



(a) Dimensionality reduction approaches trained and validated on *User Data* (ProFTP), $d_i = 304$.



(b) Dimensionality reduction approaches trained and validated on *AFLnwe Fuzzing* (ProFTP), $d_i = 304$.

Figure 6.9: Box plot showing the mean absolute reconstruction error of the dimensionality reduction approaches for different choices of d_o on *User Data* and *AFLnwe Fuzzing* (in-domain). Choosing a higher output dimension leads to a generally smaller reconstruction loss. CAPC generally leads to the smallest reconstruction errors.

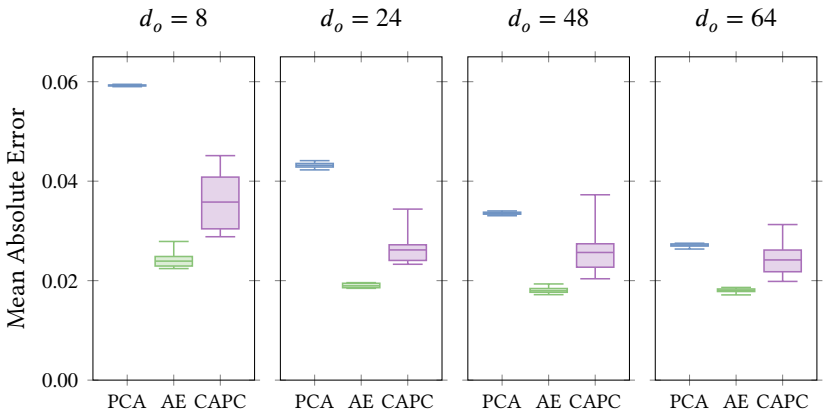
the configurations *only-user* and *only-fuzz* (see Table 6.3). As an example, Figure 6.9a shows that for *User Data*, the mean reconstruction error for PCA is 0.025 for $d_o = 8$, while it improves to 0.0002 for $d_o = 64$.

The results in Figure 6.9 show that the reconstruction error decreases with an increasing output dimension d_o. For PCA, this decrease is most apparent, while it is smaller for AE and CAPC. Moreover, it shows that the PCA generally leads to a higher reconstruction error than AE and CAPC, except for *User Data* with $d_o \geq 48$. AE generally performs worse than CAPC, and also shows a higher variance in performance.

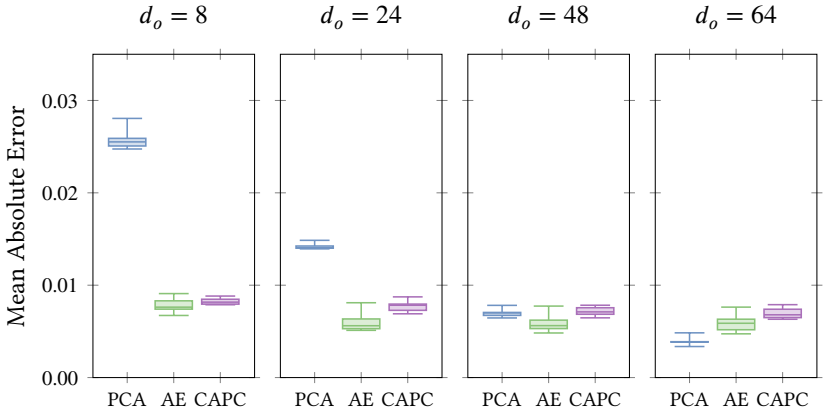
To also analyze to which extent the approaches can be generalized, we calculate the reconstruction error on the *AFLnwe Fuzzing* dataset. For this, the approaches are still trained on the *User Data* dataset. This reflects the practical approach for model-based testing, as in this scenario one might have access to user-generated data to train the models on. Subsequently, the models are to be used for model-based testing using the data generated during testing or fuzzing (see also Section 7.5). We present the results in Figure 6.10 with the same representation as the previous results. Again, we show the mean absolute reconstruction error of the dimensionality reduction to various output dimensions. Figure 6.10a shows the performance of the dimensionality reduction approaches trained on *User Data* and validated on *AFLnwe Fuzzing* (configuration *user-fuzz*). For comparison, we also show the results of approaches trained the user data recorded for LightFTP and validated on the fuzzing data for LightFTP in Figure 6.10b.

For ProFTP, our results show that the reconstruction loss of the dimensionality reduction approaches increases if we validate their performance on the *AFLnwe Fuzzing* dataset. For example, for $d_o = 48$, AE achieves a median reconstruction error of 0.018, while it achieved a reconstruction error of 0.002 for training and validation on *User Data* (*only-user*), and of 0.001 for training and validation on *AFLnwe Fuzzing* (*only-fuzz*). Note that the x-axis of Figure 6.10a differs from the one used in Figures 6.9 and 6.10b for visualization purposes.

It also shows that the area inside the boxes in Figure 6.10a are larger than the corresponding areas in Figure 6.9, meaning that the performance of the 10 runs of the cross validation are spread over a larger interval and thus the performance of the dimensionality reduction approach is less reliable. This is



(a) Dimensionality reduction approaches trained on user data and validated on fuzzing data (ProFTP, $d_i = 304$). Note that the y-axis has a different scale than the ones used in the other figures.



(b) Dimensionality reduction approaches trained on user data and validated on fuzzing data (LightFTP, $d_i = 304$).

Figure 6.10: Box plot of the mean absolute reconstruction error of the dimensionality reduction approaches when trained on user data and validated on fuzzing data. For Figure 6.10a, we use *User Data* and *AFLnwe Fuzzing*, and for Figure 6.10b, we use user data generated using LightFTP and fuzzing of LightFTP. AE generally achieves a smaller reconstruction error than CAPC.

especially apparent for CAPC. Moreover, AE outperforms CAPC for all choices of d_o, while CAPC outperformed AE in all experiments in which we used the same data for training and validation (Figure 6.9).

For LightFTP, the results also show that AE outperforms CAPC. PCA outperforms both AE-based approaches for $d_o = 64$. The absolute values of the mean reconstruction error show that it is generally easier to reduce the dimensionality of network packets in the LightFTP datasets than it is for the network packets in the ProFTP dataset.

Target Dimension

The plots presented in Figures 6.9 and 6.10 illustrate a clear dependency between the reconstruction error of the dimensionality reduction approaches and the selected output dimension d_o. They demonstrate that the reconstruction loss decreases with as the output dimension increases.

6.4.3.3 Discussion of Results

Our experiments result in new insights with respect to the research questions formulated in Section 6.4.3.1, which are discussed in the following.

RQ1 How do the dimensionality reduction approaches perform in terms of reconstruction error when applied to different network traffic datasets?

Our experiments show that CAPC outperforms PCA and AE if the training and validation is conducted on the same dataset in most cases. Only for the ProFTP dataset with $d_o = 64$, CAPC is outperformed by PCA. This implies that CAPC generalizes better than the other two approaches in an in-domain evaluation. Interestingly, CAPC finds an efficient encoding for the network packets even for an output dimension of $d_o = 8$, showcasing its strength in encoding the network packets from the *User Data* dataset.

Moreover, we show that the training time needed for the dimensionality reduction approaches varies greatly, as CAPC needs around 2.8 times longer than AE and 5,600 times more than PCA for the *User Data* dataset. However,

the importance of the training time depends highly on the use case. If the dimensionality reduction approaches are re-used for several model building purposes, the training needs to be done only once and thus a longer training time is not as significant.

RQ2 How do dimensionality reduction approaches trained on FTP user data perform in terms of reconstruction error when applied to fuzzing data?

In contrast to the previous observations, AE outperforms the other two approaches in most cases when the approaches are trained on user data and validated on fuzzing data. AE is only outperformed by PCA for $d_o = 64$ on LightFTP data. This implies that the AE generalizes better for out-of-domain tasks, while CAPC generalizes better for in-domain tasks, as discussed in *RQ1*.

Moreover, the reconstruction loss of CAPC shows a higher variance for the validation on fuzzing data, especially on LightFTP. One explanation for this could be that CAPC, caused by the convolutional layers, focuses on local correlations in the network packet to perform the dimensionality reduction. In fuzzing however, this local correlations are likely to be broken, especially in the payload of the network packet. With this, it is a bigger challenge for CAPC to reconstruct the fuzzed network packets. In contrast, AE most likely captures less detailed information of the network packet, due to the reduced number of parameters and structure the AE has in comparison to the CAE used by CAPC. With this, it generalizes better to the fuzzing dataset.

RQ3 What is the effect of the output dimension d_o on the reconstruction error?

As expected, our experiments show that the output dimension has an impact on the absolute reconstruction error of the dimensionality reduction, as the absolute error decreases with an increasing output dimension. This decrease is most apparent for PCA. Moreover, we show that the output dimension does not have an impact on the relative performance of the dimensionality reduction approaches in most cases. For example, for the configuration *only-fuzz*, where the approaches are trained and validated on *AFLnwe Fuzzing*, the median

reconstruction error of PCA is always higher than the one of AE, which again is always higher than the one of CAPC. However, for *only-user*, PCA outperforms AE and CAPC for $d_o = 64$.

RQ2 What is the effect of the input dimension d_i on the reconstruction error?

Our experiments with input dimensions $d_i \in \{304, 1504\}$ show an inconsistent influence on the reconstruction error of the dimensionality reduction approaches. For PCA, an input dimension of 304 leads to a smaller reconstruction error for all choices for the output dimension d_o. In contrast, it leads to slightly higher reconstruction errors for AE and CAPC. Moreover, the choice of the input dimension d_i impacts the training time of the dimensionality reduction approaches. A higher input dimension increases the training time, which is to be expected, since a higher input dimension leads to a higher number of trainable parameters.

6.5 Discussion

This section puts the results of NeDaP into context. It discusses the implications of NeDaP (Section 6.5.1), its limitations (Section 6.5.2) and possible future research directions (Section 6.5.3).

6.5.1 Implications

To the best of our knowledge, NeDaP presents the first performance analysis of dimensionality reduction approaches with respect to network packets. Even though we put a special focus on preprocessing the network packets for model-based fuzzing in the industrial domain, our results provide insights for other use cases of preprocessed network packets such as traffic flow classification or intrusion detection. Moreover, we created several datasets which reflect different use cases in the domain of industrial security, including fuzzing, web security testing, and user interaction traffic. These datasets could build the base for further data-driven analyses in this domain.

Building upon these experiments and results, we use the preprocessing pipeline presented in this chapter to perform blackbox model-based fuzzing using HMMs. Especially, the experimental results presented in this chapter are used to balance possible choices for hyperparameters such as d_i and d_o. This HMM-based testing approach, called `Palpebratum`, is presented in Section 7.5.

6.5.2 Limitations

In line with literature, we cut the input data to the fixed length d_i by dropping the *last* bytes of the network packet. However, these bytes usually represent parts of the payload of the network packet which might be important for the subsequent model which uses the preprocessed data. Thus, the contrary approach of dropping the *first* bytes of the network packet to cut it to d_i might be beneficial and could be investigated in future work.

For our evaluation, we mostly focus on FTP. This network protocol can be fuzzed easier than other protocols since it is a plaintext protocol, it occurs in office networks as well as in industrial networks, and is often used as fuzzing target in literature. However, since `NeDaP` is agnostic for the used network protocol, our evaluation could be extended by datasets including different industrial protocols to analyze the generalization capabilities of `NeDaP` in a more thorough way.

The work presented in this chapter focuses on the performance of the dimensionality reduction approaches. For a full evaluation of their performance however, an evaluation including the downstream application is necessary. It is possible that a dimensionality reduction approach with a higher reconstruction error still yields a good representation for the downstream application, the HMMs in our case. To this end, we extend upon the evaluation shown in this chapter by using the presented preprocessing pipeline to perform HMM-based fuzzing in Section 7.5.

6.5.3 Future Work

In future work, the experiments presented in this chapter could be extended by varying more parameters, adding more datasets, and adding more approaches for dimensionality reduction.

One parameter that could be varied is the interval the bytes of the network packets are normalized to during preprocessing. For this work we chose the interval [0,1] based on related work, but it is possible that other intervals, such as $[-1,1]$, or other normalization approaches might be better suited for some of the dimensionality reduction approaches [Cab23]. It might also be beneficial to include more choices for the input dimension d_i to the experiments, in order to further analyze the impact of this choice. Furthermore, the impact of cutting the network packets to d_i could be investigated further. As described in Section 6.5.2, we cut the last bytes of the packets to achieve the desired length. However, it might be beneficial to cut the first bytes of the packets instead since these bytes usually contain header data that might be less relevant for the models that are using the preprocessed data afterwards.

Moreover, one could extend the experiments to more datasets, and datasets from other domains. One interesting generalization analysis would be to include user data and fuzzing data that is not focussed on one single protocol, but includes several different protocols. As detailed in Section 6.4.2, the approach of NeDaP does not assume any information on the used network protocol and thus can easily be extended to other protocols.

Furthermore, additional approaches for dimensionality reduction could be included to the experimental setup. For instance, one could include insights and approaches from transformer models [Vas17] to the approach of NeDaP. These have been used successfully for intrusion detection tasks based on network traffic flows [Wu22b, Wan21a] and thus their encoding layer might also provide benefits for the dimensionality reduction tasks needed for network packet preprocessing.

The experimental setup used for NeDaP could also be used to approach additional research questions. For instance, it would be interesting to analyze whether dimensionality reduction approaches that have been trained on the user data of one FTP implementation could be reused for preprocessing network data generated by other FTP implementations. If this could be done without a significant performance drop, one would only need to train the dimensionality reduction approaches once to train several implementations of the same protocol.

6.6 Summary

With NeDaP, we present a full preprocessing pipeline for network packets, including three different dimensionality reduction approaches. We acknowledge the special requirements of the industrial use case by presenting an approach that is agnostic of the used network protocols. With this, NeDaP can be used without changes for industrial communication protocols, especially for proprietary protocols.

In our experiments, we analyze the performance of this preprocessing pipeline, with a special focus on the performance of the dimensionality reduction in terms of the reconstruction loss. We demonstrate that CAPC consistently outperforms AE and PCA across most choices of the output dimension d_o when trained and evaluated on one dataset. However, when we train the dimensionality reduction approaches on FTP user data and then validate their performance on fuzzing data, AE outperforms the other two approaches in most cases. The latter scenario mirrors a more realistic setting for model-based blackbox testing, where access to a limited set of user traffic data is a more practical assumption than access to a comprehensive dataset of fuzzing data.

Furthermore, our experiments underscore the trade-off with respect to the parameter selection, particularly concerning the preprocessing of network packets intended for use in ML models. On the one hand, a smaller output dimension decreases the number of trainable parameters in the downstream

ML model, thereby simplifying the training process. On the other hand, a smaller output dimension also decreases the quality of the data, as shown by the increased reconstruction error in our experiments.

7 Machine Learning based Blackbox Fuzzing

This chapter focuses on utilizing the information available in blackbox testing by leveraging Machine Learning (ML) models. As stated before, prior to and during blackbox testing, only limited information is available. However, this limited information could be represented adequately by a model, which then could be utilized to improve the blackbox testing efficiency.

In general, models are used in various different ways to improve the development and testing in various industrial domains, such as generating digital twins based on Hidden Markov Models (HMMs) [Gho19], testing embedded systems [Böh11, Zan17], detecting faults in Cyber-Physical Systems (CPSs) [Mai14], and model-driven security engineering of CPSs [Gei20]. Moreover, testing approaches utilize models, e.g. to represent the state of the System under Test (SuT) state in stateful testing [Nat22, Gir20, Dou12], or to represent the structure of the SuT's input [Vii08, Gas15]. Felderer et al. give a general overview of model-based security testing [Fel16b].

Moreover, ML approaches have been increasingly employed to improve security testing in general, with a particular focus on improving fuzzing. Especially in graybox fuzzing, ML approaches have, for example, been used for seed selection [Che20, Wan21b], mutation selection [Böt18], and test case generation [Wan17]. Wang et al. and Chafjiri et al. give an overview of research with respect to ML for fuzzing [Wan20a, Cha24].

However, the majority of existing approaches focus on graybox fuzzing. We identify the possibility to apply ML approaches to represent the information available in blackbox testing, which then can be used to guide the fuzzing process. More specifically, we focus on mutational blackbox fuzzing for Operational Technology (OT) components.

7.1 Problem Statement

We propose using ML techniques to make the limited information in blackbox fuzzing more accessible to a fuzzer. For this, several information sources such as the test cases sent to a SuT, the responses of the SuT, or the entire network traffic generated during fuzzing can be leveraged. Based on this information, various characteristics of the SuT's behavior can be modeled. The concrete representation of the SuT's behavior by the ML model and its utilization to guide the fuzzing process is to be determined and is explored in the following.

7.2 Contributions

This doctoral work advances the field of blackbox testing by utilizing ML models across three different approaches. Each of these approaches is focused on network fuzzing, with their characteristics being detailed in Table 7.1. A more general approach to model-based blackbox testing was registered as a patent in collaboration with Steffen Pfrang and Christian Haas [*Bor23d*].

`Smevolution` The first approach, `Smevolution`, combines evolutionary blackbox fuzzing with an ML model which approximates the function mapping a test case t to the services of the SuT that crash in response to t. This model is trained using the crash information collected during testing and guides the evolutionary fuzzing by providing additional information for mutation and test case selection. Based on this general approach, we implement three concrete instances of `Smevolution`, employing a Neural Network (NN), a Decision

Table 7.1: Comparison of the three contributions of this doctoral work with respect to ML-based blackbox testing. The work on `Palpebratum` will be published separately in the future.

	`Smevolution`	**`Palpebratum`**	**`StateBandit`**
Approach	Blackbox	Blackbox	Blackbox
Model	Explicit	Explicit	Implicit
Target	Test case ↦ crashing services	Network traffic	State ↦ coverage
Used data	Crash information during testing	Network traffic data prior to and during testing	Code coverage and crash information during testing
Reference	Section 7.4	Section 7.5	Section 7.6
Publication	[*Bor23b*]	tbd	[*Bor23a*]

Tree (DT), and a Support Vector Machine (SVM) as ML model. Our evaluation shows that the fuzzer utilizing the DT performs significantly better than a random and a blackbox baseline fuzzer in terms of triggered vulnerabilities.

Contribution 6. *Proposal, implementation, and evaluation of* `Smevolution`*, an approach to combine evolutionary fuzzing with ML models leveraging test data to help guiding the test process.*

`Smevolution` targets a blackbox test setting (Challenge 1 (Blackbox Testing)) and is independent of the tested network protocol (Challenge 4 (Insufficient Protocol Support)). The used ML model analyzes and processes the information on test cases and crashing services to provide it to a fuzzer (Challenge 2 (Missing Information)), and could be used as external mutator by mutation-based fuzzers (Challenge 5 (Choice of Testing Tool)). Due to the integration in ISuTest®, `Smevolution` allows to monitors all communication interfaces of the SuT (Challenge 3 (Insufficient Observations)).

We call this approach `Smevolution`, thus choosing a name that combines the perceived *smartness* of the ML models with the evolutionary testing approach that we use as a basis. `Smevolution` and the corresponding experiments were published in *Sensors* Vol. 23 [*Bor23b*], and the code is publicly available on

GitHub[1]. The work on `Smevolution` was conducted in collaboration with Steffen Pfrang and Martin Morawetz, and is based on the Master's thesis by Martin Morawetz [*Mor21*].

`Palpebratum` Furthermore, we present an approach that relies on the network traffic between the Test Device (TD) and the SuT, and leverages this traffic to provide information on the SuT's behavior to a network fuzzer. To this end, we train an HMM on network traffic data recorded prior to the security test. During testing, we query this model to receive information on the most likely sequence of hidden states that lead to the observed network traffic. With this, we receive an approximation of the SuT's behavior represented by a list of paths hit by a test case. As this information can be used exchangeable with the code coverage information used by graybox network fuzzers, it can be utilized transparently. Thus, the novel approach allows for applying graybox testing approaches in a blackbox test scenario.

Our experiments demonstrate that the HMM-based fuzzers are able to generate test cases that are more efficient than the test cases generated by the baseline fuzzers. However, the HMM-based fuzzers achieve significantly less coverage than the two baseline fuzzers. Possible explanations are an underestimation of the coverage achieved by the HMM-based fuzzers and differences in efficiency in terms of the number of test cases generated in a fixed time frame.

Contribution 7. *Proposal, implementation, and evaluation of `Palpebratum`, an approach to approximate the SuT's behavior based on preprocessed network traffic, facilitating graybox fuzzing approaches in a blackbox test setting.*

`Palpebratum` is designed to be used in a blackbox test setting (Challenge 1 (Blackbox Testing)) and is independent of the underlying network protocols (Challenge 4 (Insufficient Protocol Support)). The HMM is used to approximate information that would otherwise be missing in a blackbox test scenario (Challenge 2 (Missing Information)). This approximated information can be used by graybox coverage-based fuzzers (Challenge 5 (Choice of Testing Tool)).

[1] https://github.com/anneborcherding/Smarter-Evolution

The name of this approach, `Palpebratum`, is based on a fish species called *Photoblepharon palpebratum*[1]. These fish use bioluminescent organs below their eyes for several means such as attracting and finding pray, confusing predators, and communicating with other fish. Especially, these fish use their bioluminescent organs to bring at least some light into the darkness of the ocean. Our HMM-based approach has a similar objective by aiming to "bring more light" into *blackbox* network fuzzing and shifting it more into the direction of *graybox* network fuzzing.

The work on `Palpebratum` is partly based on the Master's thesis by Robert Mumper [*Mum21*], the Bachelor's thesis by Deniz İmge Avcı [*Avc23*], and the seminar work by Johannes Häring [*Här23*], all supervised during this doctoral work. Most of the work presented in this section is to be published in the future.

`StateBandit` With this approach, we leverage a Reinforcement Learning (RL) agent to address the state selection problem in stateful graybox network fuzzing. In stateful fuzzing, the fuzzer has access to a state machine and uses this state machine to guide its fuzzing process. Amongst others, the fuzzer needs to decide in which state the SuT should be fuzzed next. For this, a balance between exploration and exploitation is needed, which we approach by using RL, more specifically a Multi-armed Bandit (MaB), agents. We evaluate our approach with four different algorithms for the bandit's policy and show that all four algorithms lead to statistically indistinguishable performances in terms of code coverage. Moreover, our experiments show that the MaB-based approaches achieve significantly less coverage than the current state-of-the-art stateful graybox fuzzer `AFLNet`.

Contribution 8. *Proposal, implementation, and evaluation of `StateBandit`, an approach to delegate the state selection problem to a MaB agent.*

`StateBandit` operates independent of the network protocol that is being tested (Challenge 4 (Insufficient Protocol Support)), and helps to provide additional guidance and information for the test (Challenge 2 (Missing Information)).

[1] https://en.wikipedia.org/wiki/Photoblepharon_palpebratum

We call our approach leveraging an Multi-armed *Bandit* agent to solve the *state* selection problem `StateBandit`. The approach and experimental results were published with the EuroS&P workshops [*Bor23a*]. The work on `StateBandit` was conducted in collaboration with Mark Giraud and Ian Fitzgerald, and is based on the Bachelor's thesis by Ian Fitzgerald [*Fit22*].

7.3 Related Work

The approaches presented in this chapter focus on applying ML techniques to blackbox fuzzing. Accordingly, we provide an overview of related work in this area, specifically focussing on blackbox approaches. For a broader perspective on ML applications in fuzzing, refer to studies such as the ones presented by Wang et al., Zhang et al., and Chafjiri et al. [Wan20a, Zha23a, Cha24]. Additionally, Maximilian Kühn's Bachelor's thesis, supervised during this doctoral work, analyzes existing applications of ML within the fuzzing process [*Küh23*]. For a more detailed discussion of work related to the specific approaches presented in this chapter, refer to the corresponding sections (Sections 7.4.3, 7.5.5, and 7.6.3).

Most blackbox fuzzing approaches leveraging ML employ the ML technique to learn the SuT's input features to use them for test case generation. Gascon et al. present PULSAR [Gas15] which utilizes an HMM to learn the state machine of a network protocol and the structure of the network packets (see Chapter 3 and Section 7.5.5 for more details on PULSAR). This representation is then used to generate new test cases, but also to guide the fuzzing process by utilizing the learned state machine. Fan et al. and Zhao et al. utilize a Long Short-Term Memory Network (LSTM) to represent the network traffic and to generate new network packets as test cases [Fan18a, Zha19]. Lv et al. combine an LSTM with techniques from Generative Adversarial Networks (GANs) to generate network packets as test cases [Lv21]. Note that these approaches consider test case generation for generational fuzzing, and thus the ML model is directly used to generate new test cases for the SuT. In contrast, the approaches presented in this chapter are concerned with mutational fuzzing and leverage the ML model to guide the mutational fuzzing process.

Appelt et al. utilize a DT to guide the mutation of a blackbox fuzzer targeting Web Application (WA) firewalls [App18]. We adapt this approach for one of the specific implementations of our evolutionary fuzzing framework `Smevolution` presented in Section 7.4. The corresponding section on related work includes additional details on the work by Appelt et al. (Section 7.4.3).

Lin et al. present `IPCFuzzer` which uses an LSTM to generate initial test cases which are then mutated [Lin21]. Moreover, `IPCFuzzer` uses weighted mutations and chooses those mutations with a higher weight more often. However, these weights are not influenced by an ML model, but are chosen based on a heuristic.

7.4 Evolutionary Fuzzing

In this section, we present `Smevolution`, our approach to leverage ML models to approximate the behavior of a blackbox SuT for evolutionary fuzzing. In evolutionary fuzzing, an Evolutionary Algorithm (EA) is deployed, in which the individuals usually represent the test cases, or seeds, of the fuzzer [Man19]. First, the individuals in the population are mutated to generate new offsprings. Then, the individuals in the offspring are ranked based on a fitness function and the most promising individuals are selected to form the new population.

With `Smevolution`, we use ML models to approximate a function mapping a test case t to the services of the SuT that crash if t is presented to the SuT. This model is trained online during the fuzzing process, utilizing the data generated by sending test cases to the SuT and observing crashing services. Then, we use this model (1) to guide the mutation of the EA, and (2) to facilitate the ranking of test cases in the EA. With this, we allow for a guided fuzzing process in a blackbox setting, leveraging the information that are available in a blackbox test setting.

To evaluate our approach, we implement three fuzzers, each utilizing the new approach with a different ML model, specifically a DT, an SVM, and an NN. As baseline, we use two fuzzers that employ the same evolutionary approach, but use predefined heuristics for the mutation of test cases. The first

baseline fuzzer uses the number of crashed services as fitness value used for test case selection, while the second baseline fuzzer selects the test case from the offspring randomly. With this, we can analyze how the trained ML models perform when compared to (1) an approach that directly uses the available information, and (2) a random approach. Moreover, we evaluate the impact of the granularity of the feedback concerning the SuT's crashing services that the fuzzers receive. To evaluate the efficiency and the overhead of the fuzzers, we also evaluate the number of test cases the different fuzzers produce and set this in relation to the findings of the fuzzers.

Our experiments show that the fuzzer based on a DT is able to perform significantly better than the two baseline fuzzers in terms of found crashes. The fuzzer based on an NN significantly outperforms the random fuzzer. Moreover, we show that the dimension and interval of the feedback the fuzzers receive does not significantly impact the final performance of the fuzzers. Due to the overhead introduced by the ML models, the newly presented approaches lead to a reduced number of test cases that are sent per time. However, since two of the newly presented approaches significantly outperform the random baseline fuzzer on a fixed time budget, this shows that the fuzzers are able to generate their test cases more efficiently.

7.4.1 Approach

Our general approach is to combine an EA for test case generation with an ML model, supporting the decisions necessary within the EA. The ML model is tasked with mutating and selecting the test cases in a short-term manner, while the EA provides a feedback loop and long-term improvements. Figure 7.1 shows a graphical representation of this approach. The underlying EA is represented by the black elements of the figure, while the newly added ML model is shown in green, and the SuT is shown in blue. In the following, we give more details on the different aspects of `Smevolution`.

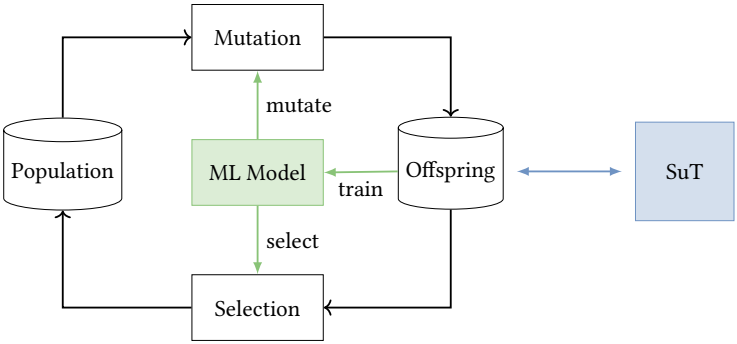


Figure 7.1: Overview of `Smevolution` based on [*Bor23b*]. The EA represented by the black elements is enhanced by an ML model, which support the mutation and the selection steps. The test cases in the offspring are executed against the SuT, thus yielding labeled data to perform the online training of the ML model.

7.4.1.1 Evolutionary Algorithm

In each evolutionary round of an EA, new individuals are generated by mutating the individuals that are in the population at the beginning of the round, forming the new offspring. Then, the most promising individuals are selected from the offspring based on a fitness function. These selected individuals form the new population. The following describes how we apply the general concept of an EA to our use case of blackbox network fuzzing. Refer to the work by Bartz-Beielstein et al. for a detailed description on EAs [Bar14].

Individuals The population and the offspring of the EA consist of test cases for the SuT. Driven by our focus on blackbox network protocol fuzzing, these test cases specify values for the fields of the network packet being fuzzed. For example, if we are fuzzing two fields of a network packet, one containing an unsigned Integer and the other a string, a test case might look like this:

$$[\ 73412, \text{“AAAA”}\ ] \tag{7.1}$$

Mutation New individuals are created by mutating individuals from the population. The objective of the mutation in the fuzzing context is to generate new test cases that reveal new anomalies in the behavior of the SuT. In a

blackbox test setting, mutations are generally performed randomly [Man19]. Common approaches to test case mutation include replacing, deleting, or adding parts of the test case [Tri23]. For instance, a mutated test case produced by flipping a bit in the last character of the test case shown in Equation (7.1) might look as follows:

[73412, “AAAC”] (7.2)

Execution The offspring is executed against the SuT, and the SuT’s behavior is observed by monitoring the SuT’s services. With this, we receive information on which services of the SuT crashed in response to sending a specific test case. This information is used to assess the fitness of each test case.

For example, assume that we aim to test a SuT including the three services Simple Network Management Protocol (SNMP), Transmission Control Protocol (TCP), and Hypertext Transfer Protocol (HTTP). If executing the test case from the previous example crashed the SNMP service, this would be represented by the bitstring b'100', indicating that the first service of the SuT crashed. With this, the labeled test case would appear as follows:

[[73412, “AAAC”], b'100'] (7.3)

Selection From the offspring, the individuals with the highest fitness are selected to form the new population. In the context of fuzzing, the fitness function used for this selection evaluates how likely a test case will help to uncover new anomalies in the future.

7.4.1.2 ML Model

We enhance the previously described EA by leveraging an ML-based representation of the SuT’s behavior that we learn during the fuzzing of the SuT. While the following description is focussed on our general approach of using an ML model to enhance the EA, Section 7.4.2.3 details the concrete ML models that we implemented and how they are used to improve the mutation and the selection step of the EA.

In `Smevolution`, we represent the behavior of the SuT as a function mapping a test case to the services of the SuT that crash after executing the test case. This function is approximated by an ML model and can be denoted as follows.

$$f : H^n \rightarrow \{0,1\}^m \tag{7.4}$$

In this equation, H represents the set of all possible hexadecimal values, n is the length of a test case, and m is the number of services of the SuT being monitored. Thus, this function maps a test case in its hexadecimal representation to a bitstring indicating which services crash after executing the test case.

Training The ML model is trained in an online fashion. It uses the labeled data generated by executing the offspring of the EA against the SuT. In this step, all test cases in the offspring are executed and the SuT's services are monitored. Thus, we receive labeled data consisting of a test case and the bitstring representing the crashed services, similar to the example provided in Equation (7.3). This labeled data is then used to train the ML model. Note that the information on crashing services can be received with differing granularity. On the one hand, services can be monitored after each test case, resulting in detailed information on the services for each test case. On the other hand, the services can be monitored after a fixed number of test cases, restricting the information to whether the set of test cases sent leads to crashing services. The latter approach increases the speed of the fuzzing process, as the monitoring of services takes a significant amount of time.

Application The ML model is used in two different steps of the EA: selection and mutation. For our use case, we define the fitness of a test case based on the number of services that it is expected to crash. The higher the predicted number of crashing services, the higher the fitness value of a test case. Moreover, we use the ML model to provide guidance for the mutation step. For example, for our experiments, we implement a fuzzer based on a DT, and one based on an NN. While both ML models are used to determine the position and the direction

of the mutation that should be conducted to improve the test case, the specifics depend heavily on the type of ML model used. We present more details on the specific models that we implement for the experiments in Section 7.4.2.3.

7.4.2 Experiments

The following describes the experiments we conduct to assess the performance and implications of `Smevolution`. We present our research questions in Section 7.4.2.1 and our methodology in Section 7.4.2.2. Section 7.4.2.3 details the fuzzers that we implement based on `Smevolution`. The model-based fuzzers use the following ML models: DT, NN, and SVM. Moreover, we implement two baseline fuzzers. Additionally, we describe our evaluation target based on `VulnDuT` in Section 7.4.2.4. The results of our experiments are presented in Section 7.4.2.5 and discussed in Section 7.4.2.6.

7.4.2.1 Research Questions

The experiments presented in this section are driven by the following research questions.

RQ1 Can the ML models used with `Smevolution` represent blackbox information that can be utilized to improve evolutionary fuzzing?

To address this question, we compare the performance of three model-based fuzzers using the approach of `Smevolution` against two baseline fuzzers, which only make use of the evolutionary aspect of `Smevolution`, but do not include a model (see also Section 7.4.2.3). This baseline consists of (1) a fuzzer that makes random decisions instead of consulting a model, and (2) a fuzzer that uses the raw information on the number of crashed services for the selection.

RQ2 How much does the granularity of the available information influence the performance of fuzzing?

We examine how varying levels of granularity in the feedback influence the performance of the model-based fuzzers. On the one hand, we analyze the impact of how many test cases are sent between the monitoring steps of the fuzzer. On the other hand, we analyze how aggregating the information on the responsiveness of the SuT's services influences the performance. This analysis aims to determine the amount of information required to achieve better fuzzing results.

RQ3 How much overhead do the ML models introduce?

The models introduce an overhead compared to fuzzers that do not use ML models. However, if the fuzzing performance is improved, this overhead might be acceptable. We address this topic by measuring the number of test cases the model-based fuzzers can generate within a given time frame compared to a random fuzzer. Note that our approach does not introduce an initial overhead, since the ML models are trained online and thus do not require a training phase before the fuzzing begins.

7.4.2.2 Methodology

We apply the following methodology to address the previously presented research questions. For our experiments, we implement three model-based fuzzers that follow the approach presented in Section 7.4.1, called A_DT, A_NN, A_SVM. Each of these fuzzers utilizes a different ML model. In addition, we implement two baseline fuzzers. While the baseline A_RAND implements a random selection for the EA, the baseline A_BASE uses the number of services that crashed for a specific test case directly as a fitness function to select the test cases. Both baseline fuzzers use predefined heuristics for the mutations. As a basis for our implementation, we use the security testing framework ISuTest® (see Section 2.2.3). More details on the implemented fuzzers can be found in Section 7.4.2.3.

Each of the configurations of the fuzzers defined below are run against VulnDuT, a deliberately vulnerable SuT where we can control the exhibited vulnerabilities (see Section 7.4.2.4). We run each fuzzing campaign for 24

Table 7.2: Configurations used for our experiments. *Fuzzer* lists the fuzzers used (see Section 7.4.2.3). *Feedback Dimension* indicates the type of feedback the fuzzer receives: either detailed information on the crashed services or binary information indicating whether at least one service crashed. With *Feedback Interval*, we define the frequency at which the fuzzer receives feedback, expressed as the number of test cases between each feedback cycle. The default value is underlined.

Parameter	Values
Fuzzer	A_DT, A_SVM, A_NN, A_BASE, A_RAND
Feedback Dimension	Multidimensional, Unidimensional
Feedback Interval	10, 50, 500

hours, as suggested by Klees et al. [Kle18], and repeat each configuration for 10 times. Note that this is less than the recommended 30 times [Kle18], which is due to resource restrictions. During fuzzing, we measure the number of vulnerabilities that have been triggered by the respective fuzzer. Since we control VulnDuT, we can directly measure how many of the exhibited vulnerabilities are triggered, and do not need to rely on proxy metrics (see e.g. Section 8.4 or [Böh22, Sch24]).

We run our experiments on an Ubuntu 20.04.6 LTS with an Intel® Xeon® Silver 4216 CPU @ 2.10 GHz (12 cores) and 32 GB of RAM. We publish the results of our experiments as well as the code that we used to define the ML models to allow for reproducible results[1].

An overview of the configurations used for our experiments is given in Table 7.2. The following paragraphs detail the configurations and how they support approaching the research questions.

RQ1 - *Model Impact*

To approach *RQ1*, we compare the performance of the three model-based fuzzers and the two baseline fuzzers with respect to the number of vulnerabilities that they trigger over the course of the 24 hour fuzzing campaigns. With

[1] https://github.com/anneborcherding/Smarter-Evolution

this, we can analyze how the utilization of the different ML models influences the fuzzing performance. Additionally, we assess the performance of each fuzzer with respect to different types of vulnerabilities.

We hypothesize that the model-based fuzzers will outperform the two baseline fuzzers in terms of the number of triggered vulnerabilities.

RQ2 - *Information Granularity*

To address the question regarding the impact of information granularity on the fuzzers' performance, we analyze the impact of two parameters: feedback dimension and feedback interval.

For the feedback dimensions, we define the following two configurations.

1 Multidimensional Information: Using this configuration, the fuzzer receives full feedback on which services crashes for a given test case. This corresponds to the example given in Equation (7.3).

2 Unidimensional Information: With this configuration, the information concerning the crashing services is reduced to a single bit. This bit is set to 1 if at least one service crashes, 0 else.

We expect that the model-based fuzzers using multidimensional feedback trigger more vulnerabilities. This expectation is based on the design of `Smevolution` by which the mutation and selection strategies aim to maximize the number of services crashed by a test case. Thus, if the fuzzer has no information on the number of crashed services, it has less information on the optimization it is supposed to perform. Additionally, more detailed information is expected to improve the quality of the ML models.

For the feedback interval, we select the following three intervals for our evaluation: (1) 10 test cases, (2) 50 test cases, and (3) 100 test cases. In industrial blackbox testing scenarios, monitoring the services of a SuT after each test case introduces significant overhead. For example, for each monitoring of the Internet Control Message Protocol (ICMP) service, the fuzzer needs to send an ICMP echo request and needs to wait for the SuT's response before it can

continue. To take this into account and to analyze the impact of this choice, we select the aforementioned feedback intervals. These intervals represent the number of test cases presented to the SuT before the next monitoring cycle runs, and the fuzzer thus receives feedback on the crashed services. Consequently, a larger feedback interval results in less accurate feedback. However, a larger feedback interval also speeds up the fuzzing process, allowing the fuzzer to send more test cases within the fixed timeframe of 24 hours.

RQ3 - *Efficiency*

Similar to *RQ1*, we analyze the overhead introduced by the models by comparing the performance of the model-based fuzzers with the two baseline fuzzers. Specifically, we analyze the number of test cases each fuzzer generates within the fixed 24 hours, while also considering the number of vulnerabilities discovered during that time.

We expect that the model-based fuzzers will generate fewer test cases than the two baseline fuzzers due to the time required for model queries and the online training of the ML models.

7.4.2.3 Fuzzers

For our experiments, we implement three model-based fuzzers and two baseline fuzzers. The following describes the details of these five fuzzers, while Table 7.3 gives an overview. Our implementation of the fuzzers is based on the security testing framework ISuTest® (see Section 2.2.3). Refer to the corresponding Master's thesis and publication for more details on the implementation and the integration into ISuTest® [*Mor21*, *Bor23b*].

Neural Network (A_NN)

Our first concrete instance for the ML model used for `Smevolution` is an NN, drawing inspiration from the work by She et al. [She19]. The authors propose an approach to approximate a program's branching behavior using

Table 7.3: Overview of the fuzzers used for our experiments. All fuzzers use the evolutionary framework introduced by `Smevolution` (see Section 7.4.1). Their implementation is based on ISuTest®.

ID	ML model	Mutation	Selection
A_NN	Neural Network	Mutate in direction of gradient	Predicted number of crashed services
A_DT	Decision Tree	Mutate to change classification	Predicted number of crashed services
A_SVM	SVM	ISuTest® heuristics	Predicted number of crashed services
A_BASE	-	ISuTest® heuristics	Actual number of crashed services
A_RAND	-	ISuTest® heuristics	Random

a smooth function modeled by NNs. Building on this idea, we use an NN to approximate the SuT's crashes. Thus, the NN used by the corresponding fuzzer A_NN represents a continuous function f that maps a test case to the services that are crashed by this test case (see Equation (7.4)). As this function is based on an NN, efficient gradient calculations are possible.

Mutation To mutate a given test case t, we first calculate the gradient of the function the NN represents at position t. Then, we use this gradient to decide in which direction the test case needs to be changed in order to increase the absolute value of $f(t)$. With this, we increase the probability that the mutated test case t' crashes more services. By taking this approach, A_NN follows a depth first search strategy.

Selection The fitness function is calculated based on the predicted number of crashing services $f(t)$. The higher this number, the higher the fitness value.

Decision Tree (A_DT)

The second instance, A_DT, is inspired by the work by Appelt et al. [App18]. The authors use a DT for test case generation within the use case of testing WA firewalls. In their approach, the DT identifies which positions within a test case should be kept and which can be mutated. In the context of testing

WA firewalls, it is important to keep those parts of the test case that help evade the configurations of the firewall. Thus, it is important to identify those parts and to preserve them.

We adapt the approach by Appelt et al. and utilize the DT as follows. As for all models within `Smevolution`, the DT is trained to approximate a function mapping test cases to the SuT's services that crash if we send the test case to the SuT.

Mutation The DT is used to decide on the position of the test case that should be mutated, and how this mutation needs to look like. For this, we classify the currently considered test case t based on the DT. Then, we can trace the decision path through the DT and analyze the nodes along this path. We aim to find a node in which the decision would need to be different in order to classify t to a leaf with more crashed services. If we find such a node, we can change t such that it would be classified differently. This is the mutation that we apply to t. This approach leads to a depth first search approach for `A_DT`.

Selection We define the fitness function based on the number of services that a test case is expected to crash, according to the DT.

Support Vector Machine (`A_SVM`)

We employ a third instance that focuses solely on basing the fitness function on a model's decision, while not influencing the mutation. For this, we choose an SVM, roughly following the approach proposed by Chen et al. [Che19]. The authors present an evolutionary fuzzing framework for cyber-physical systems, which uses sensor values as input. Since the sensor values are continuous values, they can directly be used and manipulated by the ML models. Thus, the approach can not directly be transferred to the discrete values in the use case of network fuzzing, but we adapt the idea of using an SVM for test case selection. The corresponding fuzzer `A_SVM` does not guide the mutation in a specific direction, thus we expect it to perform a broader search than the two approaches presented above.

Mutation As the SVM is not used for the mutation step, we use the heuristic-based mutation strategy provided by ISuTest®.

Selection The fitness of the test cases is determined by the predicted number of crashing services.

Maximum Crashes (A_BASE)

As a baseline for the model-based fuzzers, we implement a fuzzer that is also based on the evolutionary fuzzing approach, but takes its decisions independent of any model of the SuT.

Mutation The mutation step conducted by A_BASE uses the ISuTest® heuristics.

Selection The selection is directly based on the number of services that a test case crashes. The higher the actual number of crashed services, the higher the fitness of the test case.

Random (A_RAND)

To have a general baseline, we additionally implement a fuzzer that performs a random selection, while also using the evolutionary framework.

Mutation A_RAND also uses the heuristics provided by ISuTest®.

Selection The selection is performed at random, meaning that random individuals are selected from the offspring to form the new population.

7.4.2.4 Target

We use an artificial SuT called `VulnDuT` as target for our experiments. `VulnDuT` is an intentionally vulnerable SuT and part of the ISuTest® suite. It is deployed within a Docker container equipped with one network interface for testing, and one network interface for configuration purposes.

The testing interface provides access to various services running within the container, such as an Internet Protocol (IP) stack and a WA. Additionally, the VulnDuT testing interface facilitates communication via the VulnDuT protocol, an artificial network protocol based on UDP. The protocol's network packets include three fields: an unsigned Integer, a signed Integer, and a String. Note that we focus on the stateless VulnDuT protocol, while VulnDuT also supports the so-called *example protocol*, a stateful network protocol that was explicitly designed for evaluating Test Tools (TTs) [*Pfr19c*].

We can define and activate various scenarios for VulnDuT, each specifying which services will crash when particular inputs are sent via the VulnDuT protocol. Each scenario represents one or more deterministic vulnerabilities that can be detected by monitoring the respective services of VulnDuT. For example, a scenario might involve shutting down the WA if a VulnDuT protocol network packet containing the value 64 in the unsigned Integer field is received. The configuration interface of VulnDuT is used to choose the currently active scenario.

Vulnerabilities

To analyze the performance of the fuzzers, we define several vulnerabilities within VulnDuT. The following gives an overview of these vulnerabilities and explains the general goals of defining these vulnerabilities. Refer to the corresponding publication or the Master's thesis by Martin Morawetz for more details on the vulnerabilities [*Bor23b*, *Mor21*].

We implement two types of vulnerabilities to analyze the performance of the fuzzers: *Independent vulnerabilities*, and *linked vulnerabilities*.

The first group, *independent vulnerabilities*, are not connected to each other. Identifying these vulnerabilities is comparable to a breadth-first search. For example, we implement the following independent artificial vulnerabilities, which are triggered if the received VulnDuT protocol packet fulfills specific requirements.

1 A shutdown of the WA is induced if the unsigned Integer has a value of $2^{32} - 1$

2 A shutdown of the ICMP service is induced if the signed Integer has a value of -2^{31}

3 A shutdown of the SNMP service is induced if the String starts with the characters "KL"

In contrast, *linked vulnerabilities* are those that are closely related. Closeness in this case refers to the number of mutations required for a test case to transition from one vulnerability to another. Finding these vulnerabilities is comparable to a depth-first search. We differentiate between three approaches to linked vulnerabilities.

1 Vulnerabilities based on String matching that lead to progressively more crashes. For example, if the String starts with "A", the ICMP service is shut down. If it starts with "AB", the ICMP and the HTTPS service are shut down. For each additionally matching character, more services are shut down.

2 Vulnerabilities based on String matching that lead to only one crash at a time. For example, if the String starts with "K", the ICMP service is shut down. If it starts with "KL", the SNMP service is shut down.

3 Vulnerabilities based on ranges of signed and unsigned Integer values, causing the same set of services to be shut down if the signed or unsigned Integer falls in the specified range. For example, the HTTP and ICMP service are shut down if the signed Integer value is in the interval [1024,2024].

With the independent vulnerabilities, we can analyze how the fuzzers perform with respect to a breadth-first search. In contrast, the linked vulnerabilities test the performance with respect to a depth-first search. In total, we introduce 25 vulnerabilities to `VulnDuT`.

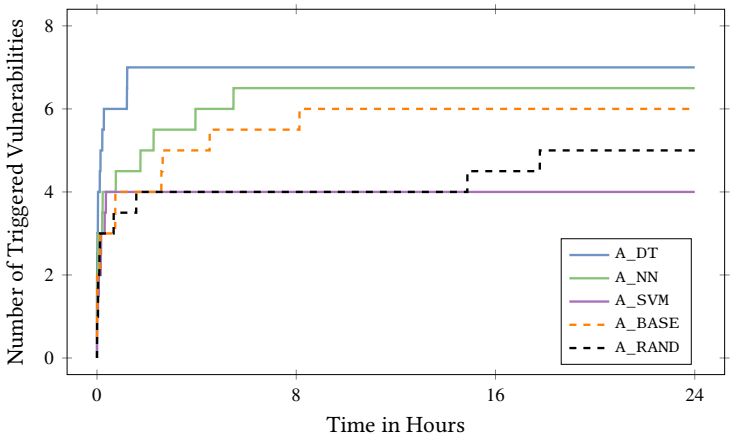


Figure 7.2: Absolute number of triggered vulnerabilities for the different fuzzers over time (mean over 10 runs). A_DT and A_NN outperform the two baseline algorithms A_BASE and A_RAND, as well as A_SVM.

7.4.2.5 Results

This section presents the results of the experiments conducted following the experimental setup outlined in Section 7.4.2.2, while Section 7.4.2.6 discusses the results and their implications with respect to the research questions. All data required to generate the reported figures and tables, as well as additional figures and data, have been published[1].

Triggered Vulnerabilities

First, we report the number of triggered vulnerabilities for the different fuzzers over the course of the fuzzing campaign. We use the default configuration for the feedback dimension (multidimensional feedback) and the feedback interval (50) for these experiments (see also Table 7.2). Figure 7.2 depicts the mean absolute number of unique vulnerabilities triggered by the different fuzzers. Each line represents the mean of 10 runs of the respective fuzzer. Note

[1] https://github.com/anneborcherding/Smarter-Evolution

that we have chosen not to visualize the confidence intervals in Figure 7.2 to maintain clarity. However, figures that include the confidence intervals are available in the published code.

For example, Figure 7.2 shows that A_DT triggered a mean number of 7.0 vulnerabilities at the end of the fuzzing campaign, while A_NN triggered 6.5. The figure demonstrates that A_DT and A_NN outperform A_BASE and A_RAND in terms of the total number of vulnerabilities triggered. Conversely, A_BASE and A_RAND outperform A_SVM in this metric.

Statistical Tests

To analyze the relative performance of the fuzzers in more detail, we conduct statistical tests on the number of triggered vulnerabilities. With this, we follow the established recommendations on fuzzing evaluations [Kle18, Sch24]. We conduct pairwise one-sided Mann-Whitney U tests on the final number of vulnerabilities triggered by the fuzzers, and present the results in Table 7.4. For each two fuzzers F_1 and F_2, this one-sided Mann-Whitney U test uses the alternative hypothesis that the underlying distribution of the number of triggered vulnerabilities of F_2 is stochastically greater than the corresponding distribution with respect to F_1 [Man47]. We choose a significance level of $\alpha = 0.05$, in accordance to literature [Paa21], and highlight each p-value smaller than α in gray for Table 7.4.

Table 7.4: p-values calculated using a one-sided Mann-Whitney U test ($\alpha = 0.05$). Significant differences are highlighted in gray. A_DT outperforms the other fuzzers significantly.

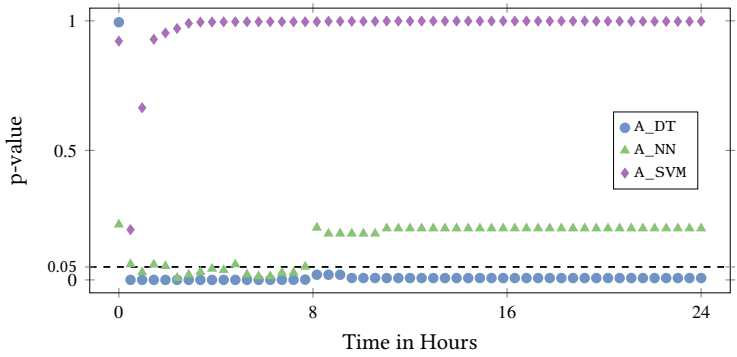
	A_DT	**A_NN**	**A_SVM**	**A_BASE**	**A_RAND**
A_DT	-	0.034	< 0.001	0.007	< 0.001
A_NN	0.972	-	< 0.001	0.199	0.002
A_SVM	> 0.999	> 0.999	-	0.998	0.995
A_BASE	0.995	0.823	0.003	-	0.056
A_RAND	> 0.999	0.998	0.006	0.953	-

Thus, if a cell is colored gray in Table 7.4, it implies that the fuzzer in the respective row triggers significantly more vulnerabilities than the fuzzer in the column. For example, A_DT significantly outperforms A_SVM, A_BASE, and A_RAND with p-values of < 0.001, 0.007, and < 0.001, respectively. The performance of A_NN is only significantly better than the one shown by A_SVM and A_RAND (< 0.001 and 0.002). A_SVM is significantly outperformed by all other fuzzers.

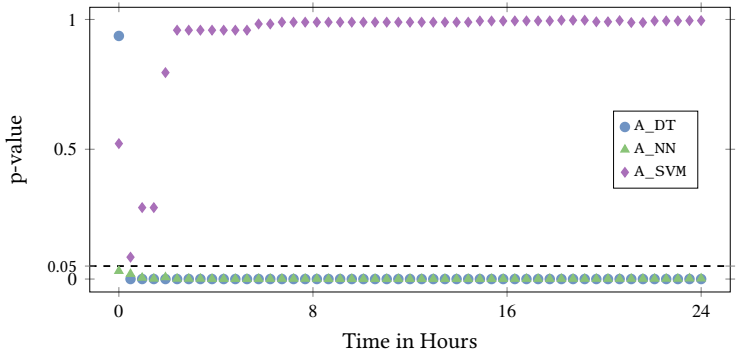
Note that we have chosen to report the values of a *two-sided* Mann-Whitney U test in the publication of `Smevolution` [*Bor23b*] and thus report different p-values there. While leading to the same conclusions, the results from a one-sided test are better suited to intuitively represent the relevant information, since we are not generally interested in significant differences, but are interested in whether a fuzzer leads to significantly *better* results.

Exceeding the requirements by best practices for fuzzing evaluations [Kle18, Sch24], we also calculate and report the p-values over time. This allows to analyze how the relative fuzzer performance changes during the fuzzing campaign. Again, we calculate a pairwise one-sided Mann-Whitney U test. Figure 7.3 shows the p-values over time for the three fuzzers A_DT, A_NN, and A_SVM when compared to A_BASE (Figure 7.3a) and A_RAND (Figure 7.3b). Data points below the horizontal line, which represents the significance level, correspond to those points in time at which the corresponding fuzzer outperformed the respective baseline fuzzer significantly.

Figure 7.3a shows that A_NN outperforms A_BASE significantly at the beginning of the campaign, while after around eight hours, the difference is no longer significant. A_DT outperforms A_BASE significantly over the whole fuzzing campaign. In Figure 7.3b, the data points concerning A_DT and A_NN are below the significance level over the whole fuzzing campaign, implying significant better performance over the whole campaign.



(a) Comparison to A_BASE. A_DT outperforms A_BASE significantly over the whole fuzzing campaign, while A_NN only outperforms it significantly in some time intervals.



(b) Comparison to A_RAND. A_DT and A_NN outperform A_RAND significantly over the whole fuzzing campaign.

Figure 7.3: p-values calculated using a one-sided Mann-Whitney U test over the time of the fuzzing campaign. The horizontal line represents the significance level $\alpha = 0.05$.

Overhead

To quantify the overhead introduced by the ML models, we measure the throughput of the fuzzers in terms of the number of test cases sent to the SuT within 24 hours. Figure 7.4 presents the mean and standard deviation of the

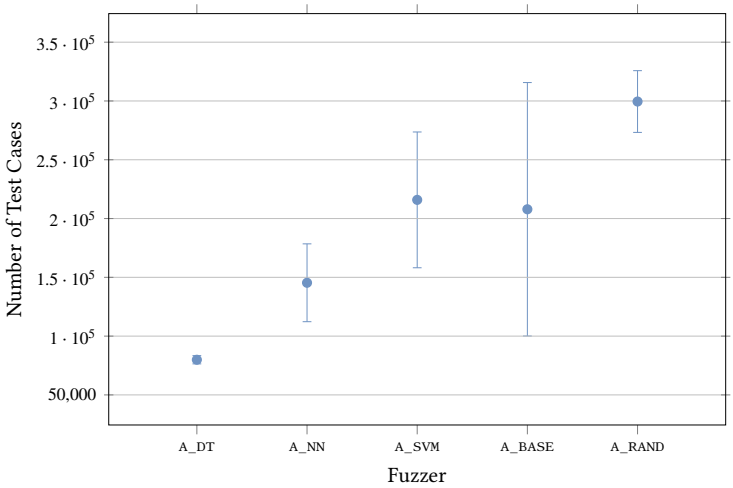


Figure 7.4: Mean number of test cases the fuzzers send over 24 hours. The whiskers represent the standard deviation.

number of test cases sent by the fuzzers, based on the data from 10 runs in the default configuration (see Table 7.2). For example, A_SVM sends 215,884 test cases on average, with a standard deviation of 57,746.

The results demonstrate that A_RAND achieves the highest throughput by generating 299,524 test cases in 24 hours. This is followed by A_SVM with 215,884 test cases, and A_BASE with 207,882 test cases, both of which do not introduce changes to the mutation step of the EA.

In contrast, A_DT and A_NN send fewer test cases, with 79,900 and 145,372, respectively. This corresponds to a reduction in the number of test cases sent compared to A_RAND by 27.92% for A_SVM, 51.47% for A_NN, and 73.32% for A_DT. Notably, A_DT exhibits the smallest variation in the number of test cases sent.

Information Granularity

We run each of the configurations presented in Table 7.2 for 10 times and analyze whether the choice for the feedback dimension and the feedback interval influence the performance of the fuzzers in terms of the number of triggered vulnerabilities. Our analyses show that neither the choice for the feedback dimension nor for the feedback interval influences the performance of the model-based fuzzers significantly (see [*Bor23b*] for details). However, the performance of A_BASE is influenced significantly by both the feedback dimension and the feedback interval. It shows that A_BASE triggers significantly fewer vulnerabilities with unidimensional feedback when compared to multidimensional feedback. Moreover, A_BASE triggers significantly more vulnerabilities with a feedback interval of 100 when compared to a feedback interval of 10.

7.4.2.6 Discussion of Results

The following presents the key conclusions drawn from the previously presented results with respect to the research questions formulated in Section 4.3.1. For a more detailed analysis and discussion of the specific vulnerabilities found by the different fuzzers as well as an analysis of the distributions of test cases generated by the fuzzers, refer to the corresponding publication [*Bor23b*].

RQ1 - *Model Impact*

Our experiments show that A_DT, which uses a DT for mutation and selection, significantly outperforms the two baseline fuzzers A_BASE and A_RAND. Moreover, A_NN, using an NN for mutation and selection, significantly outperforms A_RAND. We conclude that both models are able to learn and represent the behavior of the SuT in a way that is suitable to improve blackbox fuzzing.

Our experiments also show that A_SVM, which uses an SVM for selection, performs significantly worse than the two baseline fuzzers. Since it uses the same mutation strategy than the baseline fuzzers, we conclude that the SVM is not able to represent information that improves the selection strategy. Based

on this observation, we conclude that using a model for selection has the potential to decrease the performance in terms of triggered vulnerabilities. In future work, one could combine an NN- or DT-based mutation strategy with the baseline selection strategy implemented with A_BASE, to analyze the impact of the models on the mutation only.

RQ2 - *Information Granularity*

In contrast to our hypotheses (see Section 4.3.1), the choice of feedback dimension and feedback interval does not significantly influence the performance of the fuzzers in terms of triggered vulnerabilities. In our experiments, we observe that (1) the model-based fuzzers perform significantly better than the baseline fuzzers, and (2) that the feedback dimension does influence the performance of A_BASE, with multidimensional feedback leading to a significantly higher number of triggered vulnerabilities, supporting our corresponding hypothesis. Based on these observations, one could come to the conclusion that the model-based fuzzers can compensate for the reduced information that is introduced by the unidimensional feedback. This aligns with results from previous work which also successfully used binary feedback to improve blackbox testing [App18]. Nevertheless, additional experiments could be conducted to strengthen this conclusion.

Moreover, our experiments show that the feedback interval does not significantly impact the performance of the model-based fuzzers. While a smaller feedback interval offers higher resolution feedback, it also introduces greater overhead due to the additional service monitoring cycles required. For model-based fuzzers, these advantages and disadvantages appear to balance out, resulting in similar performance.

However, the feedback interval does affect the performance of A_BASE. Specifically, the a feedback interval of 500 leads to significantly more vulnerabilities than the a feedback interval of 10. This shows that for A_BASE, the reduced monitoring overhead for a feedback interval of 500 is more relevant than the improved feedback granularity facilitated by a feedback interval of 10. One possible explanation is that test cases within the same interval are likely to be

similar, as they have been generated from related test cases. Consequently, the information loss from larger feedback intervals might be minimal and thus can be compensated for.

RQ3 - *Efficiency*

`Smevolution` does not require an a priori training phase for the ML models as these models are trained and queried during the fuzzing process itself. To evaluate the performance overhead introduced by the models during fuzzing, we measure the number of test cases the fuzzers send to the SuT. Our results indicate that A_RAND generates the highest number of test cases which is expected since it does not utilize any model.

A_BASE, A_NN, and A_SVM produce a comparable number of test cases, while A_DT generates the fewest. Despite this, A_DT significantly outperforms A_RAND and A_BASE in terms of the number of triggered vulnerabilities within the 24 hours time frame. Additionally, A_DT identifies vulnerabilities earlier in the fuzzing campaign (see Figure 7.2). From these findings, we conclude that although A_DT produces the fewest test cases, it generates the most effective ones.

7.4.3 Related Work

EAs have been used for graybox fuzzing at least since 2007 [DeM07, Man19], while their application to blackbox fuzzing is not as established. In the following, we focus on evolutionary fuzzing in a blackbox setting, while acknowledging that there are several works using EAs for graybox fuzzing, including AFL [Zal16], AFL++ [Fio20], AFLNet [Pha20], and `libfuzzer` [LLV24].

Evolutionary Blackbox Fuzzing Appelt et al. present an approach for fuzzing WA firewalls. The authors use a DT to generate SQL injection test cases that need to first bypass the firewall [App18]. Specifically, the DT identifies the parts of a test case responsible for bypassing the firewall, ensuring these parts remain unchanged to retain their bypass functionality. To train the DT, the

authors use binary feedback, specifically the binary information on whether the test case successfully bypassed the firewall. We adapt the idea by Appelt et al. for our implementation of the DT-based fuzzer `A_DT`.

Duchene et al. present `KameleonFuzz`, an approach to WA fuzzing employing an evolutionary approach [Duc14] . `KameleonFuzz` uses the response DOM and taint inference to assess the fitness of a test case. As a basis for their fuzzing, the authors automatically infer a model of the control flow graph of the WA.

Chen et al. introduce an evolutionary fuzzing framework for cyber-physical systems that leverages ML models [Che19]. Their framework uses sensor values from the tested cyber-physical system as input for the ML models. Unlike network fuzzing, sensors in cyber-physical systems provide continuous data, which can be directly used by the ML models to select and mutate test cases. Our implementation of `A_SVM` is inspired by the work by Chen et al.

Shang et al. propose to perform mutation and selection in an EA by mutating the current test case in the direction of a test case that was successful in triggering an anomaly in the SuT in the past [Sha21]. The authors call this successful test case *suspicious point*. Moreover, the authors consider the average similarity of test cases in the population for their mutation strategy. The authors evaluate their approach based on the dispersion of the test case population. While this approach has similar objectives as `Smevolution`, it is not clear how the newly presented method affects vulnerability finding performance or code coverage.

Feedback Granularity Our experiments show that our fuzzers perform statistically indistinguishable for unidimensional and for multidimensional feedback. The approach presented by Appelt et al. also only use binary feedback [App18]. Appelt et al. use the binary information whether a test case was able to bypass the tested WA firewall (see above). The authors show that their approach successfully utilizes the binary feedback to test the corresponding SuT.

7.4.4 Discussion

The following discusses Smevolution and the results of our experiments, with a special focus on their implications, limitations, and on possible future research directions.

7.4.4.1 Implications

Blackbox testing generally uses less sophisticated approaches since the blackbox test setting lacks the necessary information to apply, for example, graybox testing approaches [Man19]. Improving upon this, Smevolution leverages ML models trained on the information on sent test cases and the corresponding behavior of the SuT to allow for a targeted mutation and selection in an evolutionary fuzzing approach. With this, it can be seen as a starting point for the application of evolutionary graybox testing approaches to blackbox testing and thus allowing for more sophisticated testing approaches.

To support further research based on our results, we published the raw data from our experiments[1], accompanied by the evaluation scripts used to generate the figures and tables in this section. Moreover, the code includes additional figures and data to provide a deeper insight into our results. We also published the code we used to define the ML models used in this work to allow for a transparent evaluation of our work.

Even though our experiments focus on the artificial SuT VulnDuT, our integration of Smevolution into ISuTest® allows to apply Smevolution to test OT components such as BC_{ex} (see Section 7.7). Since the approach of Smevolution is independent of the underlying network protocol, it can be applied to all network protocols supported by ISuTest®. Moreover, the general approach of Smevolution could also be extended to other embedded systems, which are required to be tested in a blackbox setting and show the behavior of services crashing during testing.

[1] https://github.com/anneborcherding/Smarter-Evolution

7.4.4.2 Limitations

The presented experiments give insights into the performance of evolutionary blackbox fuzzers using ML models for mutation and selection while using an artificial SuT as target. While this helps to exactly control and analyze the existing vulnerabilities, it potentially restricts the transferability of results to real-world applications of the fuzzers. To reduce this limitation, we implemented the vulnerabilities based on vulnerabilities that are known to exist in real-world OT components (see e.g. Section 5.4).

Moreover, due to resource restrictions, it was not possible to completely fulfill the requirements for fuzzing evaluations as formulated by Klees et al. [Kle18] and confirmed by Schloegel et al. [Sch24], most notably with respect to the number of trials each fuzzer was run. Even though we account for the randomness of the fuzzers and the training of the ML models by running each configuration 10 times, our results would be more statistically robust if we repeated each configuration 30 times.

7.4.4.3 Future Work

Future work could expand upon the experiments presented in this section. First, one could evaluate `Smevolution` using OT components as targets, thus evaluating the real-world applicability of `Smevolution` as well as the impact of the feedback granularity in real-world scenarios. For OT components, test cases are usually longer and vulnerabilities fewer, and thus the granularity of the feedback might have a more substantial impact. An evaluation with OT components would also enable us to assess the transferability of results from experiments using `VulnDuT` in general by comparing the results from this study to those generated by future studies. To gain more insights into the absolute performance of the fuzzers implemented within `Smevolution`, one could compare their performance with state-of-the-art network fuzzers, either blackbox or graybox.

Additionally, one could analyze the impact on performance and efficiency caused by changing parts of the general approach of `Smevolution`. On the one hand, it would be possible to only execute the test cases in the population, and not those from the bigger offspring. This would reduce the number of test cases sent and thus save time during testing, but it would also provide less information to the fuzzer. On the other hand, one could use the ML model only for mutating the input, and keep the baseline strategy for the selection step. Our experiments suggest that this might improve the overall performance of the fuzzers (see Section 7.4.2.6).

Furthermore, our experiments show that different fuzzers excel in identifying independent and linked vulnerabilities (see Section 7.4.2.4 and [*Bor23b*]). A possible approach for future work would be to combine multiple ML models to balance breadth-first and depth-first search approaches.

Another research direction could involve leveraging the explainability [Dwi23] of the NN used by `A_NN` to better understand and to improve its decision-making process.

7.4.5 Summary

We present `Smevolution`, an approach designed to bridge the gap between graybox and blackbox testing. During the fuzzing process, we train an ML model that approximates the behavior of the SuT. This model is integrated into an EA and is used to (1) select promising test cases for the next evolutionary round, and to (2) guide the mutations of the test cases. With this, we provide information to the EA that is typically unavailable in blackbox testing.

Our experiments, conducted using the artificial SuT `VulnDuT`, yield the following key insights.

1 The fuzzer using a DT significantly outperforms a baseline fuzzer which uses the EA without any model-based support. This shows that the application of the ML model indeed improves the fuzzing performance.

2 The feedback dimension and the feedback interval have no significant impact on the performance of the model-based fuzzers, but do influence the performance of the baseline fuzzer.

Our approach and our experiments provide a foundation for applying efficient graybox testing approaches to blackbox test settings. This is particularly relevant in the domain of OT components, where efficient blackbox tests are essential due to standards requirements and systems including blackbox third-party components.

7.5 Hidden Markov Models

This section presents `Palpebratum`, our approach approximating the behavior of the SuT using HMMs. As we did with `Smevolution` presented in the previous section, we again aim to leverage an ML model which represents the behavior of the SuT. With `Smevolution`, we utilized a DT, a NN, and a SVM to represent the behavior of the SuT with respect to the number of services that crash for a given test case. In contrast, with `Palpebratum`, we utilize an HMM to represent the inner workings of the SuT. For this, we interpret the network traffic that the SuT produces as emissions produced by an HMM. Thus, we collect the network traffic resulting from sending one or more test cases to the SuT, and use the Viterbi algorithm to calculate the state sequence of the HMM that has the highest probability to having generated this observation. With this, we receive information on which states where traversed by the SuT while handling the test case. We use this information similar to the code coverage information available in graybox fuzzing. If a test case takes a transition that has not been taken before, this test case is considered to be interesting and is added to the corpus. With this, we allow for an application of graybox testing approaches in a blackbox scenario.

Our evaluation of Palpebratum is twofold. First, we evaluate the performance of the HMMs with respect to their ability to approximate the interestingness of test cases in comparison to the graybox baseline AFLnwe. To this end, we propose two novel scores which quantify the HMMs' performance. Our experiments demonstrate that HMMs with more nodes tend to achieve higher scores.

Second, we implement a blackbox fuzzer which utilizes an HMM to guide the test case generation as well as two baseline fuzzers. We choose two HMMs to be used for this evaluation, based on the scores calculated in the first part of our evaluation. Subsequently, we execute these two instances of Palpebratum and two baseline fuzzers against ProFTP. Our results indicate that the HMM-based fuzzers are able to generate test cases that are more efficient than those generated by the baseline fuzzers. However, the final coverage achieved by the baseline fuzzers is significantly higher than the one achieved by Palpebratum. Explanations for this observation is the possible underestimation of the coverage achieved by Palpebratum, and differences in efficiency of the fuzzers caused by the overhead the HMMs introduce.

7.5.1 Approach

As described above, Palpebratum encompasses an HMM representing the inner workings of a SuT as it can be observed by the network traffic. This HMM is then utilized to approximate the interestingness of a test case, which is then used to guide the fuzzing process in a graybox manner while still keeping a blackbox test scenario. For this, we first train an HMM on network traffic generated by user interaction with the SuT. This network traffic is first preprocessed by the means provided by NeDaP as presented in Section 6.4. Then, this HMM is included in a fuzzer, which utilizes the HMM to decide on the interestingness of a test case. In the following, we give an overview of the fuzzing process with Palpebratum, while Section 7.5.1.1 gives information on the data representation for the HMM, including the parameter choices of NeDaP. Section 7.5.1.2 provides more details on the characteristics and the training of the HMM, and on the integration of this approach into the fuzzing library LibAFL [Fio22].

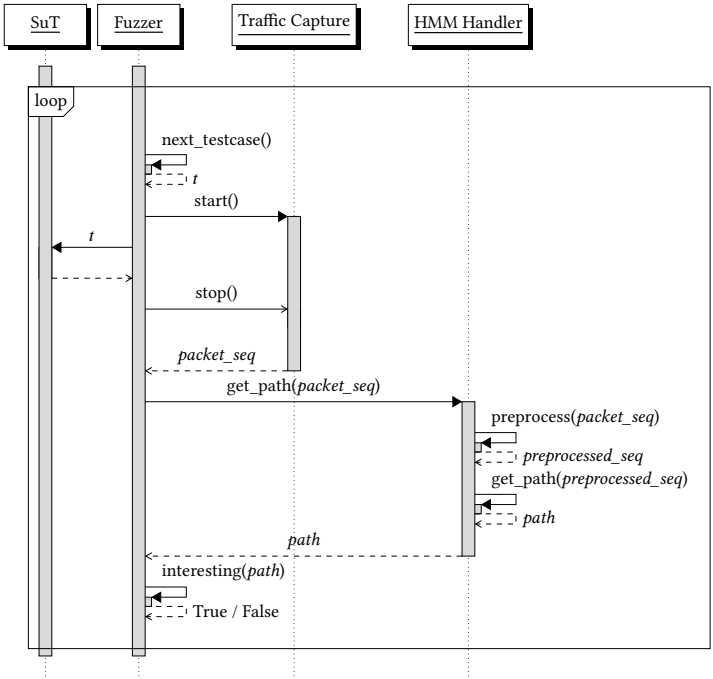


Figure 7.5: Specialities of the `Palpebratum` fuzzing loop, building upon the general approach of coverage-guided fuzzing. During the execution of the test case, the network traffic is captured. This capture is then used to calculate the state path of the HMM that has the highest probability of producing this network traffic. This path information is used to decide whether the test case is interesting for the fuzzer or not, similarly to the coverage information in graybox coverage-guided fuzzing.

Figure 7.5 shows the fuzzing process as conducted with `Palpebratum`, while omitting the details of the general coverage-guided fuzzing process and focussing on the specifics of `Palpebratum`. See Section 2.4 for more details on the general approach of coverage-guided fuzzing. The fuzzing loop as shown in Figure 7.5 starts with the fuzzer getting the next test case that should be sent to the SuT. Note that this might include several steps such as selecting the seed and mutating the seed to generate the next test case t. Then, the fuzzer starts the network traffic capture, sends the test case t via the network to the SuT, waits for the SuT's response, and stops the network traffic capture again.

The recorded network packet sequence is then given to the HMM handler. This handler first preprocesses the network packets using NeDaP, and then calculates the state sequence of the HMM that has the highest probability to lead to this network packet sequence using the Viterbi algorithm. This state sequence, or path, is then given to the fuzzer. The fuzzer treats this path information similarly to the coverage map information collected in coverage-guided graybox fuzzing. It keeps track of the states of the HMM that have already been visited, and if the path taken for a test case t includes a state that was not visited before, t is deemed to be interesting. Then, the fuzzer uses this information to decide whether a test case should be added to the seed corpus. This concludes one fuzzing loop using Palpebratum.

7.5.1.1 Data Representation and Preprocessing

We split the network traffic into sequences of packets, each sequence including the packets from one TCP connection. Each sequence is then interpreted as a sequence of observations emitted by the HMM.

To make this network data usable by an HMM, it needs to be preprocessed first. For this, we use the approach NeDaP as presented in Chapter 6. The application of NeDaP within Palpebratum is shown in Figure 7.6. We split the network traffic into sequences using the TCP connections as split criterion. Then, we feed this sequence into NeDaP and receive a sequence of preprocessed network packets. Each of these packets is represented by a byte vector of length $d_o = 24$.

Then, we feed the preprocessed packets and a pre-trained HMM into the Viterbi algorithm to calculate the state sequence withing the HMM that has the highest probability to lead to the given sequence of observations. This sequence of states of the HMM is then further used by the fuzzing as an indicator of interestingness (see also Figure 7.5).

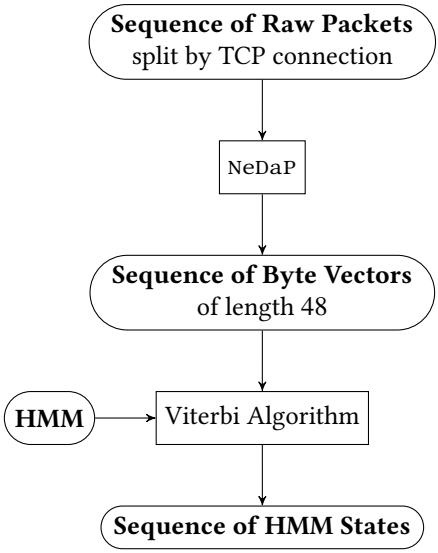


Figure 7.6: Data preprocessing and application within `Palpebratum` using `NeDaP` as presented in Section 6.4. The raw network data is preprocessed by `NeDaP` and then used to determine the most likely state sequence within the HMM.

For `NeDaP`, we choose the parameters based on the results of our experiments as presented in Section 6.4.3. These choices are also shown in Table 7.5. We acknowledge that our choice of parameters for `NeDaP` probably influences the performance of the downstream HMM building upon the preprocessed network packets.

Input Dimension

We choose the input dimension for the dimensionality reduction approaches to be $d_i = 304$, with the main goal of being consistent with the experiments shown in Section 6.4.3. The main reason to choose $d_i = 304$ instead of $d_i = 1504$ there was that the training time of the dimensionality reduction approaches was heavily reduced, while only slightly impacting the reconstruction error. We acknowledge that the training time is not as relevant for the concrete

Table 7.5: Parameter values chosen for NeDaP as used within Palpebratum.

Parameter	Value	Reason
d_i	304	Consistency with the experiments in Section 6.4.3
d_o	24	Balance between number of parameters in HMM and the reconstruction error (see Figures 6.9b and 6.10)
general seed	37	Consistency with the experiments in Section 6.4.3
specific seeds	PCA: 437883 AE: 721885 CAPC: 437883	Seed leading to the smallest reconstruction error when validating the dimensionality reduction approaches on the ProFTP fuzzing dataset (see Figure 6.10a)

use case presented in this chapter as each dimensionality reduction approach only needs to be trained once. However, to allow for a direct comparison and transferability of the results presented in Section 6.4.3, we choose $d_i = 304$.

Output Dimension

For the output dimension of the dimensionality reduction approaches of NeDaP, we choose $d_o = 24$. This values strikes a balance between the reconstruction error of the dimensionality reduction approaches and the number of trainable parameters in the resulting HMM. As we use the dimensionality reduction approaches trained on user data on fuzzing data, we base our decision on the corresponding experiments conduced in Section 6.4.3, shown in Figure 6.10. For ProFTP, $d_o = 24$ only leads to a slightly higher reconstruction error for AE and CAPC, while also reducing the variance. For example, the median reconstruction error of CAPC for $d_o = 48$ is 0.0257 with a lower quartile of 0.0227 and an upper quartile of 0.0274. For $d_o = 24$, the median is 0.0262 with a lower quartile of 0.241 and an upper quartile of 0.0272. Similar behavior can be seen for LightFTP.

However, increasing the dimension of the emissions from 24 to 48 leads to an increased number of trainable parameters for the emission distribution of the HMM. We call this number of trainable parameters $\boldsymbol{P}$. Since we are using a multivariate Gaussian distribution (see below), the choice of the emission

dimension influences size of the vector of means and the covariance matrix. For each node in the HMM, we need one vector of means of the size D and a covariance matrix of size $D \cdot D$. Since the covariance matrix is symmetric, the covariance has $D + \frac{D(D-1)}{2}$ trainable parameters. Increasing the emission dimension from 24 to 48, we would increase P by 3.78 (see Equation (7.5)). This is why we decide to use $d_o = 24$ for `Palpebratum`.

$$\frac{P_{48}}{P_{24}} = \frac{48 + 48 + 48 \cdot 47 \cdot 0.5}{24 + 24 + 24 \cdot 23 \cdot 0.5} = \frac{1224}{324} = 3.78 \tag{7.5}$$

Seeds

For `Palpebratum`, we choose the seeds for the randomness used during the preprocessing of network traffic based on our experiments with `NeDaP`. We choose the general seed to be 37, keeping it the same as for the experiments with `NeDaP` to be able to produce comparable results. For the three dimensionality reduction approaches, we choose the seeds that lead to the smalled reconstruction error for the approaches trained on the ProFTP user data dataset and validated on the ProFTP fuzzing dataset (see Section 6.4.1.2). Table 7.5 shows the concrete numbers of these seeds.

7.5.1.2 Hidden Markov Model

The preprocessed network packets are fed into an HMM. In the following paragraphs, we first motivate selecting HMMs as the foundation for `Palpebratum`, and then provide a description of the specific characteristics of the HMMs used.

Choice of Model

In our setting, we aim to model sequential, multivariate data of which the underlying state is not observable. For this, several modeling approaches could be suitable, such as HMMs, Recurrent Neural Networks (RNNs) and LSTMs, all of which are able to model sequential multivariate data. For our specific

use case, we aim to use the learned state representation to approximate the internal behavior of the SuT. While RNNs and LSTMs would implicitly model the internal states, an HMM explicitly models the states. Moreover, using the Viterbi algorithm, we can calculate the state sequence that has the highest probability to have caused the observation sequence. With this, the internal behavior of the SuT can be approximated easier.

Furthermore, research from other domains suggests that HMMs are better suited than LSTMs in cases in which only little training data is available [Tad20, Pan16].

While we acknowledge that other modeling approaches such as RNNs or LSTMs might be suited better to accurately model the network packet sequences given enough training data, an HMM makes the utilization of this model for guided fuzzing easier. As the goal of `Palpebratum` is to improve blackbox fuzzing by leveraging the information on the SuT as learned and represented by a model, we choose to use an HMM as a basis for `Palpebratum`.

One disadvantage of HMMs in comparison to RNNs and LSTMs is that long range dependencies can not be modeled, since the underlying states are modeled as Markov chain (see e.g. [Kha21]). We mitigate this shortcoming by including HMMs based on second order Markov chain in our experiments, which is also explained in more detail in the next paragraph.

Characteristics

As the observations emitted by the HMM are byte vectors, we need to use a multivariate HMM for `Palpebratum`. The emissions of multivariate HMMs are multidimensional, as opposed to univariate HMMs, which only emit scalar values [Liu10].

As second-order Markov chains can be represented by first-order Markov chains (see Section 2.6.5), we use HMMs using first-order Markov chains with a quadratic number of states to represent second-order Markov chains for our experiments.

Note that the HMM that we train and utilize during fuzzing is not intended to comprehensively represent the behavior of the SuT, but rather is expected to represent the information given by the network traffic in a way such that it can be used by a fuzzer. For a comprehensive model of the SuT's behavior richer data would be needed [Xav22].

Training

We train the HMM on the user-generated network traffic which is first pre-processed using `NeDaP`. For this, we apply the Baum-Welch algorithm which is used to fit HMMs to data (see Section 2.6.5). In our current approach, the HMM is not further trained during the fuzzing process, which could be investigated in future work.

7.5.2 Research Questions and Methodology

Our experiments with `Palpebratum` are driven by the following research questions.

RQ1 How can we assess the performance of the HMMs `Palpebratum` is based on?

The first research question is concerned with the assessment of the HMMs's performance with respect to the approximation of the SuT's behavior. We focus on comparing an HMM's behavior approximation with a graybox code coverage, while other baselines could also be used. We propose two different means to assess this performance: by (1) defining a similarity measure for coverage curves, and by (2) analyzing which test cases are deemed interesting by the HMMs and the baseline (see Section 7.5.3).

RQ2 How does the behavior approximation of the HMMs compare to the behavior approximation done by `AFLnwe`?

Based on the performance measures defined within *RQ1*, we assess the performance of the HMMs in comparison to the behavior approximation based on code coverage as calculated by AFLnwe. For this, we first train HMMs on user-generated data for File Transfer Protocol (FTP) implementations. Then, we run AFLnwe for eight hours against these implementations, and record the test cases that AFLnwe deems interesting as well as the network traffic that is generated during this run. We then use this recorded network traffic sequences to query the HMMs for the most likely hidden state sequence that lead to this sequence. This is used as a measure for the interestingness of a test case as described in Section 7.5.1. Subsequently, we can compare which test cases were deemed to be interesting by AFLnwe and by the HMM. With this information, we can decide how the behavior approximation as expressed in the interestingness of the test cases differs between the blackbox HMMs and the graybox fuzzer AFLnwe.

RQ3 Does the HMM-based behavior approximation improve blackbox fuzzing in terms of code coverage?

We integrate the HMMs into the fuzzing library LibAFL (see Section 7.5.2.1). The predicted state sequence of the HMM is used as a basis for the interestingness assessment for the test cases, and thus the HMM is used to guide the blackbox fuzzing process.

For our experiments, we choose the HMMs with the best performance, as calculated based on the scores presented in *RQ2*, and implement fuzzers using these HMMs. As reference fuzzers, a blackbox fuzzer which has no additional guidance, and a fuzzer receiving random feedback (see Section 7.5.2.1 for more information on these baseline fuzzers). We run each fuzzer for 24 hours against the FTP server ProFTP (see Section 7.5.2.3), and repeat each configuration 30 times to account for the fuzzers' randomness. With this, we can assess the relative performance of the HMM-based fuzzer `Palpebratum` in comparison to an unguided blackbox fuzzer as well as a random fuzzer.

RQ4 Which impact does the dimensionality reduction approaches used to preprocess the network traffic have on the HMMs' and the fuzzers' performance?

Table 7.6: Fuzzers used during our experiments. The newly presented HMM-based blackbox fuzzer `Palpebratum` is evaluated against three reference fuzzers.

Fuzzer	Approach	Feedback	Mutation
`Palpebratum`	Blackbox	HMM predicted state sequence	LibAFL default
RAND	Blackbox	Random Feedback	LibAFL default
BLACKBOX	Blackbox	Crash Feedback	LibAFL default
AFLnwe	Graybox	Coverage Feedback	AFLnwe default

In Section 6.4, we proposed and evaluated three different dimensionality reduction approaches to be used for the network packet preprocessing approach NeDaP. The HMMs as utilized by `Palpebratum` use this preprocessing and thus represent a downstream application for NeDaP. We analyze how the different dimensionality reduction approaches influence the performance of the HMMs with respect to the behavior approximation as discussed for RQ2 and with respect to performance of a fuzzer utilizing this specific HMM.

7.5.2.1 Fuzzers

For our experiments, we use the HMM-based fuzzer `Palpebratum`, as well as three reference fuzzers to allow for a relative assessment of `Palpebratum`'s performance. Table 7.6 gives an overview of the used fuzzers.

`Palpebratum` `Palpebratum` utilizes an HMM to assess the interestingness of a test case as described in Section 7.5.1. While still being a blackbox fuzzer, `Palpebratum` thus is able to guide the fuzzing as it is used in graybox fuzzing.

RAND This reference fuzzer receives random feedback on whether a given test case is interesting or not. Each test case has a probability of 0.5 to be considered interesting. With this, we can use it as a baseline to decide whether the HMM within `Palpebratum` performs better than random.

BLACKBOX This reference fuzzer receives information on crashes of the SuT and thus receives the same amount of information as a usual blackbox fuzzer does. With this, we can analyze whether `Palpebratum` performs better than a usual blackbox fuzzer.

AFLnwe We also compare the performance of the blackbox fuzzers to the stateless graybox network protocol fuzzer AFLnwe[1]. It observes the code coverage a test case achieved and uses this information for its test case generation.

Implementation

We implement Palpebratum, RAND, and BLACKBOX based on the fuzzing library LibAFL v0.11.1, which is written in Rust [Fio22]. To this end, we implement several new modules in the framework given by LibAFL such as a module to allow for fuzzing via TCP, and a module to allow to use an HMM to observe the behavior of the SuT. To capture the network traffic during fuzzing, we use tcpdump[2].

We implement the HMM in Python using the library pomegranate[3] v0.14.3 as there was no sufficient HMM implementation available in Rust at the time of our work on Palpebratum. During the work on Palpebratum, pomegranate was rewritten from using Cython to using PyTorch aiming to provide a faster implementation with additional features. However, the new implementation aggravates an issue of the old implementation that occurs if HMMs are trained on only a few observation sequences[4]. Based on this observation, we decided to stick to the older version of pomegranate for our experiments.

Implementing the HMM in Python requires to connect it to the fuzzing part of Palpebratum which is written in Rust. We implement this link using PyO3[5].

7.5.2.2 HMMs

For our experiments, we need to determine the number of nodes the HMMs should include, and the data the HMMs should be trained on.

[1] https://github.com/thuanpv/aflnwe
[2] https://www.tcpdump.org/
[3] https://github.com/jmschrei/pomegranate/tree/master
[4] https://github.com/jmschrei/pomegranate/issues/633
[5] https://github.com/PyO3/pyo3

Numbers of Nodes

To this end, we leverage information from literature and studies conducted during the course of this doctoral work. Gascon et al. train a Markov chain to approximate the behavior of FTP implementations, while choosing a different approach to train and utilize the resulting models. The authors use a second order Markov chain and their experiments result in a model with 6 nodes [Gas15, Figure 3]. Note that our models as well as the model shown by Gascon et al. include an additional start and end node, which is added to the number of inner nodes in the following. As we choose to represent the second order Markov chain as a first order Markov chain for our HMMs, we include a model with $6 \cdot 6 + 2 = 38$ nodes in our experiments. Moreover, to analyze the impact of adding or deleting a node, we include models with $5 \cdot 5 + 2 = 27$ and $7 \cdot 7 + 2 = 51$ nodes.

The Master's thesis by Robert Mumper, which was supervised during this doctoral work, analyzes the number of nodes necessary to represent the user-generated network traffic for different FTP server implementations [*Mum21*]. While depending on the sever, the experiments show that models with 16 to 21 nodes are suited to represent a dataset based on LightFTP data, while models with 7 to 8 nodes are suited to represent ProFTP data. Thus, we decide to include models with $4 \cdot 4 + 2 = 18$ and with 7 nodes.

In summary, we decide to use models with the following numbers of nodes for our experiments: 7, 18, 27, 38, 51.

Training Data

We choose to use FTP data for training, as FTP is a text-based protocol frequently used in OT components for which user-generated data can be acquired by using established tools such as Filezilla[1]. Moreover, the fuzzing evaluation framework `ProFuzzBench` supports several FTP implementations as target [Nat21].

[1] https://filezilla-project.org/

We capture user-generated traffic for each of the FTP servers supported by ProFuzzBench, following a defined script of user actions [*Här23*]. As the data captured for one FTP server does not suffice to successfully train a HMM with more than 7 nodes, we combine the user data captured for the following FTP servers: BFTP, LightFTP, ProFTP, PureFTP. This results in a dataset consisting of 137 sequences of FTP traffic.

7.5.2.3 Target

We use the FTP implementation ProFTP as target for our experiments. In contrast to the other FTP implementations supported by `ProFuzzBench`, ProFTP parses FTP with several commands, and processes each of these commands separately. The other FTP implementations only process the first command. As stateless fuzzers such as `AFLnwe` send single packets with several commands in as test cases, these test cases are usually not fully processed by the targets, and the targets only reply with a single response. Thus, the resulting network traffic is not as elaborated as for ProFTP.

Based on this observation we choose to use ProFTP as target for our experiments with respect to `NeDaP` (see Section 6.4.3), and, for consistency, also use ProFTP for our experiments with respect to `Palpebratum`. Nevertheless, the fuzzers implemented in this section, namely RAND, BLACKBOX, $\texttt{Palpebratum}_{\text{AE}}$, and $\texttt{Palpebratum}_{\text{CAPC}}$ send each command in a single packet and thus can also be applied to other FTP implementations.

7.5.3 HMM Performance Assessment

The first two research questions outlined in Section 7.5.2 focus on the performance of the HMMs in approximating the SuT's behavior and determining the interestingness of a test case. To address these questions, we introduce two scores to measure the HMMs' performance against a graybox baseline (Section 7.5.3.1). Then, we evaluate the HMMs based on these scores (Section 7.5.3.2).

7.5.3.1 Similarity Scores

We propose two scores to assess the performance of the HMMs' behavior approximation in comparison to a baseline. For our experiments, the baseline is the graybox behavior approximation and interestingness assessment conducted by AFLnwe, which uses code coverage.

Interestingness Similarity Score

On the one hand, we measure performance by analyzing which test cases are deemed interesting by both the baseline and by the HMM. While this method abstracts from the actual approximation of the SuT's, it directly evaluates the alignment between the graybox baseline and the blackbox HMM in terms of their impact on guided fuzzing. We refer to this score as s_i, as it is based on the interestingness of the test cases. It is calculated as follows:

$$s_i = \frac{|T_b \cap T_h|}{|T_b \cup T_h|}, \tag{7.6}$$

where T_b and T_h denote the set of test cases deemed interesting by the baseline and the HMM, respectively.

Coverage Similarity Score

On the other hand, we propose a new score to compare the graybox behavior approximation based on code coverage and the blackbox HMM state coverage. Figure 7.7 shows an example of two coverage curves to motivate the proposed score. The x-axis represents the test cases generated by AFLnwe, while the y-axis shows the relative coverage achieved by both AFLnwe and the HMM. Note that we used the same test cases, in the same order, to calculate the graybox coverage using AFLnwe and the blackbox coverage using the HMM. With this, we allow for a direct comparison of these two coverage curves.

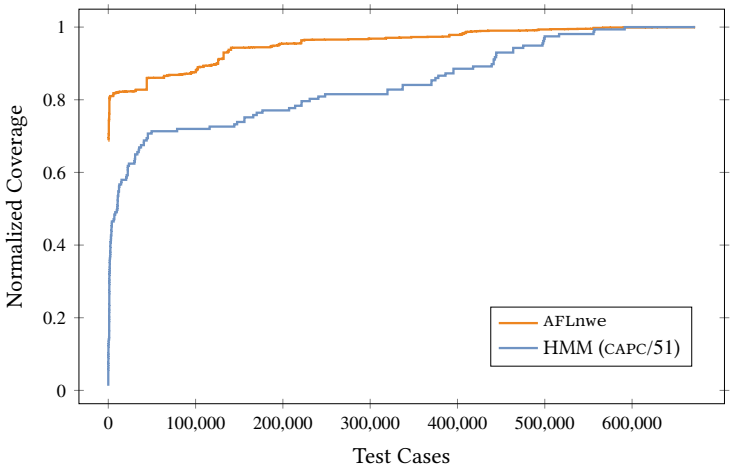


Figure 7.7: Coverage curves as measured based on AFLnwe and an HMM with 51 nodes which uses CAPC for the dimensionality reduction. Both curves are normalized to [0, 1].

While the HMM is not expected to perfectly model the code coverage directly but to approximate the SuT's behavior good enough to guide a fuzzer, a similar coverage curve indicates a comparable impact on the fuzzer's guidance. This, in turn, provides insights into the relative performance of the blackbox HMM and the graybox baseline. Since this score is derived from the coverage information, we call it s_c.

The general approach of s_c involves applying a sliding window approach to align increases in the HMM's coverage curve with corresponding increases in the baseline coverage curve. With this sliding window approach, coverage increases that occur in different points along the x-axis, which represents the sequence of test cases sent to the SuT, can be matched during the score calculation. This might be necessary if the baseline does not provide exact coverage information after each test case. Moreover, by using this sliding window approach, we allow small differences in the behavior approximation of the HMM and the baseline, as test cases next to each other tend to be similar. Nevertheless, the size of the sliding window can be set to 1 to enforce to only count perfect matches for the score calculation. Additionally, we consider the

gradient of the matched increases for the calculation of s_c. With this, the score reflects not only whether new coverage was generated but also the magnitude of the coverage increase produced by the respective test case.

s_c is calculated as

$$s_c = \frac{\sum_{i=0}^{|M|} d_i + |T_h^u| \cdot d_{max}}{|T| \cdot d_{max}}, \tag{7.7}$$

where M is the set of matched coverage increases between the two curves, and d_i the distance between the two coverage increases in a match $m_i \in M$. This distance includes the distance between the two increases along the x-axis and the distance between the gradient of the coverage increases. T_h^u refers to the set of test cases that lead to a coverage increase for the HMM, but not for the baseline, and T denotes the set of all test cases. d_{max} represents the maximum distance that could be achieved between two This formula sums the distances between coverage increases in the HMM coverage curve that have a matching within the sliding window in the baseline curve. Coverage increases in the HMM curve that lack a matching increase are penalized with the maximum possible distance. The sum is then normalized by dividing it by $|T| \cdot d_{max}$.

For a more detailed explanation of the calculation of s_c, including pseudocode, refer to the corresponding publication [*Här23*]. This publication also details why existing similarity scores for curves are not suitable for our use case of comparing coverage curves.

> We propose two different scores to assess the performance of the HMMs in comparison to a graybox baseline.
>
> 1 s_i, a score measuring the count of test cases deemed interesting based on the HMM approximation compared to the graybox baseline approximation.
>
> 2 s_c, a score based on the coverage curves generated by the HMM and the graybox baseline approximation.
>
> s_c provides a more detailed view on the HMM coverage approximation accurarcy, while s_i assesses the downstream implications of the approximation, particulary its impact on fuzzer guidance.

7.5.3.2 Experiments

In order to assess the performance of the blackbox HMMs in comparison to graybox AFLnwe, we run AFLnwe against the FTP stack ProFTP and compare the interestingness decisions by AFLnwe with those of the HMMs. More specifically, we perform the following steps.

1 Execute AFLnwe against ProFTP for 8 hours, recording all network traffic generated during the run. This results in 9 GB of network traffic which includes 672,214 test cases generated by AFLnwe.

2 Replay the recorded network traffic for the HMMs, and:
   a identify which test cases are considered interesting based on the HMM coverage, and
   b concurrently record the number of non-zero entries in the coverage map of the HMM.

3 Gather the test cases that are considered interesting by AFLnwe from the final corpus.

4 Calculate the scores s_i and s_c as defined in Section 7.5.3.1 utilizing the information collected from the above steps.

We execute AFLnwe in the first step to generate fuzzing data than we can then use to replay to the HMMs. As this data is not intended to evaluate the performance of the fuzzer itself, we

When interpreting the Interestingness Similarity Score s_i for the HMMs and AFLnwe, it is important to consider the following. AFLnwe considers test cases interesting if they lead to new coverage [Pha20]. Consequently, the interestingness of a test case is influenced by the sequence of test cases that was already sent and analyzed. For instance, if test cases t_1 and t_2 produce the same coverage, AFLnwe would not consider t_2 to be interesting if t_1 was analyzed first. However, if t_2 were processed before t_1, it would be deemed interesting. This scenario could lead to the HMM considering t_1 not interesting while AFLnwe deems it interesting, which in turn might cause the HMM to later consider t_2 interesting when AFLnwe does not.

This discrepancy would influence the score s_i calculated for the HMM, while it is not necessarily a poor decision by the HMM. Rather, it could stem from differences in how the SuT's behavior is represented. These differences are to be expected, as the HMM utilizes blackbox information only.

To account for this, we also analyze the distribution of interesting test cases over time when evaluating HMM performance. If the HMMs deem more test case interesting later in the process, it is more likely that the scenario described may have occurred. If the HMMs initially consider other test cases to be interesting, this could cause a propagation of differences through subsequent decisions, as each decision is influenced by earlier ones. In this case, other test cases would still be considered interesting later in the process, since the HMMs only have a slightly different interpretation as AFLnwe. However, if the HMMs report most interesting test case earlier in the process, it is more likely that they generally have a different representation, leading to an interpretation of the test cases differing from the one from AFLnwe. Note that we do not expect the HMMs to come to exactly the same conclusions as AFLnwe. Instead, we are interested in the HMMs' relative performance to understand the impact of the HMMs's number of nodes and the dimensionality reduction approaches better. Furthermore, we use these insights to decide which HMMs to use for the subsequent fuzzer evaluation (see Section 7.5.4).

Table 7.7: Comparison of the HMM-based interestingness decisions compared to AFLnwe's interestingness decisions. We report the number of test cases that were considered interesting by both the HMM and AFLnwe (True Positive (TP)), the number of test cases that where considered interesting by the HMM (False Positive (FP)), and the precision (Pr). The value reported for TP corresponds to the Interestingness Similarity Score s_i. In total, AFLnwe considered 3,220 out of 672,214 test cases to be interesting. The HMMs with the highest precision and highest TP for each dimensionality reduction approach ratio are highlighted for AE and CAPC.

HMM Nodes	PCA			AE			CAPC		
	TP	FP	Pr	TP	FP	Pr	TP	FP	Pr
7	1	9	0.10	1	15	0.06	2	9	0.18
18	2	6	0.25	2	40	0.05	2	39	0.05
27	2	6	0.25	2	65	0.03	1	7	0.12
38	1	8	0.11	4	45	0.08	3	56	0.05
51	2	20	0.09	5	116	0.04	5	109	0.04

In addition, we analyze the coverage curves of the HMMs to gain a deeper understanding of their performance. Note that the absolute coverage values of the HMMs and AFLnwe are not directly comparable, as these values depend on the total number of nodes in the respective HMM (N). Thus, we normalize the coverage curves to the interval [0, 1] for comparisons among the HMMs and AFLnwe.

The following sections present the results of our experiments with respect to the HMMs' performance, while Section 7.5.3.3 provides a discussion. Section 7.5.4 further explores and discusses the performance of fuzzers utilizing the HMMs described in this section. While evaluating the performance of the HMMs is important for understanding their general capabilities, the most significant performance indicator remains the overall effectiveness of the fuzzer.

Interestingness Similarity Score

We first calculate the Interestingness Similarity Score s_i for the HMMs. In addition to this score, we report the number of false positives, representing to the number of test case that have been classified as interesting by the respective HMM, but not by AFLnwe, and the precision. The precision is calculated as the ratio of true positives to the sum of true positives and false positives.

Table 7.7 presents these values for the considered HMMs, which differ in the dimensionality reduction approach used in NeDaP and in the number of hidden nodes (see Section 7.5.2.2). For example, among the test cases the HMM with 51 nodes using CAPC for dimensionality reduction deems interesting, five are also considered to be interesting by AFLnwe, while 109 are not. This results in a precision of 0.04.

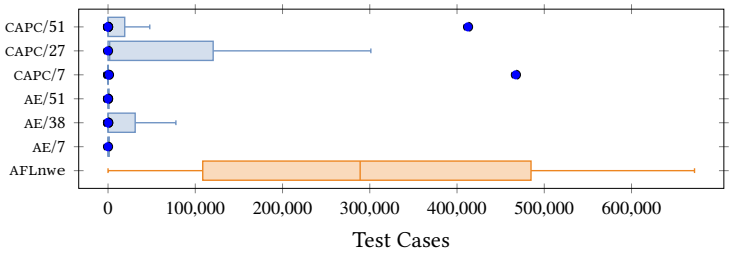
Interesting Test Case Distribution

We also report the distribution of test cases considered to be interesting by the HMMs over the total sequence of test cases, as illustrated in Figure 7.8. The figure displays a corresponding box plot for those HMMs that achieved the highest precision and AFLnwe (see Table 7.7). We exclude the HMMs utilizing PCA for dimensionality reduction with the highest precision from this analysis, as they only considered 8 test cases to be interesting, which is insufficient for statistical analyses. However, we include the HMMs with 51 nodes for both AE and CAPC, as these models achieved the highest true positive values (Table 7.7).

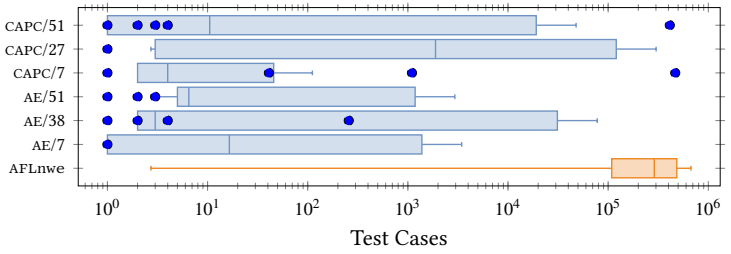
In Figure 7.8, the boxes represent the interquartile range, spanning from the lower quartile to the upper quartile, with the median indicated by the line within the box. The whiskers extend to the most distant points within 1.5 times the interquartile range. Test cases considered to be interesting by both AFLnwe and the respective HMM are marked with blue circles.

The two plots in Figure 7.8 display the same data, but with different scaling. Figure 7.8a provides a more intuitive visualization to analyze the distribution of interesting test cases across all test cases, wile Figure 7.8b uses a logarithmic scale to highlight the distribution of test cases that were deemed interesting by both AFLnwe and the respective HMM.

Figure 7.8a reveals that the majority of test cases deemed interesting by the HMMs are executed at the very beginning of the fuzzing campaign. For example, for the HMM with 38 nodes using AE (AE/38) results in a lower



(a) Distribution of test cases on a linear scale. Most test cases that are classified as interesting by the HMMs are located in the first half.



(b) Distribution of test cases on a logarithmic scale. Most test cases that are deemed interesting by an HMM and AFLnwe are located in the first 1,000 test cases.

Figure 7.8: Distribution of test cases that are considered to be interesting by the HMMs. Test cases that are deemed interesting by an HMM and AFLnwe are shown as blue circles. The boxes span from the lower quartile to the upper quartile, and the whiskers show the farthest point within the 1.5 interquartile range. The median is shown as vertical line in the box. In total, 672,214 test cases were executed.

quartile of 2.0 and an upper quartile of 31,122.75. In total, 672,214 test cases were executed. In contrast, the test cases deemed interesting by AFLnwe are distributed over the entire fuzzing campaign.

Figure 7.8b offers a more detailed view of the earlier stages of the fuzzing process. This figure shows that most test cases deemed interesting by both the HMMs and AFLnwe are located within the first 1,000 test cases. For example, for AE/38, these include test cases 1, 2, 4, and 257. Additionally, the figure highlights that all HMMs and AFLnwe agree in deeming the very first test case as interesting.

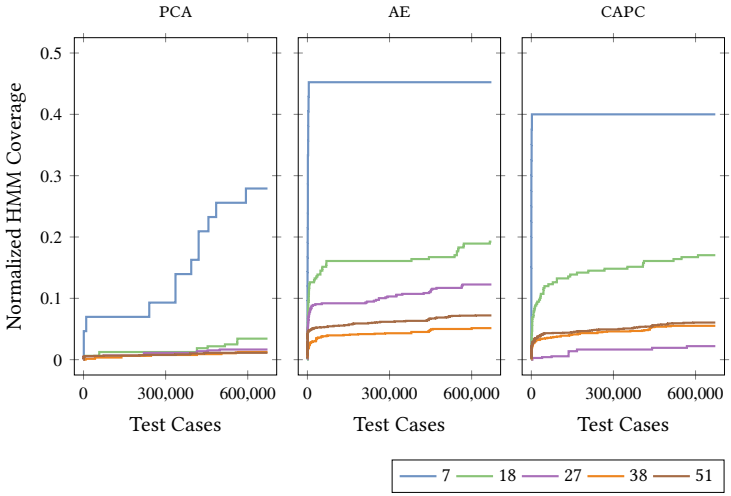


Figure 7.9: HMM transition coverage for different dimensionality reduction approaches and HMM sizes. The coverage is normalized to the interval [0,1]. The HMMs with 7 nodes reach their maximum coverage quickly, and achieve a relatively high coverage.

Coverage

While the previously reported results are based on the binary information on whether a test case is deemed interesting by the HMMs and AFLnwe, the following is based on the relative coverage as identified by AFLnwe and the HMMs. We base our experiments on the *line* coverage as reported by AFLnwe, and on the transition coverage within the hidden states of for the HMMs. Figure 7.9 illustrates the coverage of the HMMs over the test cases generated by AFLnwe. The coverage values for the HMMs are normalized to the interval [0, 1], allowing for a relative comparison of coverage across different HMMs.

The figure indicates that the HMMs with 7 nodes reach their respective maximum coverage relatively quickly, both for AE and CAPC. Additionally, when comparing HMMs with the same number of nodes but different dimensionality reduction approaches, it becomes evident that HMMs using PCA generally result in lover coverage values.

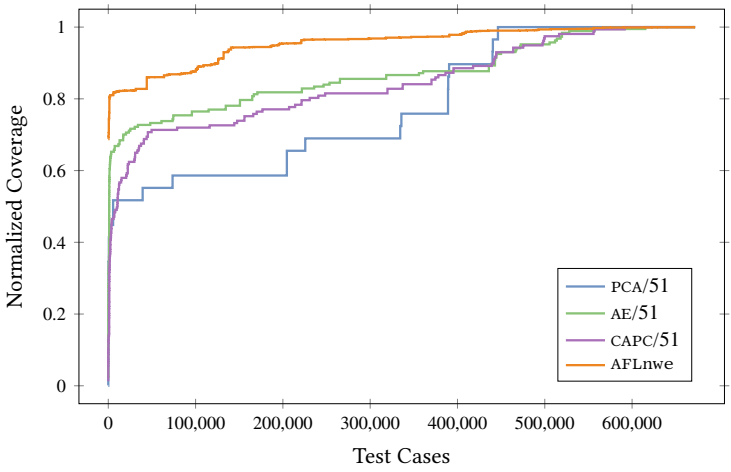


Figure 7.10: Coverage as measured by AFLnwe (line coverage) and three HMMs (transition coverage), normalized to the interval [0, 1].

Figure 7.10 presents the coverage of HMMs with 51 nodes compared to the line coverage reported by AFLnwe. In this case, all coverage values are normalized to the interval [0, 1], facilitating a more direct visual comparison. The figure shows that the coverage curves generated by AFLnwe generally have smaller gradients compared to those provided by the HMMs.

Coverage Similarity Score

To analyze the similarity between the coverage curves, we utilize the Coverage Similarity Score s_c as presented in Section 7.5.3.1. We calculate s_c for each of the considered HMMs and the line coverage as reported by AFLnwe. A smaller score indicates a higher similarity between the coverage curves.

The results of these calculations are presented in Table 7.8. It shows that, across all dimensionality reduction approaches, the coverage curves produced by the HMMs with 51 nodes exhibit the highest similarity to the coverage curve generated by AFLnwe. For example, for CAPC, the HMM with 51 nodes achieves a score of 0.5, whereas the HMM with 27 nodes achieves a score of 0.92. This is consistent with the visual representation of the coverage curves

Table 7.8: Coverage Similarity Score s_c calculated with respect to AFLNet line coverage, using a sliding window size of 37,000. The lowest score, which indicates the highest similarity between the HMM and the AFLNet coverage, is highlighted for each dimensionality reduction approach.

HMM Nodes	PCA	AE	CAPC
7	0.9	0.84	0.89
18	0.93	0.63	0.63
27	0.92	0.61	0.92
38	0.91	0.59	0.56
51	0.79	0.57	0.50

in Figures 7.9 and 7.10 and with the number of test cases deemed interesting by these two HMMs (Table 7.7). CAPC/51 produces to a coverage curve which is visually aligns more closely with AFLnwe's coverage curve, while CAPC/27 results in a curve with only few increases due to it considering only 7 test cases as interesting.

When interpreting the s_c scores with respect to AFLnwe, the following needs to be considered. AFLnwe does not record coverage for each individual test case, but only at fixed time intervals. Changing this would require larger changes in the code base. As a result, for the 672,214 test cases processed, only have 3,233 coverage measure points are reported by AFLnwe. To compare the coverage curves and to calculate s_c, we need to extend AFLnwe's coverage curve to match the x-axis range of the HMM coverage curves. This adjustment allows for comparison but introduces some inaccuracy to the s_c score.

Despite the limitation, we present the results of these calculations but acknowledge the constraint in our assessment of the HMMs' performance (see Section 7.5.3.3). To mitigate the impact of this inaccuracy, we set the sliding window size to 37,000, aligning with the time interval of AFLnwe's coverage measurements.

7.5.3.3 Discussion

While the HMMs are not expected to replicate AFLnwe's representation of the SuT exactly, comparing the two approaches offers valuable insights into the HMMs' performance.

Our calculations of the Interestingness Similarity Score s_i reveal that the HMMs consider fewer test cases as interesting compared to AFLnwe. This outcome aligns with our expectations, as the number of possible transitions in the HMMs is significantly smaller than the number of code lines considered by AFLnwe's coverage. Consequently, AFLnwe achieves finer granularity in coverage, allowing it to detect more subtle changes in the SuT's behavior and thus consider a greater number of test cases as interesting.

Consistent with this observation, our experiments suggest that HMMs with a larger number of nodes tend to consider more test cases as interesting compared to those with fewer nodes. This is because HMMs with more nodes can model the SuT's behavior with finer granularity, leading to a more detailed representation, and consequently, a higher number of test cases being deemed interesting.

Our analysis of the distribution of test cases deemed interesting by the HMMs reveals that most of these test cases are located in the first half of the sequence of test cases. Specifically, we observe that those test cases deemed interesting by both AFLnwe and the HMMs are located within the first 1,000 test cases. As discussed in the beginning of Section 7.5.3.2, this analysis provides insight into whether the HMMs merely exhibit slightly different interpretations of the test cases, leading to different decisions on the interestingness of a test cases which then has an impact on all following decisions. However, our findings suggest that the HMMs likely represent the SuT in a fundamentally different way, leading them to consider different test cases as interesting compared to AFLnwe.

It is important to note that this difference in representation does not necessarily imply that the HMMs are unsuitable for enhancing blackbox fuzzing. Nevertheless, it influences our choice of HMMs for subsequent experiments, as discussed later in this section.

Furthermore, the coverage curves generated by the HMMs reinforce our previous observations. For instance, HMMs that deem only a few test cases interesting, such as those with 7 nodes and those using PCA for dimensionality reduction, exhibit coverage curves with fewer steps. This pattern is expected, as each test case deemed interesting leads to an increase in the coverage curve. The consistency of our results across different metrics further validates the findings from our experiments.

Following the performance evaluation of the HMMs presented in this section, we assess the effectiveness of a fuzzer guided by such an HMM in Section 7.5.4. This evaluation examines the entire pipeline, starting from raw network traffic through NeDaP and the HMMs, and then using the HMMs' coverage information to guide the fuzzing process. The objective of this evaluation is to determine whether the HMMs can effectively improve the fuzzing process.

For this subsequent evaluation, we select specific HMMs to be used within a fuzzer. We choose to evaluate two fuzzers, each employing one of the following HMMs:

1. AE/51, the HMM with 51 nodes using AE for dimensionality reduction in the network packet preprocessing conduced with NeDaP, and
2. CAPC/51, the HMM with 51 nodes using CAPC for dimensionality reduction.

We come to this decision based on the following considerations. First, we decide not to further consider the HMMs that use PCA for dimensionality reduction. These HMMs consider only up to two 22 test cases interesting, suggesting that they lack the capability to guide the fuzzing process effectively over a 24-hour campaign.

Then, we evaluate the performance of the HMMs based on the two scores s_i and s_c. AE/51 and CAPC/51 achieve the highest scores in terms of s_i. However, considering the precision, AE/38 and CAPC/7 show the best performance. Our analysis indicates that s_i is the more important measure for our purposes. The distribution of test cases considered interesting by the HMMs is concentrated in the first half of the full sequence of test cases, whereas AFLnwe considers test cases interesting throughout the entire sequence. We infer that the differences in interestingness assessment is not merely based on subsequent faults, and thus focus on the number of True Positive results (s_i) of the HMMs as a basis to select HMMs to be used in the fuzzer. Based on s_i, AE/51 and CAPC/51 are considered the HMMs with the highest performance. Moreover, these two HMMs are also considered to be the two HMMs with the highest performance when evaluating s_c.

Based on these considerations, we decide to use AE/51 and CAPC/51 for our subsequent experiments.

7.5.4 HMM-based Fuzzer Assessment

To assess the performance of the downstream application of the HMMs, we implement four fuzzers based on LibAFL [Fio22]: the blackbox and random baseline fuzzers BLACKBOX and RAND as presented in Section 7.5.2.1, and two instances of `Palpebratum`, one using AE/51 and one using CAPC/51. We run each of the fuzzers for 24 hours against ProFTP (see Section 7.5.2.3), and repeat each run 30 times to account for the fuzzers' randomness. Our experimental setup for these runs is based on `ProFuzzBench` [Nat21]. As initial corpus, we use the seeds provided by `ProFuzzBench`. With this, we follow the recommendations by Klees et al. [Kle18]. To assess the performance of the fuzzers, we measure the code coverage in basic block hits using the code provided by `ProFuzzBench`. We run our experiments on an Ubuntu 20.04 LTS server with an Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz CPU (6 physical cores) with 125 GB of RAM. To ensure a fair resource distribution between the runs, we only run six experiments in parallel.

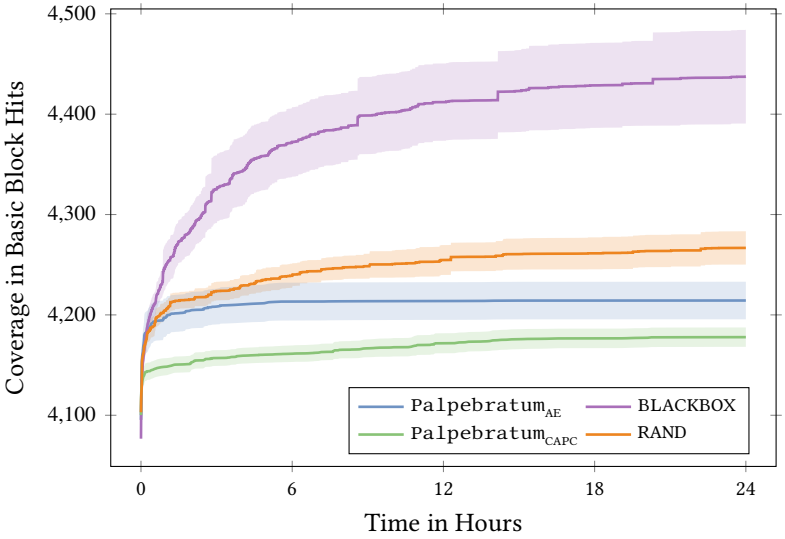


Figure 7.11: Coverage in basic blocks for the four fuzzers over the course of 24 hours. Each line represents the mean of 30 runs while the shaded area represents the 95% confidence interval. BLACKBOX and RAND achieve significantly higher coverages than $\texttt{Palpebratum}_{\text{AE}}$ and $\texttt{Palpebratum}_{\text{CAPC}}$, and $\texttt{Palpebratum}_{\text{AE}}$ significantly outperforms $\texttt{Palpebratum}_{\text{CAPC}}$.

7.5.4.1 Results

We first report the coverage in basic block hits achieved by the different fuzzers over the course of the 24 hour campaign. Note that this coverage is calculated based on the evaluations provided by ProFuzzBench and thus calculates only the coverage of the test cases deemed interesting by the respective fuzzer. Figure 7.11 shows the mean of 30 runs as well as the 95% confidence interval. It demonstrates that BLACKBOX performs better than RAND, and that RAND performs better than the two instances of Palpebratum, $\texttt{Palpebratum}_{\text{AE}}$ and $\texttt{Palpebratum}_{\text{CAPC}}$. Moreover, $\texttt{Palpebratum}_{\text{AE}}$ outperforms $\texttt{Palpebratum}_{\text{CAPC}}$.

Table 7.9: Experimental results for the four fuzzers. We report the final coverage in basic blocks, the coverage achieved by the newly generated test cases, the number of test cases deemed interesting, and the ratio between the coverage and the number of interesting test cases. All values shown for the blackbox fuzzers represent the mean of 30 runs, while AFLnwe has been run six times. T_n denotes the set of interesting test cases newly generated by the fuzzer, thus excluding the seeds, and C_n denotes the coverage generated by the test cases in T_n

Fuzzer	Total coverage	C_n	$\lvert T_n \rvert$	$C_n / \lvert T_n \rvert$
$\texttt{Palpebratum}_{\text{AE}}$	4214.33 (15.31%)	118.93	59.47	2.0
$\texttt{Palpebratum}_{\text{CAPC}}$	4177.83 (15.18%)	77.67	76.97	1.01
BLACKBOX	4437.43 (16.12%)	360.77	4735.83	0.08
RAND	4266.74 (15.50%)	163.74	3690.43	0.04
AFLnwe	5137.17 (18.66%)	1078.0	3311.83	0.33

We verify these results by calculating a two-sided Mann-Whitney U test with the significance level of $\alpha = 0.05$. Our results show that for every pair of coverage distributions, the test rejects the null hypothesis that the two fuzzers achieve the same coverage with a p-value < 0.01. This supports the visual impression given by Figure 7.11.

For a more comprehensive view of the fuzzers' performance, we report additional values on the fuzzing campaigns in Table 7.9. Each value shown for the blackbox fuzzers represents the mean of 30 runs. AFLnwe was only run 6 times as these values only serve as a baseline to compare the absolute values of the blackbox fuzzers against and are not used for statistical calculations themselves. We report the total coverage achieve by the corpus generated by each fuzzer after 24 hours. This value corresponds to the rightmost values of the plot shown in Figure 7.11. For example, $\texttt{Palpebratum}_{\text{AE}}$ achieve a total coverage of 4214.33 basic blocks, while the graybox fuzzer AFLnwe covers 5137.17 basic blocks.

Moreover, we report C_n, which denotes the coverage achieved by the corpus excluding the seeds given to the fuzzer Thus, this value shows the coverage that is achieved by the corpus newly generated by the fuzzers. The test cases newly generated by $\texttt{Palpebratum}_{\text{AE}}$ achieve a coverage of 118.93 basic blocks.

In order to highlight the efficiency of the test cases, Table 7.9 also reports the ratio between the C_n and the number of the test cases in the corpus that were newly generated by the fuzzers. This value shows the coverage that was achieved per test case on average. For $\texttt{Palpebratum}_{\text{AE}}$, this ratio is 2.0, while RAND achieves a ratio of 0.04.

7.5.4.2 Discussion

Our experiments with respect to the performance of the HMM-guided fuzzers provide the following insights.

Impact of HMM First, our results suggest that the choice of the underlying HMM, including the choice of the preprocessing algorithm, influences the fuzzer performance. $\texttt{Palpebratum}_{\text{AE}}$, based on an HMM with 51 nodes which uses AE for preprocessing, results in a significant higher coverage than $\texttt{Palpebratum}_{\text{CAPC}}$, based on an HMM with 51 nodes using CAPC for preprocessing. This is in line with the experiments conducted with NEDAP. Within these experiments, AE showed better out-of-domain generalization capabilities than CAPC, which was expected to have a positive impact on the downstream fuzzers' performance (see Section 6.4.3.3).

Relative Achieved Coverage Our results indicate that $\texttt{Palpebratum}_{\text{AE}}$ and $\texttt{Palpebratum}_{\text{CAPC}}$ achieve significantly less code coverage than the two baseline algorithms BLACKBOX and RAND. As expected, it shows that the graybox fuzzer AFLnwe achieves more coverage than the blackbox fuzzers.

As stated before, we base our experiments on the coverage measurement as usually conducted for graybox fuzzing evaluations and as implemented by ProFuzzBench [Nat21]. For this coverage measurement, test cases that are included in the corpus after the fuzzing campaign are collected and replayed against a version of the SuT that was compiled for coverage analysis. However, the corpus only includes those test cases that were deemed interesting by the fuzzer, and the initial seeds. Thus, the measured coverage only shows the coverage of those test cases and not of the whole fuzzing campaign.

In graybox fuzzing evaluations, this does not impact the reliability of the coverage measurement, as the fuzzer uses the same coverage analysis to guide its fuzzing and thus correctly selects those test cases for the corpus that leads to new coverage. In blackbox fuzzing however, the fuzzer cannot decide correctly which test cases to include in the corpus and thus might miss test cases that lead to new coverage. Thus, it is to be expected that the reported coverage underestimates the coverage of the fuzzing campaign for $\texttt{Palpebratum}_{\text{AE}}$, $\texttt{Palpebratum}_{\text{CAPC}}$, and RAND. As BLACKBOX includes all test cases in the corpus, the coverage measurement for BLACKBOX includes the full coverage of the fuzzing campaign. Thus, the coverage reported for these experiments helps to understand how the coverage of the interesting test cases evolves over the fuzzing campaign. However, for an evaluation of the total achieved coverage, a novel performance measure that includes all test cases generated by the fuzzer would be necessary.

Effectiveness of test cases Nevertheless, the reported coverage provides insights into the effectiveness of the generated test cases, measured as the ratio of achieved coverage and generated test cases (see Section 2.4.3). As BLACKBOX deems all test cases interesting, the effectiveness of its test cases can be used as a baseline for relative comparisons. The test cases generated by RAND are less efficient than the ones generated by BLACKBOX. A reason for this might be that the coverage is not uniformly distributed over the test cases and thus a random selection of test cases might exclude those test cases leading to higher coverage. Nevertheless, both $\texttt{Palpebratum}_{\text{AE}}$ and $\texttt{Palpebratum}_{\text{CAPC}}$ generate test cases with a higher effectiveness, showing that the test cases that are deemed interesting by the approaches actually improve coverage. Interestingly, $\texttt{Palpebratum}_{\text{AE}}$ deems less test cases interesting than $\texttt{Palpebratum}_{\text{CAPC}}$, but achieves a higher final coverage. This again highlights the impact of the HMM chosen as a basis for the fuzzer.

Efficiency Another factor that might influence the relative performance of the baseline fuzzers and the HMM-based fuzzers is their general *efficiency*. In line with Gopinath et al., we define the efficiency of a fuzzer as the number of test cases that are executed in a given time frame [Gop22]. In comparison to the two baseline fuzzers, the HMM-based fuzzers need to capture and preprocess

the network traffic, and query the HMM to determine the interestingness of a test case. For the baseline fuzzers, this decision is either trivial as it returns always `true` for BLACKBOX, or is based on a simple query to a random oracle for RAND. As a result, it would be expected that the baseline fuzzers show a higher efficiency than the HMM-based fuzzers. With this, the assessment of relative performance of the HMM-based fuzzers in comparison to the baseline fuzzers might change in favor of the HMM-based fuzzers. However, the metrics as provided by `ProFuzzBench` do not allow to analyze the efficiency of the evaluated fuzzers. Thus, additional measurements would be needed to further analyze the impact of the fuzzers' efficiency on the code coverage they achieve.

Impact of model quality In addition, our experiments demonstrate that BLACKBOX performs significantly better than RAND. Recall that BLACKBOX utilized no information on the behavior of the SuT and deems all test cases interesting, while RAND randomly decides whether a certain test case is deemed interesting. With RAND, each test case has a probability of 0.5 to be interesting. Both BLACKBOX and RAND are directly comparable in terms of their efficiency as they base on the same fuzzer and only differ in the function that defines the interestingness of a test case. Thus, as BLACKBOX performs better than RAND, this suggests that selecting the wrong test cases can decrease the fuzzing performance and that in this case, just selecting all test cases for the corpus might be beneficial. This insight was also shown by our experiments with respect to `Smevolution`, where the fuzzer based on an SVM yielded significantly less vulnerability coverage than the random approach (see Figure 7.2). These results emphasize purely blackbox approach might achieve higher coverage than a model-based fuzzer if the model employed does not capture relevant information.

7.5.5 Related Work

`Palpebratum` models the behavior of the SuT as expressed by the network traffic using HMMs. Markov chains and HMMs have been used to model and to generate network traffic by various works. Wright et al. use a Markov chain to model and to generate user events which lead to network traffic which is

then used to test network applications [Wri10]. Note that in this case, the Markov chain is used to model and generate user events and not to generate network traffic directly.

Dainotti et al. propose to use an HMM to model network packet behavior from internet traffic sources [Dai08]. More specifically, they model the inter packet time and the packet size with the HMM. The authors apply their approach to SNMP, HTTP, the network game *Age of Mythology*, and MSN messenger. Their experiments show that the HMM is able to model the dependencies and temporal structures of the network traffic. Moreover, they show that the HMM is able to replicate and predict future network packet properties. In contrast to the work by Dainotti et al., we use the network packets themselves as input to the HMM instead of using traffic flow parameters. With this, we allow for a more detailed representation of the single packets. This accounts for the differing use case as Dainotti et al. handle large sequences of internet network traffic, while `Palpebratum` considers short sequences of fuzzer traffic.

Gascon et al. present PULSAR, which uses a second-order Markov chain to represent the network packets [Gas15]. The main goal of PULSAR is to provide a generative model which can analyze and simulate network traffic to support a blackbox fuzzer. More specifically, PULSAR learns a second-order Markov chain which represents the state machine of the protocol under test as well as a set of templates and rules which define the specifics of a network packet, and the dependencies between network packets. This Markov chain is then transformed to a Deterministic Finite Automaton (DFA). This DFA is used to generate fuzzing responses to requests received by the SuT, leveraging so-called *fuzzing masks* which indicate which parts of the network packet should be fuzzed next. The authors evaluate PULSAR using FTP and the proprietary network protocol OSCAR. In contrast to `Palpebratum` which aims to approximate the SuT's behavior to guide fuzzing, PULSAR focuses on modelling the network protocol in detail to allow generating new packets as response to a SuT's requests. With this, more training data and modelling overhead is necessary within PULSAR.

Moreover, several publications use Markov chains and HMMs to model different aspects of blackbox and graybox fuzzing (see e.g. [Zhu22]). Böhme et al., Rawat et al. present approaches which apply a Markov chain to model the seed selection problem in graybox fuzzing [Böh16, Raw17], while Salem et al. use a Markov chain for blackbox test case generation [Sal21].

7.5.6 Discussion

The following sections discuss our results by first addressing the research questions in Section 7.5.6.1. Then, we discuss the implications of our findings in Section 7.5.6.2, and present identified limitations in Section 7.5.6.3. Section 7.5.6.4 outlines possible future research directions.

7.5.6.1 Research Questions

Our experiments address the research questions presented in Section 7.5.2, and our findings are discussed in the following.

RQ1 How can we assess the performance of the HMMs `Palpebratum` is based on?

We present two scores that can be used to assess the performance of the HMMs. First, we present s_i, a score that compares the count of test cases deemed interesting based on the HMMs to a graybox approach. Second, we present s_c, a score based on the coverage curves generated by the HMMs and a graybox baseline. The first score directly compares the assessment of the HMM to the assessment of a graybox baseline. In our experiments, the second score demonstrates that it is capable to represent the similarity of coverage curves, both for expected coverage curves and for edge cases [*Här23*].

Nevertheless, as both scores use a graybox fuzzer as baseline, they suffer from a limitation stemming from the graybox interestingness assessment. The graybox fuzzer used as a baseline deems those test cases interesting which lead to new coverage, considering the coverage all previous test cases achieved. Thus, if a HMM classifies one test cases differently than the graybox baseline,

this leads to consequential differences in the interestingness assessment, resulting in a high impact of a single difference in classification on the score. This needs to be considered when assessing the performance of HMMs based on the proposed scores.

RQ2 How does the behavior approximation of the HMMs compare to the behavior approximation done by AFLnwe?

Our experiments demonstrate that the HMMs consider fewer test cases interesting compared to AFLnwe. Moreover, it shows that HMMs with more nodes tend to lead to more interesting test cases. Both outcomes are in line with the general observation that a model with more nodes can represent the information with a higher granularity. Thus, as each identified difference in behavior leads to a test case to be considered interesting, models with more nodes are expected to lead to more interesting test cases. As AFLnwe considers the coverage in basic blocks, it represents the SuT's behavior with the highest granularity.

Comparing the specific test cases that are deemed interesting by the HMMs and AFLnwe as well as comparing the coverage curves shows high differences. For example, AE/51 deemed 121 test cases interesting, of which only 5 where also deemed interesting by AFLnwe. Moreover, as most test cases deemed interesting by the HMMs are located in the first half of the fuzzing campaign, it less likely that the differences are based on subsequent errors. Nevertheless, this insight does not necessarily imply that the HMMs are unsuited to improve the performance of blackbox testing, as they still might represent information relevant for testing.

RQ3 Does the HMM-based behavior approximation improve blackbox fuzzing in terms of code coverage?

We measure the code coverage as usual in graybox fuzzing evaluations by calculating the coverage achieved by the test cases deemed interesting by the respective fuzzer. Based on this measure, the two baseline fuzzers BLACKBOX and RAND outperform the two HMM-based approaches $\texttt{Palpebratum}_{\text{AE}}$ and

Palpebratum$_{\text{CAPC}}$. However, as the HMM-based approaches tend to deem less test cases interesting, this coverage measure is expected to underestimate the performance of the HMM-based fuzzers.

In addition, we analyze the effectiveness of the test cases by analyzing the coverage achieved by an interesting test case on average. Our results indicate that the HMM-based approaches generate test cases that show a higher effectiveness than those generated by the baselines. Palpebratum$_{\text{AE}}$ leads to 2.0 basic blocks per test cases on average, while Palpebratum$_{\text{CAPC}}$ achieves a values of 1.01, and the two baselines BLACKBOX and RAND values of 0.08 and 0.04, respectively.

In summary, the HMM-based approaches generate more effective test cases, but achieve less final coverage. The latter is influenced by the coverage measurement which potentially underestimates the actual coverage of the HMM-based approaches. Moreover, the HMMs introduce an overhead to the fuzzers. As a result, the baseline fuzzers operate more efficiently and generate more test cases per time frame. This potentially also influences the relative assessment of the fuzzers.

RQ4 Which impact does the dimensionality reduction approaches used to preprocess the network traffic have on the HMMs' and the fuzzers' performance?

The dimensionality reduction approaches have an impact on the performance of the HMMs as well as on the performance of the subsequent fuzzers.

With respect to the performance of the HMMs, measured by the Interestingness Score s_i and the Coverage Score s_c, our results demonstrate a performance impact. While the HMMs using PCA for dimensionality reduction tend to deem less test cases interesting, the performance of HMMs based on AE and CAPC achieve comparable results.

Comparing the fuzzer performance of Palpebratum$_{\text{AE}}$, which uses AE for preprocessing, and Palpebratum$_{\text{CAPC}}$, which uses CAPC for preprocessing, provides the following insights. Palpebratum$_{\text{AE}}$ significantly outperforms Palpebratum$_{\text{CAPC}}$ in terms of final code coverage, while also deeming less test

cases interesting. Specifically, $\texttt{Palpebratum}_{\text{AE}}$ achieves a mean final coverage of 4,214.33 basic blocks, while $\texttt{Palpebratum}_{\text{CAPC}}$ achieves 4,177.83. The test cases generated by $\texttt{Palpebratum}_{\text{AE}}$ achieve a coverage of 2.0 per test case on average, while the test cases generated by $\texttt{Palpebratum}_{\text{CAPC}}$ achieve a mean coverage of 1.01. Thus, the test cases generated by $\texttt{Palpebratum}_{\text{AE}}$ are more efficient. This observation suggests that the choice of the dimensionality reduction approach has a significant impact on the downstream fuzzer performance.

7.5.6.2 Implications

Our experiments with `Palpebratum` show that HMM-based blackbox fuzzing is able to generate test cases that are more effective than those generated by the baseline fuzzers. With this promising result, it demonstrates how a blackbox information source, the network traffic generated during testing, can be leveraged to guide test case generation. This builds a starting point for applying established graybox testing approaches to blackbox testing in order to improve the overall blackbox test performance.

In addition, comparing the two baselines used in our experiments suggests that choosing an unsuitable model can lead to a performance decrease. More specifically, we use a blackbox fuzzer which deems all test cases interesting, and compare it to a random fuzzer for which each test case has the probability of 0.5 to be interesting. The blackbox fuzzer shows a significantly better performance than the random fuzzer. As the random fuzzer behaves like a fuzzer using an unsuitable model, this indicates that using a blackbox approach can be preferable over using a model-based approach with an unsuitable model.

7.5.6.3 Limitations

For our experiments, we used established metrics from graybox testing, which showed to suffer from certain limitations when applied to blackbox fuzzing evaluations. Especially, our experiments identified the need for refined metrics for blackbox fuzzing evaluations. On the one hand, the proxy metric of measuring code coverage of the interesting test cases usually used in graybox

fuzzing evaluations has its limitations when applied to blackbox fuzzing as ist might underestimate the actual achieved coverage. On the other hand, proposing new means to approximate the behavior of a SuT requires a metric to assess the quality of this approximation. In this work, we proposed two scores to compare the interestingness assessment of blackbox approaches to a graybox baseline. While these scores provide valuable means to quantify the performance, they suffer from the fact that the interestingness of a test case depends on previous test cases in graybox fuzzing. Using a known ground truth to compare the blackbox interestingness assessment against would ensure reliable and comprehensible scores. However, how such a ground truth could be acquired remains an open question for further research.

For our experiments regarding the fuzzers' performance, we utilized ProFTP as a target. While this provides insights into the performance of the fuzzers, an evaluation including more targets would provide additional insights with respect to the general performance of the fuzzers.

7.5.6.4 Future Work

Currently, the HMMs used for `Palpebratum` are trained on a dataset of user data generated with several FTP servers. Future work could analyze the impact of the training dataset on the performance of the HMMs and the downstream fuzzers. Specifically, the similarity of HMMs trained on datasets which each include user-generated traffic for one FTP server could be analyzed. For this, one could use the distance measure for HMMs as presented by Juang et al. [Jua85], for which an implementation is provided by Manuel Pineda[1]. However, preliminary analyses conducted during the course of this doctoral work suggest that there is no correlation between the distance of HMMs and the FTP server that was used to generate the respective training data [*Mum21*].

Nevertheless, this similarity analysis could provide insights with respect to the transferability of the HMMs. To reduce the overhead necessary for each test, it would be beneficial to train one HMM which then could be used to test several

[1] https://github.com/pin3da/hmm-dist/blob/master/src/probdist.py

SuTs. This could be feasible if the HMMs trained for different SuTs are similar enough. A similar approach was taken by Aichernig et al. [Aic21]. The authors first train a behavioral model of the general system, and then utilize the model to test specific SuTs. They apply their technique to test implementations of the network protocol Message Queuing Telemetry Transport (MQTT) which is often used in Internet of Things (IoT) environments, and reveal several bugs.

While we chose HMMs for `Palpebratum` due to their explicit state representation and suitability for applications with only small training datasets, future work could investigate whether more recent approaches could be applied to approximate the SuT's behavior as well. LSTMs or RNNs could be possible choices for these investigations.

7.5.7 Summary

We present `Palpebratum`, an approach to utilize the network traffic accessible before and during a blackbox test to approximate the behavior of the SuT. Specifically, we train a HMM on the network traffic prior to testing. During testing, we query the HMM to decode the sequence of network packets generated by one test case, resulting in the sequence of hidden states that has the highest probability to having produced this sequence of network packets. We use this state sequence to define the interestingness of a test case by deeming those test cases interesting that take transitions in the HMM that were not taken by any previous test case.

Based on this approximation, we apply coverage-guided mutational graybox fuzzing to a blackbox test setting by integrating it into the fuzzing framework LibAFL [Fio22]. We evaluate two instances of `Palpebratum`, one based on an HMM using AE to preprocess the network traffic and one based on an HMM using CAPC, against a blackbox fuzzer and a fuzzer receiving random feedback. We execute each fuzzer for 24 hours, using ProFTP as target, and repeating each run 30 times to account for the fuzzers' randomness.

Our experiments indicate that `Palpebratum` is able to generate effective test cases, but the coverage achieved by the test cases deemed interesting is significantly less than the one achieved by the two baseline fuzzers. This difference could be caused by an underestimation of the actual coverage achieved by `Palpebratum`, and differences in efficiency, i.e. the number of test cases that are generated during the course of 24 hours. Moreover, our experiments show that the choice of the HMM impacts the fuzzer's performance significantly.

7.6 Multi-Armed Bandit

In contrast to the two approaches presented above, `StateBandit`, as detailed in this section, does not construct an explicit model of the SuT's behavior. Instead, `StateBandit` applies RL to address a specific decision in fuzzing: the state selection problem encountered in stateful network fuzzing. *Stateful* fuzzing recognizes that the SuT incorporates several states and incorporates explicit or implicit knowledge on the states in the fuzzing strategy (see e.g. [Ba22, Dou12, Liu22] and Sections 2.4 and 2.5). Consequently, the fuzzer can, for example, target deeper states within the SuT to uncover vulnerabilities located there. Those vulnerabilities are less likely to be revealed by stateless fuzzers, since it is usually very unlikely that they provide the specific sequence of inputs necessary to reach the vulnerable state [Ba22]. Since network protocols mostly encompass multiple states, stateful fuzzing is an important tool to consider for a thorough test of network interfaces and stacks.

Stateful fuzzers need to effectively select the next state of the SuT that should be tested, which is generally called the state selection problem [Liu22]. In general, for each fuzzing round, stateful fuzzers choose one state of the SuT that should be tested during this cycle. The SuT is then placed into this chosen state, and a predetermined number of fuzzing cycles are run while the SuT is in this specific state. Subsequently, in the next round, the fuzzer repeats this process by selecting the next state for testing (see also Section 2.5). The decision on which state should be tested in the next fuzzing round would be

expected to have an impact on the efficiency of the fuzzing process. However, Liu et al. show in their work that the different current state selection algorithms do not lead to significant differences in a fuzzer's performance [Liu22].

In this section, we introduce a novel approach to address the state selection problem in stateful network fuzzing by formulating it as a RL problem, specifically a Multi-armed Bandit (MaB) problem (Section 7.6.1). We choose to use a graybox test setting for this approach to analyze whether an MaB agent can utilize graybox information to approach the state selection problem and thus to ultimately improve fuzzing. Since we leverage the MaB problem to approach the state selection problem, we call our approach `StateBandit`.

For our experiments, we choose two different approaches to formulate the state selection problem: a stochastic MaB, and an adversarial MaB. The main difference between these two approaches is that a stochastic MaB assumes stationary reward distributions while an adversarial MaB can handle non-stationary reward distributions (see Section 7.6.2 for more details). In fuzzing, only *new* code coverage or *new* crashes are rewarded. As more code is covered over time, the amount of code that can be newly covered in a state of the SuT changes over time, leading to a non-stationary reward distribution. Thus, we expect that the adversarial MaB to model the state selection problem better and thus expect a respective fuzzer to perform better (see Section 7.6.1.1 for details).

Conversely, our experiments show that both approaches yield statistically indistinguishable fuzzer performance in terms of code coverage. Furthermore, we find that the baseline fuzzer `AFLNet` achieves significantly better results compared to the MaB-based approaches. These findings from graybox testing suggest that attempting to formulate the state selection problem as a MaB problem in a *blackbox* setting would likely be unsuccessful, given that the agent would receive even less information. Therefore, we have decided not to pursue this research direction further in this doctoral work. Nevertheless, the work on formulating the state selection problem in *graybox* network fuzzing, as presented in this section, was published with the EuroS&P workshops [*Bor23a*]. The work was conducted in collaboration with Mark Giraud and is based on the Bachelor's thesis by Ian Fitzgerald [*Fit22*].

7.6.1 Approach

The state selection required for efficient stateful fuzzing can be viewed as an instance of the exploitation versus exploration dilemma. For fuzzers, it is crucial to exploit states that have previously yielded new coverage and crashes, while also exploring new states to uncover additional coverage and crashes. For example, when testing a network protocol, there might exist a state in which the connection of the network protocol is fully established and data packets can be sent. Intuitively, this state is the one that includes most functionality of the SuT and thus should be exploited to cover this functionality. However, the other states of the network protocol, such as intermediate connection establishment states, need to be tested as well to reveal bugs there.

One possibility to formulate exploitation versus exploration problems is through MaB problems. In MaB problems, a non-contextual RL agent selects from actions to maximize the cumulative rewards resulting from the actions [Sut18]. We propose to formulate the state selection problem as MaB and leverage established algorithms designed for the MaB problem to approach the state selection problem. The intuition for this is that the actions the agent selects are mapped to the states of the SuT and thus the agent selects the state of the SuT that should be tested in the next fuzzing round. As soon as one fuzzing round is finished, the agent is queried to select the next action and thus the next state of the SuT to be tested.

7.6.1.1 Multi-armed Bandit

In a MaB problem, figuratively speaking, an agent faces multiple slot machines (one-armed bandits) and aims to maximize its total winnings over several rounds. In each round, the agent needs to decide which slot machine to pull, knowing that each slot machine has its own underlying probability distribution from which rewards are drawn. Over time, the agent can learn these probability distribution for each slot machine. Alternatively, one can envision this scenario as an agent facing a single slot machine with multiple independent arms (see Figure 7.12), which forms the name of the Multi-armed Bandit problem.

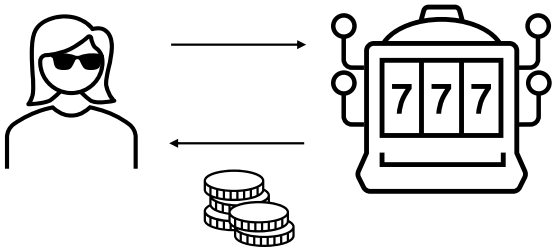


Figure 7.12: Visualization of the Multi-armed Bandit problem. In each round, the agent needs to decide which arm of the slot machine to pull. Then, the agent receives a reward based on the reward distribution of that particular arm.

More formally, in the MaB problem, a non-contextual agent repeatedly selects from several actions, each leading to rewards drawn from a probability distribution specific to that action. The agent's goal is to maximize the expected total reward. Without prior knowledge of the reward distributions, the agent needs to decide whether to exploit known actions or explore unknown ones. For our work, we consider a non-contextual MaB, where the agent receives only its reward after selecting an action, without receiving any additional information about the environment, the fuzzer, or the SuT influencing its decision [Bou20]. Note that the MaB assumes a *stateless* environment and thus the agent receives a reward directly after a single action was chosen.

Literature proposes various approaches to formulate MaB problems [Sli19]. In general, all approaches balance between exploiting known good actions for short-term benefit, and exploring unknown actions for long-term benefit. However, these approaches diverge in how they achieve this balance and what kind of reward distributions they assume. In our setting, we adopt two distinct approaches: a *stochastic* MaB and an *adversarial* MaB. The main difference between these two approaches lies in the nature of the reward distributions. For the stochastic MaB, the reward is based on stationary probability distributions, whereas these distributions can change over time for the adversarial MaB [Aue02b].

The general approach of a stochastic MaB is to balance between exploration and exploitation while aiming to find the action that yields the highest reward, while also aiming to minimize the cumulative regret by converging to the best action as quickly as possible. In contrast, fuzzing aims to maximize the general coverage and bugs found, which is potentially distributed over different states of the SuT.

Adversarial MaBs do not assume stationary reward distributions and thus the corresponding algorithms are designed to handle the case that the reward for a specific action changes over time. We base the name of this MaB on its original publication by Auer et al., who establish the setting of an adversary controlling the rewards, as opposed to the rewards being drawn from a probability distribution [Aue02b]. In fuzzing, success is generally determined by the discovery of *new* coverage or *new* bugs. Repeatedly encountering the same coverage or bugs does not provide new insights and should not yield review for the agent. Thus, our hypothesis is that the adversarial MaB is suited better to approach the state selection problem in fuzzing, since the reward for this problem changes over time.

For both approaches, various policies have been proposed. In our experiments, we use the following commonly used policies. For the stochastic MaB, we choose the the ε-greedy algorithm [Sut18], the UCB1 algorithm [Aue02a], and a tuned version of UCB [Aue02a]. Accounting for the differences in a stochastic MaB, which assumes a stationary reward distribution, and the state selection problem, we configure the stochastic algorithms to give a rather high weight on the exploration. For example, we choose $\varepsilon = 0.2$ for the ε-greedy algorithm. For the adversarial MaB, we choose the Exp3 algorithm [Aue02b].

ε-greedy The ε-greedy algorithm selects the action with the highest expected reward with a probability of $1 - \varepsilon$, exploiting the knowledge on rewarding probability distributions. To also account for exploring new probability distributions, ε_{GR} chooses a random action with a probability of ε [Sut18].

UCB1 The UCB1 algorithm presented by Auer et al. [Aue02a] takes the known uncertainty with respect to an action a into account, representing this by the ratio of the total choices that have been taken and the number of times action

a has been chosen [Sut18]. With UCB1, all actions will be selected eventually. However, actions that have already been chosen frequently or that lead to a lower expected reward are chosen with decreasing frequency over time.

Tuned UCB Auer et al. also present a tuned version of the UCB algorithm [Aue02a], which also takes the estimated variance of an action's rewards into account. In contrast to UCB1, the authors do not derive theoretical guarantees for this algorithm, but it shows that it outperforms the other algorithms in several experiments [Aud07].

Exp3 The Exp3 algorithm, presented in a different publication by Auer et al. [Aue02b], makes no statistical assumptions about the distributions of rewards, but assumes an arbitrary and unknown sequence of rewards for each action. Thus, it is better suited to model the state selection problem for stateful fuzzing, in which the reward is not drawn from a stationary distribution, but can change over time. As the specifics of this algorithm are not essential for understanding the following descriptions and evaluations, we refer to the original publication for more details on Exp3 [Aue02b].

7.6.1.2 Problem Formulation

Regarding the state selection problem in stateful network fuzzing, the agent's task is to select the state of the SuT to be tested during the next fuzzing cycle. Note that the agent still operates in a stateless environment, since each state of the SuT represents a possible action for the agent, but does not translate to a state for the agent.

Assumptions We make the following assumptions as a basis for formulating the MaB problem.

1 The fuzzer has access to the deterministic state machine that represents the behavior of the SuT. Amongst other information, that state machine includes the set of states the SuT can be in, denoted as S.

2 Each state in S is reachable from the start state.

3 The fuzzer can coerce the SuT into a specific state s, $s \in S$.

Given these assumptions, the state machine itself can be transparent for the agent. Note that this state machine is needed for the fuzzing process, but does not influence the agent and the environment the agent operates in. For further details on how the fuzzer, the agent, and the SuT interact, please refer below.

Actions We define the set of possible actions A for the agent to be equal to the set of states in S ($A \sim S$). Note that S represents the states of the SuT, whereas the agent, operating in a non-contextual environment, remains stateless. With this construction and our assumptions, each action selection of the agent corresponds to selecting a state from the SuT's state machine.

Reward The agent receives a reward if it finds *previously unseen* coverage or crashes. New coverage is rewarded with a reward of 1, and a new crash is rewarded with a reward of 10. If the agent finds neither new coverage nor a new crash, it receives a reward of 0. Since our ultimate goal is to apply `StateBandit` to blackbox fuzzing, we already aggregate the information available in graybox fuzzing and thus give the agent a relatively coarse-grained reward. Note that this results in a non-stationary reward distribution. For our experiments, we choose three algorithms that are designed for stationary reward distributions, and one algorithm which is also designed for non-stationary reward distributions (Exp3 algorithm). Thus, we would expect the Exp3 algorithm to yield better performance.

Framework The interaction between the agent, the fuzzer, and the SuT is illustrated in Figure 7.13. When the agent selects an action a, it determines the subsequent state s of the SuT for testing, since $A \sim S$. Then, the SuT is put into state s by sending the necessary inputs to the SuT. Following the state transition, the fuzzer generates the fuzzing input and sends it to the SuT. The SuT executes the input, and metrics such as code coverage and crashes are captured. Note that our setup assumes a graybox scenario where we have access to code coverage data during fuzzing. Based on these measurements, the fuzzer computes the reward for the agent and provides it accordingly. This completes one round of the fuzzing process, prompting the agent to repeat the cycle by selecting the next state. Note that this entire process operates in a non-deterministic manner due to the fuzzer's non-deterministic mutations, and the agent's non-deterministic choices. Additionally, note that we assume

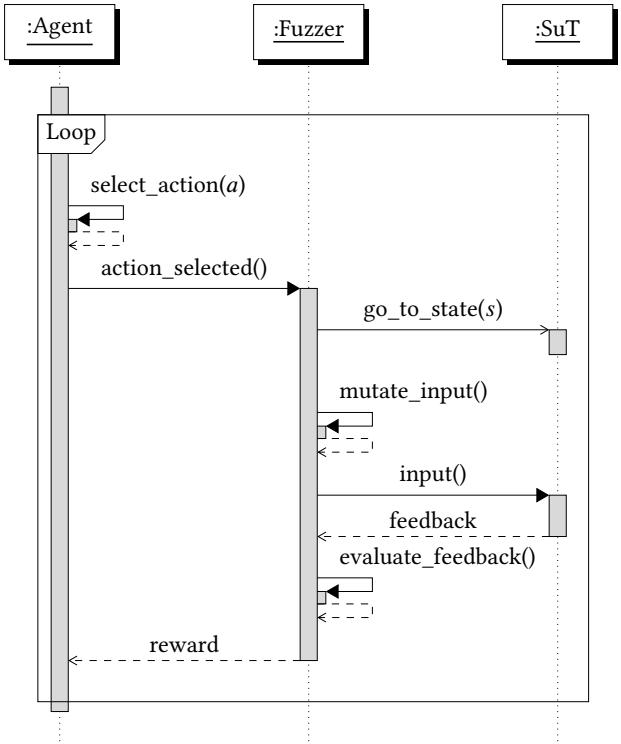


Figure 7.13: Sequence diagram of the interaction between the agent, the fuzzer, and the SuT (based on [*Bor23a*]).

the agent to train its policy during the fuzzing process, meaning that it still balances between exploration and exploitation in the setting as opposed to selecting the action leading to the highest rewards deterministically.

7.6.2 Experiments

Our experiments aim to investigate the implications of formulating the state selection problem in stateful fuzzing as MaB problem and employing an MaB agent to approach the state selection. To this end, we implement the four MaB-based fuzzers based on the four agent policies presented above. Subsequently,

we run these fuzzers against an implementation of the highly stateful network protocol OPC Unified Architecture (OPC UA), and measure the achieved code coverage. As a baseline, we use the state-of-the-art stateful graybox fuzzer AFLNet [Pha20].

Our experiments indicate that all four MaB-based fuzzers achieve statistically indistinguishable code coverages. This finding is interesting since, as discussed above, the adversarial MaB approach was expected to model the state selection problem better and thus to result in a higher code coverage. Nonetheless, this aligns with the findings of Liu et al., whose results also demonstrate similar cove coverages for the different state selection approaches they analyzed [Liu22]. Moreover, the baseline AFLNet outperforms the MaB-based fuzzers significantly, showing possible future research directions to further improve the MaB-based fuzzers (see also Section 7.6.4.3).

7.6.2.1 Experimental Setting

We run our experiments on an Ubuntu 20.04 LTS server with an Intel® Xeon® CPU ES-1650 v3 @ 3.50GHz (12 physical cores) with 64 GB of RAM. As SuT, we choose open62541[1], an open source implementation of OPC UA.

Research Questions

Our experiments are driven by the following two research questions.

- *RQ1* Do the different policies for the MaB agent affect the performance of stateful fuzzers based on a MaB state selection?
- *RQ2* How do stateful fuzzers using a MaB-based state selection compare to the state of the art fuzzer AFLNet?

[1] https://www.open62541.org/

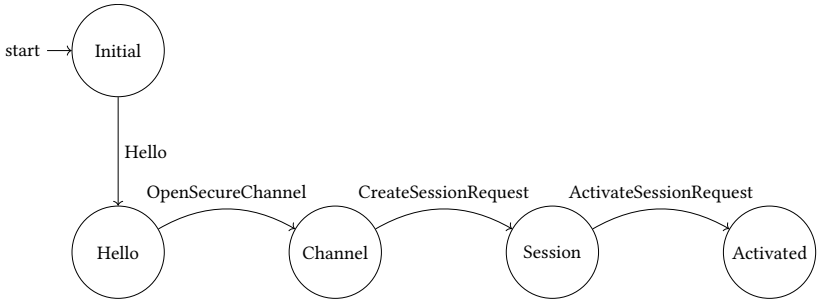


Figure 7.14: State machine of OPC UA that is used for the MaB-based fuzzers (based on [*Bor23a*]). The protocol states as well as the OPC UA messages that are necessary to transition between these states are shown. During fuzzing, the MaB agent first selects one of these states, and the fuzzer then sends the OPC UA messages to reach this state to the SuT, followed by the fuzzing input.

Methodology

To approach these questions, we implement four MaB-based fuzzers based on AFL++ [Fio20]. Each of these fuzzers implements one of the policies ε-greedy, UCB1, tuned UCB, and Exp3. We call these fuzzers εGR, UCB, UCBT, and EXP, respectively. As baseline, we use AFLNet [Pha20], a stateful network fuzzer that uses a manually crafted heuristic to approach the state selection problem. This heuristic is based on the number of times a state has been selected in the future, and on how successful the fuzzer was in finding new coverage in this state [Pha20].

As stated before, our current approach requires a pre-defined state machine of the network protocol that should be tested. Thus, we create a state machine including the five states of OPC UA that are needed to establish connection (see Figure 7.14). When started, the SuT is expected to be in the initial state of the state machine. By sending OPC UA messages, the fuzzer can subsequently transition the SuT into the state selected by the MaB agent. Afterward, the fuzzer sends its fuzzing input and thus test the SuT while it is in this specific state. For our implementation, we rely on the mutation-based input generation approach as implemented in AFL++ [Fio20]. Note that this state machine is

not intended to encapsulate the entire behavior of the SuT. Instead, it focuses on representing states of the SuT relevant for fuzzing. Therefore, it excludes error states nor the states necessary to close a session.

We run our four fuzzers against the SuT open62541, and measure the code coverage the fuzzers achieve over time. As coverage metric, we choose the basic block hits as measured by AFL-based fuzzers [Wan19].

To ensure reliable results, we follow the recommendations by Klees et al. [Kle18]. These recommendations include executing each fuzzing run for a minimum of 24 hours, and repeating each configuration at least 30 times to account for the randomness of the fuzzer. However, given the exploratory nature of our experiments in this chapter, we run each fuzzer configuration 10 times for 24 hours. Note that this still accounts for the randomness of the MaB algorithms.

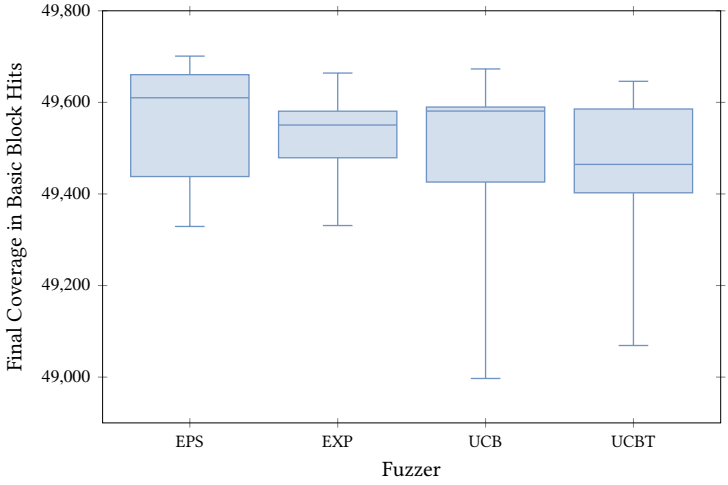


Figure 7.15: Box plot of the final coverages achieved by 10 runs of the MaB-based fuzzers. The boxes show the range between the lower and upper quartile, the whiskers represent the minimum and maximum. The lines in the boxes show the median. None of the MaB-based fuzzers leads to significantly different results.

7.6.2.2 Results

We present the results of our experiments in this section, while Section 7.6.2.3 discusses the results and their implications with respect to our research questions.

Comparison between MaB-based Fuzzers

First, we present the results achieved by the four MaB-based fuzzers. Figure 7.15 illustrates the final coverage that has been achieved across 10 runs of the four MaB-based fuzzers. In the figure, each box spans from the lower quartile to the upper quartile, with the median depicted as a line within the box. The whiskers extend the minimum and the maximum. Higher coverage indicates better performance for each respective fuzzer.

The box plot indicates that there is no statistically significant difference in performance amongst the four fuzzers. To validate this statistically, we conduct a Mann-Whitney U test and present the resulting p-values in Table 7.10. Since none of these p-values is smaller than 0.05, we cannot reject the null hypothesis that all fuzzers achieve the same final coverage. This finding confirms the visual impression provided by Figure 7.15. We choose the threshold of the p-value $\alpha = 0.05$ according to established best practices [Kle18, Paa21].

Table 7.10: Resulting p-values of a Mann-Whitney U test regarding the statistical significance of the distance of the resulting coverages. No algorithm performs significantly better than one of the others ($\alpha = 0.05$).

	UCB	**UCBT**	**εGR**	**EXP**
UCB	-	0.545	0.364	0.879
UCBT	0.545	-	0.198	0.449
εGR	0.364	0.198	-	0.542
EXP	0.879	0.449	0.542	-

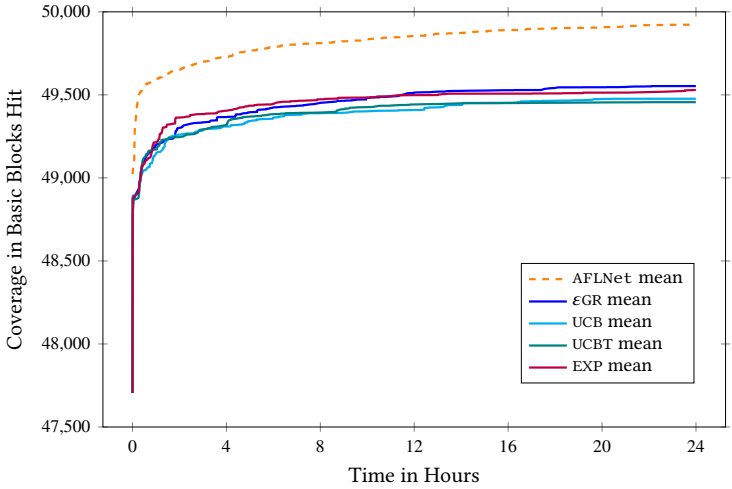


Figure 7.16: Mean code coverage achieved by 10 runs of the MaB-based fuzzers and AFLNet. AFLNet outperforms the MaB-based fuzzers.

Comparison to AFLNet

For an evaluation of the absolute performance of the MaB-based fuzzers, we compare them against the state-of-the-art stateful fuzzer AFLNet. Figure 7.16 displays the mean coverage over time for 10 runs of each of the fuzzers. With this, we follow the usual presentation of fuzzing results [Kle18], since our evaluation focuses on the fuzzing performance of the MaB-based fuzzers. For a more detailed analysis of the performance of the MaB agents, one would need to analyze the performance in terms of total actions taken as opposed in terms of time. Again, higher coverage indicates better performance for each respective fuzzer. The figure clearly demonstrates that AFLNet clearly outperforms the MaB-based fuzzers. Moreover, AFLNet achieves higher coverage right from the start of the fuzzing campaign compared to the MaB-based fuzzers.

Additionally, Figure 7.16 provides insights into the relative performance of the MaB-based fuzzers. While Figure 7.15 details the final coverage achieved by the fuzzers, Figure 7.16 illustrates the mean coverage throughout the course of the fuzzing campaign. Although these differences are not statistically significant,

it is noticeable that the EXP algorithm, which considers non-stationary reward distributions, performs better initially. Towards the end of the campaign, εGR (based on the ε-greedy algorithm), achieves the highest mean coverage.

Note that we decided to omit the presentation of the confidence band in this figure to improve the visibility of the mean values. However, our statistical analysis as presented in Table 7.10 shows that the four MaB-based fuzzers to not result in significant differences, while AFLNet significantly outperforms the MaB-based fuzzers.

7.6.2.3 Discussion of Results

Our experiments result in new insights regarding the research questions posed in Section 7.6.2.1, which we discuss below.

RQ1 Do the different policies for the MaB agent affect the performance of stateful fuzzers based on a MaB state selection?

Our experiments demonstrate that all four MaB-based fuzzers achieve statistically indistinguishable performances. This result is noteworthy considering that the state selection problem in fuzzing inherently involves a non-stationary reward distributions. Thus, we would expect the EXP fuzzer to perform better, as it employs an algorithm specifically designed for non-stationary reward distributions. Nevertheless, our results align with the findings by Liu et al. [Liu22], who show that existing approaches for the state selection problem result in comparable fuzzing performances in most cases.

RQ2 How do stateful fuzzers using a MaB-based state selection compare to the state of the art fuzzer AFLNet?

Compared to the baseline AFLNet, all MaB-based fuzzers show significant lower values for the final code coverage. This observation suggests that the manually crafted heuristic used in AFLNet outperforms the policies utilized by the RL agents in terms of code coverage. However, for a more thorough comparison between these two approaches, it is essential to consider the inherent differences between the AFL++-based MaB fuzzers and AFLNet. For

instance, AFLNet requires a full sequence of input messages as input, while the agents do not receive any additional information on the states and their structure. Consequently, AFLNet can start by fuzzing the deeper states, where it might expect to find more coverage and bugs. However, some preliminary analyses that we did based on the work presented in this chapter show that fuzzing deeper states do not necessarily lead to more coverage than fuzzing the other states. Thus, to draw more conclusive insights, a comprehensive analysis of these differences would be necessary.

7.6.3 Related Work

Several publications have formulated different aspects of the fuzzing process as a RL or MaB problem, demonstrating improvements in overall fuzzing performance. However, to the best of our knowledge, no publication has yet formulated the state selection problem as MaB problem.

Seed Selection One crucial aspect of the fuzzing process is determining the seed for the next fuzzing cycle. Gohil et al. formulate this decision as a MaB problem, focusing on simulation-based hardware fuzzing [Goh24]. Similar to our experiments, the authors employ ϵ-greedy, UCB, and EXP3 policies. Moreover, the authors address the non-stationary reward distributions by explicitly deleting seeds from the corpus, effectively "resetting" one of the arms of the Multi-armed Bandit [Goh24]. Furthermore, the authors modify the policies to accommodate these resets. In their experiments, the authors show that their MaB-based approach achieves an increased vulnerability detection speed when compared to the baseline TheHuzz. Note that this publication was published after the publication of the work presented in this section.

Yue et al. propose to formulate the power schedule process of AFL, used for seed selection, as a MaB problem [Yue20]. The authors show that their approach achieves a higher code coverage while using less test cases than the baseline AFL.

In another study, Fang et al. apply a general RL approach to formulate the seed selection problem within the domain of fuzzing 4G/LTE Android devices [Fan18b]. In contrast, Woo et al. address the seed selection problem by utilizing the *Coupon Collector's Problem*, focusing on finding unique bugs during fuzzing [Woo13]. Their approach also includes determining the mutation ratio for each seed, which is why the refer to the problem they are approaching as the *scheduling problem* of fuzzing.

Coverage Metric Selection Wang et al. employ an RL-based approach to determine the coverage metric with the appropriate level of detail for the next fuzzing cycle. Their experiments compare this approach to AFL and AFLFast, demonstrating enhancements over these baselines in terms of both bug discovery and achieved coverage [Wan21b].

Mutation Selection Binosi et al. use an RL-approach to decide on the position within the current input that should be mutated. The authors show that their approach, `Rainfuzz`, outperforms a random policy and that its performance can be further improved by combining it with AFL++ in a collaborative fuzzing setting [Bin23]. Wu et al. model the mutation selection using a MaB which is tasked to select the number of mutators and the mutators to be used [Wu22a].

Böttinger et al. formulate the problem of deciding on which mutations to perform on a given input as a RL problem. Their approach involves an RL agent selecting actions corresponding to specific probabilistic mutation rules. The authors show that their approach performs better than a random policy [Böt18].

While proposing various improvements with respect to graybox fuzzing, Pham et al. also suggest using a RL approach to select one out of nine possible mutation actions [Pha24]. As one result of their evaluation, they show that their approach outperforms AFL++.

Fernandez et al. present a grammar-based fuzzer in which a MaB is used to decide on how to generate the next test case [Fer22]. For this, they introduce annotations to the grammar which enable or disable certain parts of the grammar's definition. The MaB then decides which annotations to enable and thus influences how the next test case is generated.

State Selection Problem While our approach to formulate the state selection problem as a MaB problem is novel, there exist other approaches addressing this problem in stateful fuzzing. Guo et al. propose to use a state selection algorithm based on Monte Carlo Tree search [Guo24]. However, the authors do not evaluate the performance of the state selection algorithm separately but focus on the general performance of the fuzzer.

Liu et al. compare the three state selection approaches implemented in AFLNet against a newly proposed approach called AFLNETLEGION [Liu22]. The three approaches implemented in AFLNet are FAVOR, RANDOM, and ROUND-ROBIN. FAVOR, which we also use for our evaluation, uses a manually crafted heuristic that aiming to balance exploration and exploitation. Liu et al. show with their experiments that the three algorithms implemented in AFLNet achieve comparable code coverage. Moreover, they find that while AFLNETLEGION achieves higher coverage than the AFLNet algorithms for specific cases, it does not consistently improve the overall fuzzing performance. StateBandit, as presented in this section, enhances the work by Liu et al. by implementing additional state selection approaches based on the MaB problem. However, our experiments suggest that the policy that is used by the MaB agent does not significantly impact the overall fuzzing performance, with all MaB-based fuzzers achieving lower coverage compared to AFLNet using the FAVOR algorithm.

7.6.4 Discussion

This section sets the approach and the results of StateBandit in a broader context and explains their implications (Section 7.6.4.1). Moreover, we discuss the limitations of StateBandit (Section 7.6.4.2) and outline possible directions for future work (Section 7.6.4.3).

7.6.4.1 Implications

The state selection problem in stateful fuzzing remains an ongoing area of research where optimal algorithms and their impact on fuzzing performance are not yet fully understood. With StateBandit, we contribute to this field

by introducing the new approach of formulating the state selection problem as a MaB problem and comparing different agent policies. Our findings include that the different agent policies we evaluated resulted in statistically indistinguishable fuzzer performances. This aligns with existing literature, which also reports indistinguishable fuzzer performances for several approaches to the state selection problem. Moreover, we demonstrate that AFLNet significantly outperforms the MaB-based fuzzers. While is comparison provides valuable context for understanding the absolute performance of MaB-based fuzzers, more detailed experiments are needed to directly compare the MaB-based state selection to the state selection approaches used by AFLNet.

We conduct our experiments in a graybox test setting with the ultimate goal of transferring the results to blackbox testing. However, since the MaB-based fuzzers did not achieve comparable or superior results to state-of-the-art state selection approaches, it is not to be expected that they would perform better in a blackbox test setting. Evidently, the agents are unable efficiently leverage the aggregated graybox information provided via the reward to enhance state selection. Consequently, we anticipate that the agents would similarly struggle to compete with blackbox state selection approaches, as the rewards would incorporate even fewer information. Therefore, we decide not to pursue this research direction further in the context of this doctoral work. Nevertheless, we point out several possibilities to improve upon `StateBandit` in the domain of graybox testing (Section 7.6.3).

Even though `StateBandit` was not suited to improve the state selection process, it revealed a bug in the SuT, the open source OPC UA stack open62541. We reported the finding and the bug was fixed by Mark Giraud[1].

7.6.4.2 Limitations

Our experiments aim to provide initial insights into whether formulating the state selection problem as MaB problem has the potential to improve a graybox fuzzer's performance and how the chosen MaB policy influences the

[1] https://github.com/open62541/open62541/pull/4906/commits

fuzzer's performance. For a more thorough analysis and evaluation, it would be necessary to expand the experiments by increasing the number of runs per configuration and by increasing the number of tested SuTs. Moreover, to directly compare the performance of the MaB-based approaches with the heuristic used in AFLNet, both strategies would need to be implemented within the same fuzzer. Note that for our experiments, we implemented the MaB-based approaches based on AFL++.

7.6.4.3 Future Work

Future research could investigate approaches to improve the performance of the MaB-based fuzzers. One approach would be to explicitly incorporating knowledge of how the rewards are affected by exclusively rewarding *new* coverage and crashes in the agent's policies. The approach presented by Gohil et al. [Goh24] could be used as a basis for these new policies.

Additionally, conducting thorough hyperparameter tuning could lead to discovering more optimal configurations for the agents, thereby improving their efficacy in state selection. Another potential approach for future work could involve integrating our MaB-based approaches into AFLNet or implementing both strategies within the fuzzing framework LibAFL [Fio22]. The latter approach would modularize the state selection algorithms, facilitating easier reuse and comparison with other algorithms.

Furthermore, exploring the deeper question of how different state selection strategies influence the fuzzer's performance remains pertinent. Both the findings from Liu et al. [Liu22] and our experiments suggest that the choice of state selection algorithms may not significantly influence the fuzzer's performance. However, substantiating this observation would require more extensive evaluations and experiments.

7.6.5 Summary

We present a novel approach to address the state selection problem in stateful graybox network fuzzing by formulating it as a MaB problem. Within this framework, we explore four different agent policies, with one of the policies specifically designed to accommodate the non-stationary reward distributions in fuzzing caused by only rewarding *new* coverage and crashes. We implement each policy in a separate fuzzer based on AFL++, and we conduct evaluation amongst them and against the baseline AFLNet. Our experiments show that all four MaB-based fuzzers achieve statistically indistinguishable performances, while AFLNet significantly outperforms the MaB-based approaches. These results are consistent with those reported by Liu et al., who show that different approaches addressing the state selection problem lead to comparable code coverage.

7.7 Testing BC_{ex}

As we integrated `Smevolution` into the industrial security testing framework ISuTest®, it can be applied to test OT components such as BC_{ex} directly. However, as `StateBandit` is a graybox approach and `Palpebratum` is based on the software fuzzing framework LibAFL, both approaches need significant additional work to apply them to OT component testing in practice.

`Smevolution` reveals one new Denial of Service (DoS) vulnerability of BC_{ex} based on an IP packet. If one sends an empty IP packet which includes the IP address of `102.251.189.167` as source address, the WA of BC_{ex} crashes and remains unresponsive until BC_{ex} is power cycled. Further testing with `Smevolution` revealed additional anomalies in the behavior of BC_{ex}, as the FTP server becomes temporarily unresponsive for several test cases. However, BC_{ex} shows resilience with respect to these test cases and the FTP server becomes responsive again, without the need to power cycle BC_{ex}.

7.8 Discussion

In this discussion, we aggregate the findings and insights from the three ML-based approaches presented in this chapter: `Smevolution`, `Palpebratum`, and `StateBandit`.

7.8.1 Implications

The presented approaches leverage ML models trained on different data sources to improve the testing process. `Smevolution` and `Palpebratum` aim for a targeted test case generation in a blackbox test setting, while `StateBandit` addresses the state selection problem in graybox fuzzing. Both `Smevolution` and `Palpebratum` show that the trained models are able to improve upon the baseline fuzzers in some aspects, highlighting the potential of ML-based blackbox fuzzing. For `Smevolution`, utilizing a DT or NN as ML model improves performance, while using an SVM decreases performance. Our experiments with `Palpebratum` suggest that the HMM fuzzers generate more efficient test cases, and that the choice of the dimensionality reduction approach used for data preprocessing has a significant impact on a HMM-based fuzzer's performance. These results indicate that ML-based fuzzing has the potential to improve blackbox testing performance, but that the underlying models need to be chosen carefully. Utilizing ML models that are unsuited for the given task can lead to a performance worse than the one achieved by a blackbox fuzzer that does not incorporate any feedback from the SuT.

For `StateBandit`, we chose a graybox test setting to analyze whether the MaB-based approach is able to improve performance in this scenario. However, it shows that the novel approach cannot outperform the state-of-the-art, which uses a manually crafted heuristic. This further supports the insight that, even with graybox information, models need to be adequate to the given task in order to actually improve the fuzzing performance.

Moreover, the presented approaches demonstrate that blackbox fuzzing can be improved by utilizing the accessible information with selected models, and how graybox fuzzing approaches can be applied to blackbox testing. The presented approaches and insights advance the domain of blackbox fuzzing and build the basis for further improvements.

7.8.2 Limitations

The evaluations presented in this chapter are based on an artificial SuT and on software implementations of network stacks. The artificial SuT allows to control the exhibited vulnerabilities, while the network stacks allow for measuring graybox baselines to compare the blackbox approaches against. While using these SuTs strengthens the experimental possibilities with respect to the foundational capabilities of the approaches, this also results in limited conclusions with respect to the practical applicability of the approaches in OT component testing.

For our evaluations, we implemented several fuzzers to evaluate the performance of the proposed approaches. While these allow for an evaluation of the approaches, the implementations have not been improved with respect to general performance. Thus, to compete against fuzzers that consider general runtime performance, the implementations presented in this work would need to be refined.

7.8.3 Future Work

Leveraging the insights presented in this doctoral work, future work could analyze different ML models which could further improve blackbox fuzzing. For example, LSTMs or RNNs could be used for processing the network traffic generated during testing. Moreover, transformer models could be used for network traffic processing or for test case generation.

Our evaluations demonstrate that existing metrics for fuzzing evaluations focus on graybox fuzzing and show limitations when applied to blackbox testing. Future work could investigate additional metrics that are tailored for a blackbox test setting, and analyze how these metrics influence the assessment of the performance of blackbox fuzzers.

In addition, future work could examine the optimization objective that guides the fuzzing process for blackbox fuzzing. The blackbox approaches presented in this chapter aim to either maximize the number of crashing services or an approximated coverage. While maximizing the number of crashing services helps in finding vulnerabilities, a fuzzer might potentially focus on crashes that are close to each other, while missing crashes that are farther away. Here, the distance between two test cases is defined by the number of mutations required to generate one test case from the other. Maximizing code coverage is an established optimization goal which is shown to correlate with the number of bugs a graybox fuzzer finds [Böh22]. However, it remains an open question whether the approximated coverage can achieve similar results. Thus, it might be beneficial to design optimization objectives specifically for blackbox testing that consider these insights.

7.9 Summary

This chapter presents three approaches that leverage ML techniques to address challenges in fuzzing. `Smevolution` employs an ML model to approximate a function mapping a test case to the services of the SuT that are expected to crash in response to said test case. We implement three instances of `Smevolution`, using a DT, a NN, and an SVM as underlying ML model, respectively. Our experiments based on a SuT exhibiting artificial vulnerabilities show that the fuzzer based on the DT outperforms the evolutionary blackbox baseline fuzzer and a random fuzzer significantly, while the fuzzer based on the NN only outperforms the random fuzzer significantly. The fuzzer based on the SVM achieves significantly less performance than the random fuzzer and the blackbox baseline fuzzer.

`Palpebratum` leverages an HMM to represent the behavior of the SuT as it is exhibited in the network traffic generated by the SuT. For our experiments, we compare two instances of `Palpebratum`, which use an HMM with 51 nodes each and use AE and CAPC for dimensionality reduction respectively, to a blackbox and a random baseline fuzzer. The results demonstrate that the HMM-based fuzzers overall generate more efficient test cases, while achieving less final coverage. Moreover, it shows that the instance of `Palpebratum` using AE for dimensionality reduction significantly outperforms the instance using CAPC.

`StateBandit` focuses on stateful graybox fuzzing and leverages an MaB agent to address the state selection problem. We implement four different approaches for the agent, three using a policy for a stochastic MaB problem and one a policy for an adversarial MaB. Our experiments reveal that the four fuzzers based on one of the agent policies respectively do lead to statistically indistinguishable performances. Moreover, the state-of-the-art fuzzer `AFLNet` outperforms the MaB-based fuzzers significantly.

In summary, the three approaches show that ML-based approaches have the potential to utilize the information accessible in blackbox fuzzing to improve the fuzzing performance. However, it also shows that the characteristics of the used ML model significantly influence a blackbox fuzzer's performance.

8 Evaluation of Test Tools

The preceding chapters of this doctoral work focussed on proposing new testing approaches and Test Tools (TTs), each accompanied by an evaluation in accordance with established evaluation guidelines such as the recommendations for fuzzing evaluations by Klees et al. [Kle18]. Additionally, this doctoral work contributes to the research field of evaluating TTs, specifically with respect to (1) the evaluation of stateful web application testing, and (2) the evaluation of stateless fuzzers. Both contributions are described in the following.

8.1 Problem Statement

As discussed in Section 1.3 with respect to Challenge 5 (Choice of Testing Tool), several possible TTs exist for the different areas of testing. To make an informed decision, a tester needs to retrieve information on the performance of the TTs. Moreover, researcher and developers who propose new TTs or enhancements of existing TTs need to evaluate whether their new TT or their enhancement improves upon the current state-of-the-art. This requires established evaluation frameworks and strategies.

In this doctoral work, we concentrate on the evaluation of two subtopics of industrial security testing: (1) blackbox Stateful Web Application Testing (SWAT), and (2) stateless fuzzing. This focus aligns with the topics covered in the previous chapters.

Evaluating Blackbox SWAT For general blackbox Web Application (WA) testing, several studies have been published, comparing the bug finding capabilities of various open source and commercial Web Vulnerability Scanners (WVSs) (see Section 4.3.2). However, those studies mostly focus on the WVSs

as a whole and do not directly analyze the performance impact of individual design choices or testing approaches within the WVSs. With this, it is more difficult to identify the root causes of performance differences between the WVSs. This is especially true for SWAT for which fewer comparative evaluations exist (see Section 8.3.2) although some approaches specific to SWAT have been proposed [Dou12, Has22]. Thus, their relative performance and the impact of the authors' design choices on their performance remain unclear. Notably, a modular approach allowing to easily change and thus to compare the impact of individual design choices on the performance of stateful WVSs does not yet exist. Therefore, we identify the need for a modular evaluation framework focusing on stateful blackbox WVSs. For this contribution, we focus on blackbox SWAT with a special emphasis on the task of automatically inferring a state machine of the System under Test (SuT).

Evaluating Stateless Graybox Fuzzers In the domain of fuzzing, evaluation guidelines have been established, most notably those proposed by Klees et al. [Kle18]. Literature shows that most fuzzing evaluations use metrics such as the code coverage or the number of bugs a fuzzer reveals for assessing the relative performance of a fuzzer [Kle18, Böh22]. This finding is also supported by a more recent study by Schloegel et al. [Sch24]. However, a study by Fioraldi et al. indicates that the relative performance of fuzzers depends on the chosen evaluation metric [Fio22]. Moreover, other metrics used in traditional software testing and benchmarking, such as the memory utilization and CPU load, have not been extensively applied in fuzzing evaluations. Therefore, we identify the need to evaluate the impact of both established and new fuzzing metrics on the relative performance assessment of fuzzers. For this, we focus on stateless graybox fuzzing, since this is the subdomain of fuzzing which received most attention recently [Mal23].

8.2 Contributions

This doctoral work makes two contributions to the research area of evaluating TTs: `SWaTEval` and `MEMA` (see Table 8.1). `SWaTEval` focuses on blackbox SWAT, while `MEMA` focuses on stateless graybox fuzzers.

Table 8.1: Comparison of the two contributions of this doctoral work with respect to evaluating TTs. During the work on MEMA, a more extensive work on the same topic was published [Li21], and we thus refrain from publishing this work separately.

	SWaTEval	**MEMA**
Evaluation target	Blackbox state machine inference	Stateless graybox fuzzing
Approach	Modular evaluation framework	Analysis of the impact of different performance metrics on the evaluation results
Publication	[*Bor23c*]	-

SWaTEval We design and implement a modular evaluation framework for stateful blackbox WA testing. We call it SWaTEval, short for Stateful Web Application Testing Evaluation.

Contribution 9. *Proposal, implementation, and evaluation of SWaTEval, an evaluation framework for SWAT.*

For our evaluation, we focus on specific design choices related to the *state machine inference* necessary for stateful blackbox testing. In blackbox testing, we cannot assume that information on the SuTs's state machine is available. Thus, we need to infer the state machine dynamically during the test to allow for stateful testing. This inference requires determining whether two responses from the SuT should be considered the same (see Sections 2.3 and 8.3.1). In our case, the responses of the SuTs are web pages, and thus a similarity measure for web pages is necessary. For this similarity measure, several options exist and we choose an analysis of these options as use case for our evaluation of SWaTEval. With this use case, we first demonstrate the effectiveness of SWaTEval by reproducing results from literature. Additionally, we provide new insights regarding SWAT by evaluating the impact of the similarity measures using SWaTEval. Our evaluation shows (1) that SWaTEval can be used to reproduce insights regarding the performance of similarity measures from the literature, and (2) that the choice of similarity measure impacts the quality of the state machine inference and, therefore, should be considered during the design of TTs for SWAT.

Contribution 10. *Development of a new similarity measure for web pages and use of* `SWaTEval` *to analyze the impact of the choice of similarity measures on the performance of state machine inference.*

`SWaTEval` is based on the Bachelor's thesis by Simon Zimmermann [*Zim21*], the Master's thesis by Felix Hofmann [*Hof21*], and the work by Johanna Kindler, Sara Joosten and Nikolay Penkov, all supervised during this doctoral work. It was presented at the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023), in collaboration with Nikolay Penkov and Mark Giraud. Below, we describe the approach and evaluation of `SWaTEval` with a focus on contextualizing the work, while the corresponding publication focuses on the more technical details of the implementation and the evaluation [*Bor23c*]. The source code is publicly available on GitHub[1].

MEMA MEMA focuses on the domain of stateless graybox fuzzing and is concerned with the impact of performance metrics on the relative evaluation of fuzzers. As an analytical basis, we identify the different dimensions of fuzzing evaluations.

Contribution 11. *Analysis and formulation of the dimensions of fuzzing evaluations.*

Then, we integrate additional metrics for fuzzing evaluations into the fuzzing benchmark tool MAGMA [Haz20], and assess how the selection of metrics influences the relative rating of fuzzers. To achieve this, we evaluate three fuzzers against five targets using six different metrics. Our experiments show that the choice of performance metrics impacts the relative ranking of fuzzers as thus has an impact on the evaluation of fuzzers. This insight is also supported by the study by Li et al. [Li21] published during the work on MEMA (see Section 8.4.1).

Contribution 12. *Integration of new metrics into an existing fuzzing benchmark framework and use of this framework to analyze the impact of these metrics.*

[1] https://github.com/SWaTEval

As we conduct an evaluation of the impact of performance metrics and base our experiments on the fuzzing benchmark MAGMA, we call our approach Metrics Evaluation for MAGMA (MEMA). MEMA is based on the Bachelor's thesis by Tom Blankefort [*Bla23*], which was supervised during this doctoral work.

During the course of this research, Li et al. introduced UNIFUZZ, an evaluation framework for fuzzers that also examines the impact of different performance metrics [Li21]. This indicates that the topic discussed in this section is considered relevant by the community. UNIFUZZ includes six categories of performance metrics and the authors evaluate 8 fuzzers against 20 targets, demonstrating that the choice of performance metrics influences the relative assessment of fuzzers. Their work exceeds the results presented in this chapter in terms of the number of evaluated metrics, fuzzers, and targets. However, the fuzzers and targets chosen by Li et al. only have a small intersection with only one fuzzer and one target: Only AFL and *sqlite3* are evaluated by Li et al. and the study presented in this chapter. With this, MEMA complements the results by Li et al., and we present our approach and findings in Section 8.4, without pursuing an additional publication of these results.

8.3 Stateful Web Application Testing

Even though first publications with respect to SWAT were published more than ten years ago (e.g. [Dou12]), a modular framework to evaluate the different approaches and design choices for SWAT has yet to be developed. Such a framework needs to provide a modular structure that allows the implementation of different SWAT approaches and includes a stateful target WA for performance analysis and evaluation. With `SWaTEval`, we design and implement such a framework with a corresponding target WA.

We conduct a twofold evaluation. On the one hand, we show that `SWaTEval` can be used to reproduce results from the study published by Yandrapally et al. [Yan20], suggesting that `SWaTEval` yields relevant results and insights. On the other hand, we utilize `SWaTEval` to conduct an evaluation of the impact

of one specific design choice in SWAT on the performance of the state machine inference. With this, we show how `SWaTEval` can help to provide new insights in the domain of SWAT.

Among the several challenges associated with SWAT, our evaluation focuses on one specific aspect to allow for a precise evaluation. For blackbox stateful testing, access to a state machine of the SuT is generally not assumed [Dou12]. Thus, the state machine needs to be inferred during testing, which is usually done based on the responses which the TT receives from the SuT (see Section 2.5). To infer the state machine from those responses, it is necessary to decide whether two responses are to be considered the same response. To achieve this, a similarity measure for the responses is necessary. In the domain of SWAT, the responses considered during state machine inference are web pages, and thus a similarity measurement for web pages is necessary. We focus our evaluation on the impact of this similarity measure on the quality of the state machine inference. We assess the quality of the state machine inference by the number of correctly identified states in the inferred state machine.

8.3.1 Concepts and Terminology

The following provides information on the concepts and terminology used in this section, with a particular focus on highlighting terms that are used inconsistently in literature.

Web Page We consider the SuT to be a WA with its full functionality, consisting of one or more *web pages*. For example, it could provide a home page, a login page, and a unique profile page for each user.

Endpoint An *endpoint* exposes functionality to the user and facilitates communication with the WA. In most cases, these endpoints can be directly mapped to the web pages provided by the WA. For example, a WA might offer the endpoint `/login.php`, which serves the login page when accessed.

Interaction We call a pair consisting of a request to an endpoint and the corresponding response an *interaction*.

State Similar to Doupé et al., we use the term *state* to describe the underlying internal state of a WA [Dou12]. For example, this state could differ before and after a successful user login. In this example, a WA would provide different functionality for an authenticated user, such as access to their profile, which would not be accessible for an unauthenticated user. Note that other works, such as those by Yandrapally et al. and Zhang et al., use a different terminology, considering each individual web page as a state of the WA [Yan20, Zha23b]. Their definition of a state corresponds to our definition of an endpoint.

State Machine Inference As we cannot assume access to the state machine of the SuT in blackbox testing, the TT needs to automatically infer the state machine based on its interactions with the SuT. Thus, *state machine inference* in this context refers to the automated blackbox inference of the state machine of a WA. This is done by interacting with the WA and by interpreting its responses. According to Doupé et al., state machine inference involves the following three main challenges [Dou12].

1 Clustering similar endpoints to avoid analysing duplicates, which do not contribute to the state machine inference

2 Detecting state changes to refine the state machine

3 Clustering similar states within the state machine to prevent state explosion

Each of these three tasks is addressed by different *detectors* in `SWaTEval` (see Section 8.3.3).

8.3.2 Related Work

The research presented in this section spans the domain of SWAT with an emphasis on similarity measures for web pages. The following provides an overview of related work in these areas.

Stateful Web Application Testing

SWAT has been a significant topic for some time and remains relevant. The foundational work by Doupé et al. introduced an approach that incorporates the underlying state machine of WAs into testing and fuzzing [Dou12]. However, their implementation is non-modular and has not been extensively compared with other state-aware approaches. As shown by Hassanshahi et al. [Has22], several approaches for SWAT and also for stateful crawling have emerged since then. Especially, Drakonakis et al. contributed to this topic by proposing a framework for SWAT, focusing specifically on authentication and authorization within SWAT [Dra20]. However, to the best of our knowledge, no existing work aims to design and implement a modular framework for evaluating different aspects of SWAT.

Several intentionally vulnerable WAs and benchmarks are available for evaluating WA testing, including WackoPicko [Dou10], JuiceShop[1], DVWA[2], and OWASP benchmark[3]. The discovery rate of the known vulnerabilities of these targets is used as a general performance metric of WVS (see e.g. Section 4.3.2 and [*Pfr19b*, *Bor20*]). However, to understand the impact of *stateful* testing methods, a more specific target is necessary. For example, it is required that the target exhibits different internal states and that the state machine is known and manageable in size.

Moreover, various works address different aspects of WA testing that complement our approach. First, the work presented in this dissertation in Chapter 4 and [*Bor20*] proposes an approach to enhance the performance of WVSs by injecting additional information into the WVS's requests via a proxy. We adopt a similar strategy to provide an interface for existing fuzzers in `SWaTEval` (see Section 8.3.3.1). Additionally, Li et al. propose an approach to test stateful web services based on a state machine derived from the web service's specification

[1] https://github.com/juice-shop/juice-shop
[2] https://github.com/digininja/DVWA
[3] https://owasp.org/www-project-benchmark/

given in the Web Services Description Language [Li18a], and several publications focus on stateful testing of REST APIs [Atl19, Yam21, Fel23]. In contrast, we construct the state machine from the blackbox communication with the WA itself. Other approaches focus on stateful testing of network protocol implementations [Aic21, Pfe22], whereas our work targets the user interface of the WA. While we implement a blackbox approach, other approaches focus on stateful graybox testing of WAs, e.g. by leveraging coverage and taint information which is only available in graybox testing [Gau21].

Similarity Measures

As mentioned in Section 8.3.1, measuring the similarity between two web pages is essential for automated state machine inference. Lin et al. propose a similarity measure based on the similarity of links on web pages [Lin06]. Other studies utilize the Levenshtein distance to compute the distance between web pages [Mes08, Pop16]. In contrast, Doupé et al. introduce a distance measure based on a prefix tree [Dou12]. Other works suggest measuring the similarity of web pages by analyzing their content, such as input fields [Lin17], or elements like buttons, anchors, and images [Ali19].

Oliver et al. present the locality sensitive hash Trend Micro Locality Sensitive Hash (TLSH) [Oli13]. With TLSH, small changes in the input result only in small changes in the corresponding hashes. This property is advantageous as it provides a means to represent data in a compressed format while keeping the distances between inputs similar, thus allowing for calculating similarity measures between the TLSH hashes. To this end, Oliver et al. define a distance measure on TLSH that approximates the Hamming distance, called the TLSH score. We choose TLSH and theTLSH score as one method for representing web pages and calculating similarities (see Section 8.3.3.3).

Yandrapally et al. conduct an extensive study on the performance of various similarity measures across multiple real-world WAs [Yan20]. This study includes, among others, the Levenshtein distance as well as TLSH. We use the results of their study to compare our own evaluation results against in Section 8.3.3.3.

8.3.3 Evaluation Framework

Our evaluation framework consists of a modular framework representing the state-aware TT, and a target WA with a known and manageable state machine. The following paragraphs describe the requirements, the derived design choices, and selected implementation details. Please refer to the corresponding publication for more details on the design and the implementation [*Bor23c*].

8.3.3.1 Test Tool Framework

The objective of the framework for the TT is to provide a modular structure in which different approaches for SWAT can be implemented and thus be evaluated individually.

Requirements

We formulate the following requirements for the TT (Tool Requirements (TRs)).

TR1 *Modularity:* The different parts necessary for SWAT are divided into modules such that they can easily be modified or replaced.

Without a modular structure, simple modifications and replacements of certain parts of the testing approach would not be possible. However, this is crucial for evaluations of individual design choices in order to understand their impact on the performance of the fuzzer and the state machine inference.

TR2 *Interaction:* The crawling and fuzzing modules can work in an interlaced fashion.

Both Hassanshahi et al. and Doupé et al. identify the need to allow for an interaction between the crawler and the fuzzer [Has22, Dou12]. With this interaction, the crawler can first identify endpoints which are then used as targets for the fuzzer. During the fuzzing of these endpoints, potentially again new endpoints are identified. These new endpoints can then again be used by

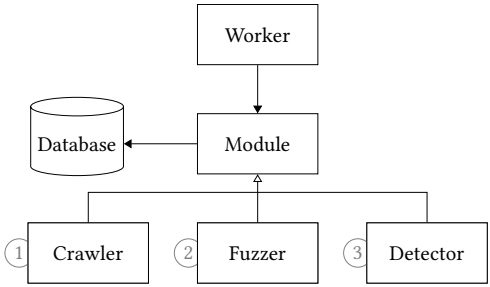


Figure 8.1: Overview of the general design of the TT framework of SWaTEval, providing a modular structure for crawlers, fuzzers, and detectors. During execution, the modules are called in the order indicated by the gray numbers [*Bor23c*].

the crawler to identify even more endpoints. Since this interaction is proposed and used in literature, SWaTEval should support this interlaced approach to be able to implement existing approaches.

TR3 Traceability: All generated data can be traced back and can be reproduced.

In order to understand the impact and behavior of the different modules, it is necessary to trace and reproduce the data they generated. To achieve reproducible data, it is necessary to implement deterministic behavior.

General Design

Figure 8.1 gives an overview of the TT framework's architecture. We design the framework to consist of several modules, which can be exchanged and extended easily. These modules can be classified in the following three categories: (1) crawlers, responsible for state selection and state traversal, (2) fuzzers, which generate payloads and analyze the SuT's responses for possible vulnerabilities, and (3) detectors, tasked with pattern detection and information inference. Each module has read and write access to the centralized database. This centralized access facilitates seamless interaction between the modules and enables efficient information sharing. Especially, this allows for an information sharing between the crawlers and the fuzzers.

Table 8.2: Modules implemented for `SWaTEval`.

Module	Source
Crawler (Basic)	Adapted from [Dou12]
Fuzzer (Dummy)	Newly presented in [*Bor23c*]
Fuzzer (External)	Adapted from [*Bor20*]
Detector (Clustering)	Newly presented in [*Bor23c*]

During execution, the modules are called in a predefined order, to allow for an interleaved execution and information sharing between the different modules. This predefined order is visualized by the gray numbers 1 - 3 in Figure 8.1. First, all crawler modules are run, followed by the fuzzing modules, and then the detector modules. After this, the loop is restarted and the crawler modules start again.

In the following, we detail the general task of the different modules and present the concrete implementations of these modules that we provide with `SWaTEval`. An overview of the implemented modules is given in Table 8.2.

Crawlers

The crawlers' primary objective is to explore the WA and to identify new endpoints for fuzzing. Information gathered by the crawlers is stored in the database and later utilized by the fuzzers to conduct their tests and by the detectors to construct the state machine. This state machine then helps the crawler to determine which states should be explored next, by saving information on which states have been explored already. Moreover, the state machine saves information on the known endpoints that are accessible for each state. For example, an endpoint `/profile.php` could only be accessible if the user authenticated first and thus the WA is in the respective state.

For our experiments, we implement a basic crawler which employs a straightforward approach to crawling and is based on the work presented by Doupé et al. [Dou12]. The basic crawler selects the first available state of the WA that has not been explored yet, and subsequently visits all accessible endpoints within that state. To maintain consistency, the WA is reset to the selected

state before each iteration of crawling. Note that there are more sophisticated approaches to crawling (see e.g. [Has22]) as well as to state selection for testing (see Section 7.6), which could be integrated into SWaTEval in the future.

Fuzzers

The primary objective of the fuzzers is to generate payloads to test the WA and to reveal vulnerabilities. In our design, fuzzers also play a crucial role in revealing new states within the WA by providing various inputs to the WA that potentially could tigger a state transition. Like the crawlers, information generated by the fuzzers is stored in the centralized database. For our implementation of SWaTEval, we provide two different fuzzers: (1) a dummy fuzzer, tailored for our target WA, and (2) an interface for external fuzzers.

The dummy fuzzer mocks the functionality of a fuzzer, but incorporates knowledge on the target WA of SWaTEval. To this end, it sends input to the WA which is known to tigger state transitions in the current state. Additionally, the fuzzer also sends other requests that will explicitly not trigger a state transition, to mimic the behavior of a real fuzzer. With this dummy fuzzer, efficient evaluations of the other modules of SWaTEval are possible, since the existing states in the WA can be visited easily.

In addition to the dummy fuzzer, we provide an integration for external fuzzing tools. This integration is based on the proxy-based approach to WA testing as presented in Section 4.3 and [*Bor20*]. For SWaTEval, we start the external fuzzer from within SWaTEval and intercept its traffic using mitmproxy[1]. This approach enables us to capture and analyze the responses generated by the SuT for state machine inference. Moreover, we can inject state data, such as cookies and headers, to the requests made by the external fuzzer to be able to influence the state of the SuT.

[1] https://mitmproxy.org/

Detectors

The detectors build the base for the automated state machine inference. To this end, they analyze the responses received by the SuT, considering the requests that were made before receiving these responses. With this, the detectors can use this information to infer the state machine of the SuT based on the following intuition. If a request A lead to response X before sending some fuzzing input, but the same request A leads to a response Y after the fuzzing input, it is likely that the fuzzing input caused a state transition of the SuT. It is generally assumed that the SuT presents the same response for a certain request if no state change took place. Note that this approach needs to use a similarity measure for the responses in order to be able to decide whether a response X should be considered as equal, or similar, to a response Y. This general approach can be implemented in various different ways. Especially, the question of how to represent the information found by the crawlers and fuzzers needs to be answered. Doupé et al. represent the information in so-called *page link vectors* [Dou12], while we present a hash-based approach in the following.

For our implementation of `SWaTEval`'s modules, we propose a new approach to the state machine inference and content representation necessary for the detector modules. The following describes the challenges and the approach on a high level, for more details refer to the corresponding publication [*Bor23c*]. According to Doupé et al., a blackbox state machine inference needs to solve the following challenges: (1) decide which endpoints are to be considered the same, (2) decide whether a state transition happened, and (3) decide which states of the state machine are to be considered to be the same [Dou12].

We solve all these challenges by hashing the necessary information using the locality sensitive hash TLSH (see Section 8.3.2, and performing a clustering step afterward. Note that TLSH preserves the locality of the input values, and thus similar inputs lead to similar hashes. For the first challenge, we hash and cluster the endpoints in the `EndpointDetector`. Then, we consider those endpoints in the same cluster to be the same endpoint. For the second challenge (`StateChangeDetector`), we generate hashes of the interactions and perform a clustering on all interactions that share the same request. If all interactions

based on the same request are clustered in the same cluster, we assume that no state change happened. In the opposite case, where one or more interactions are clustered differently, we assume that a state change happened. Based on this observation, we can infer the state machine as proposed by Doupé et al. [Dou12], which follows the intuition presented above. We approach the third challenge similar to the first challenge by hashing the state information, and calculating clusters on these hashes (`StateDetector`). Then, the states that are clustered in one cluster are considered to be the same state and are merged in the state machine. For more details on the detectors, refer to the corresponding publication [*Bor23c*].

8.3.3.2 Target

The target WA of `SWaTEval` is designed to provide a testing target for stateful TTs, with a special focus on the state machine inference. It can be utilized either together with the evaluation framework or as a standalone target. This target incorporates various challenges specific to stateful TTs while maintaining a manageable and comprehensible state machine. This allows for manual in-depth analyses of the TTs' performance.

Requirements

We define the following requirements for the target WA (Web Application Requirements (WRs)).

WR1 *Similar Pages:* The target includes web pages that are similar but not equal, which should be recognized and treated as the same web pages by the TTs.

Interacting with the same web page several times does not contribute to the state machine inference and thus, a stateful TT needs to decide which web pages should be treated as the same web page [Dou12]. Thus, the target WA should include web pages that are to be treated as the same web page to analyze whether the corresponding detector of the TT identifies them correctly.

WR2 *Different Pages:* The target includes web pages that are significantly different from one another and should be recognized and treated as different web pages by the TTs.

Complementary to *WR1*, we also require web pages that are significantly different and should be classified accordingly by the respective detector.

WR3 *Statefulness:* The target provides the possibility to change its state through external requests.

Since the target should be used for evaluating stateful TTs, it needs to include an internal state machine which can be inferred by a blackbox TT. For this, it is necessary that the some requests sent by the TT trigger state transitions of the WA's state machine.

WR4 *Complexity:* The state machine of the target remains manageable and comprehensible to humans.

To allow an in-depth analysis of the performance of a stateful TT and the impact of individual design choices, we require the state machine of the target to be manageable in size and comprehensible to humans.

Design

Similar to the approach taken by Doupé et al., we implement the target as a server-side WA [Dou12]. In this setup, the DOM elements for each request are generated in the back-end and are included in the response of the WA each time. We implement the following functionality for the target.

User Login The target WA includes login functionality, providing a regular user and an admin user.

Chained Links The target provides an endpoint that accepts a number as an input parameter and the resulting web page contains a link to the same endpoint with an incremented number as parameter. This endpoint specifically challenges stateful TT by providing an endless chain of links. If the detectors do not detect that these web pages are to be treated as the same, the crawler

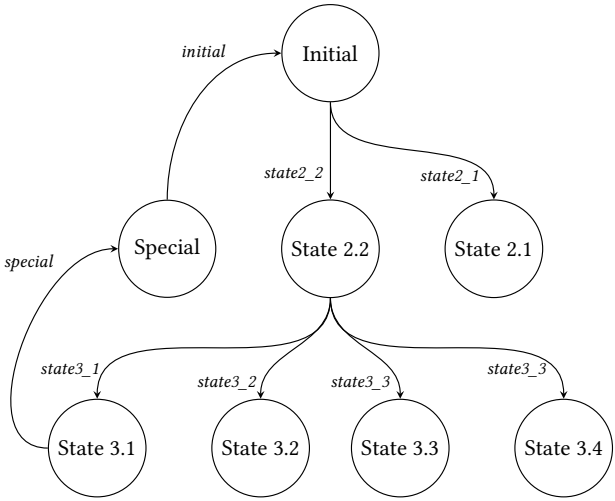


Figure 8.2: Visual representation of the state machine of the target, including the keywords that are necessary to perform a state transition [*Bor23c*]. Note that the state machine includes a loop which poses a challenge to the detectors, since this loop and possible duplicate states need to be identified.

will be stuck in an endless chain of requests. This implementation is inspired by the approach of Doupé et al. who include an endless calendar in their vulnerable WA WackoPicko [Dou10].

Pages with Links and Content The target encompasses web pages that include links or content that can either be dynamically generated or be constant. With this, the target provides means to evaluate the performance of the various detectors.

State Machine Next to the aforementioned web pages, the target includes an underlying state machine. State transitions of this state machine can be triggered by certain keywords which need to be provided to the WA. The states, state transitions, and the required keywords are shown in Figure 8.2. Note that the state machine includes a transition that resets the WA to the initial state (keyword *initial*). With this, it poses a special challenge to the detectors which

need to identify that the states on this loop are to be considered the same. If this would not be recognized, the inferred state machine would include several copies of the actual state machine, chained after one another.

8.3.3.3 Evaluation

Our evaluation of SWaTEval, including the TT framework and the target, consists of a qualitative evaluation and a quantitative evaluation. The qualitative evaluation shows that our design and implementation of the TT framework and the target fulfill the respective requirements defined in Sections 8.3.3.1 and 8.3.3.2. The qualitative evaluation shows that SWaTEval can be used to reproduce the results from large studies in literature. We show this by reproducing the results presented by Yandrapally et al. with respect to the performance of different similarity measures for web pages [Yan20].

Qualitative Evaluation

As the first step of our evaluation, we analyze how SWaTEval meets the requirements for a SWAT evaluation framework as formulated in Sections 8.3.3.1 and 8.3.3.2.

TR1 Modularity By dividing the framework into modules with the same functionality (crawler, fuzzer, detector) and thus encapsulating their individual approaches, we hide complexity and create abstract workflows. With this, we ensure that SWaTEval fulfills *TR1*.

TR2 Interaction We facilitate interaction between the modules of SWaTEval by establishing a database as the core point of information exchange. The modules of SWaTEval are run in a sequential order, executing their logic only when the conditions specified in their configurations align with the current status of the framework. For example, a fuzzer will generate its fuzzing requests in the current state only if the crawler and the detectors have marked the currently selected state of the state machine as fully explored.

TR3 Traceability We make the influence and behavior of different modules and their functionality visible by separating the generated data in a centralized database. This allows for a centralized analysis of the current status of `SWaTEval` and its modules. This separation also allows for manual data editing during runtime, facilitating experiments with edge cases and a deeper analysis of the modules' behavior. The database contains all relevant information on the behavior of the modules and its content can be stored and used for documentation or analysis purposes, thereby fulfilling *TR3*.

WR1 Similar Pages The target WA includes web pages with similar content, which should be classified as the same web page. This holds, for example, for the web pages at `/views/const-content/const-links/random-page`. For more details on the web page, refer to the corresponding publication [*Bor23c*] or the source code on GitHub[1].

WR2 Different Pages Additionally, the target features web pages with distinctly different content which should be classified accordingly, such as the pages at `/views/dynamic-content/const-links/random-page`.

WR3 Statefulness The target WA implements mechanisms to change the underlying state of the WA through (1) user login, and (2) an artificial state machine controlled by different keywords, as illustrated in Figure 8.2.

WR4 Complexity The state machine of the target WA consists of eight states with easily traceable transitions and is thus suitable to be manually analyzed.

> In summary, our qualitative evaluation shows that `SWaTEval`, consisting of a TT framework and a target WA, fulfills the requirements for a SWAT evaluation framework as formulated in Sections 8.3.3.1 and 8.3.3.2.

[1] https://github.com/SWaTEval/evaluation-target

Quantitative Evaluation

The objective of our quantitative evaluation is to determine if SWaTEval can be used to replicate results from literature. Specifically, we compare the performance of various similarity measures based on SWaTEval's target with the evaluation results presented by Yandrapally et al. [Yan20]. Note that Yandrapally et al. use different terminology, referring to each individual web page as a *state*. In SWaTEval, this corresponds to an *endpoint*. Consequently, their evaluation of similarity measures for states can be directly compared to our evaluation with respect to similarity measures for endpoints.

We conduct the following experiments to evaluate the suitability of SWaTEval. First, we use a selection of three similarity measures to cluster the web pages Yandrapally et al. used for their evaluation [Yan20]. Then, we use the same similarity measures to cluster the web pages of SWaTEval's target. Thus, this allows us to gain insights into the performance of different similarity measures both with respect to the targets by Yandrapally et al. and with respect to SWaTEval's target.

Similarity Measures We use three different similarity measures for our experiments, as shown in Table 8.3: (1) Euclidean distance, (2) TLSH score, and (3) Levenshtein distance. For the first two similarity measures, we choose the locality sensitive hash TLSH as representation for the web pages. Thus, we calculate the TLSH for each considered web page and then calculate the similarity of the hashes based on the respective similarity measure. For comparison, we include a similarity measure based on the DOM representation of the web page, using the Levenshtein distance as similarity measure. After

Table 8.3: Content representation and similarity measures used for our quantitative evaluation.

Name	Representation	Similarity Measure
TLSH score	TLSH (String)	TLSH score
Euclidean	TLSH (Integer)	Euclidean distance
Levenshtein	Response body (String)	Levenshtein distance

calculating the respective similarities, we use this information to cluster the web pages using DBSCAN. Refer to the publication on SWaTEval for more information on our configuration of DBSCAN [*Bor23c*].

To leverage the Euclidean distance, we first need to convert the text-based hash into an Integer representation. Since the TLSH algorithm always outputs a hash consisting of 72 characters, we can directly interpret this as an Integer vector by converting characters to their corresponding ASCII Integer.

Oliver et al., who introduce TLSH, also propose a similarity measure for TLSH hashes known as the TLSH score. This score approximates the Hamming distance and is calculated by comparing every two bits. Furthermore, it penalizes large differences with disproportionately higher scores [Oli13].

The Levenshtein distance, being a text-based distance, can be directly calculated on the DOM of the web page. This choice for this similarity measure is based on the observation that this approach is regularly used in literature (see Section 8.3.2 and e.g. [Mes08, Pop16, Yan20]) and thus is included as a baseline.

Results Figure 8.3 illustrates the accuracy of clustering the web pages based on the different similarity measures for the dataset by Yandrapally et al. and SWaTEval's target. The x-axis shows the similarity measures, while the y-axis shows the achieved accuracy values. For example, the clustering based on the Levenshtein distance achieves an accuracy of 0.69 on the dataset by Yandrapally et al., and an accuracy of 0.89 on SWaTEval's target. For both datasets, clustering based on the TLSH score achieves the relatively lowest accuracy, while the Euclidean distance achieves the highest accuracy.

Discussion

Given that Yandrapally et al.'s data comprise more diverse WAs, it is expected that the clustering would show generally lower accuracy on their dataset compared to SWaTEval's target. Our results support this expectation.

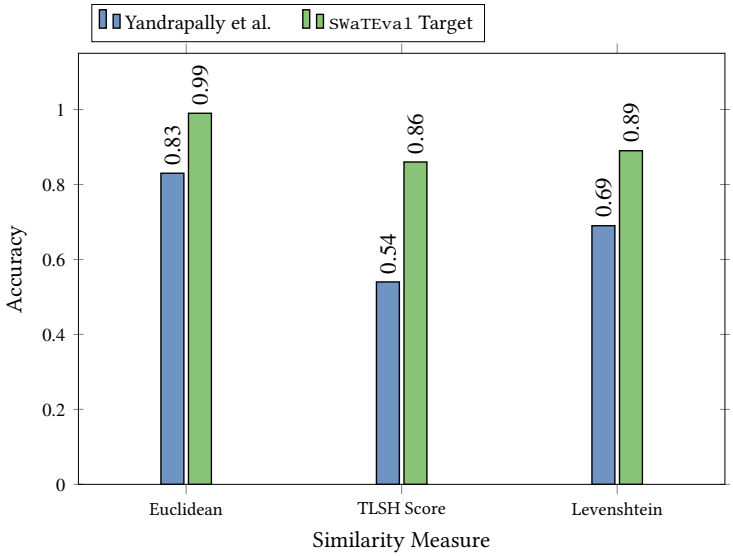


Figure 8.3: Accuracy of clustering web pages from the datatset by Yandrapally et al. and SWaTEval's target using different similarity measures (based on [*Bor23c*]). The relative performance of the three similarity measures is the same.

Moreover, it shows that, despite differences in absolute values, our results exhibit the same trends and relative performances for the different similarity measures. Thus, our results show that our artificial target WA produces results comparable to those of Yandrapally et al.'s study. This suggests that our target WA effectively incorporates the essential features of real-world WAs.

8.3.4 Similarity Measures

After having presented and evaluated SWaTEval, this section utilizes SWaTEval to evaluate the impact of choosing different similarity measures on the quality of the inferred state machine. With this, we aim to provide new insights regarding design choices for stateful WA TTs, and also to present how SWaTEval can be applied to a concrete use case. The following shows an aggregated view on the experiments and results, while the corresponding publication presents a more detailed view [*Bor23c*].

8.3.4.1 Experimental Setting

We again consider the three similarity measures already introduced in Table 8.3 in the previous section: Euclidean, TLSH score, and Levenshtein. For our experiments, we configure the three detectors that are included in `SWaTEval`, `EndpointDetector`, `StateChangeDetector`, and `StateDetector` (see Section 8.3.3.1) with either the Euclidean distance or the TLSH score. For the `StateChangeDetector`, we additionally add a configuration using the Levenshtein distance on the DOM. Thus, we use $2 \cdot 2 \cdot 3 = 12$ configurations in our experiments. We run each of these configurations 20 times to account for potential influences of the TLSH calculations (see [*Bor23c*]).

For each of the runs, we calculate the count of correctly identified states, the count of identified endpoints, and the count of conducted interactions. Out of these performance metrics, the count of correctly identified states in the final state machine is considered to be the most important metric.

For this implementation, we utilize the modularity of `SWaTEval`'s design and implementation to implement easily interchangeable variants of the three detectors.

8.3.4.2 Results

Figure 8.4 shows an aggregated view on our results with respect to the number of correctly identified states. While we run all 12 configurations described above and present those results in the corresponding publication [*Bor23c*], we now focus on the configurations which use the same similarity measure for all detectors. This allows us provide a broader view on the experiments and to highlight the impact and context of our results. Figure 8.4 presents the number of correctly identified states of the state machine for the configuration in which all detectors use the Euclidean distance, and in which all detectors use the TLSH score. The bar labeled *Levenshtein* represents the configuration in which the `StateChangeDetector` uses the Levenshtein distance. For this configuration, changing the similarity measure used for the other two detectors does not change the final result. The whiskers show the standard deviation

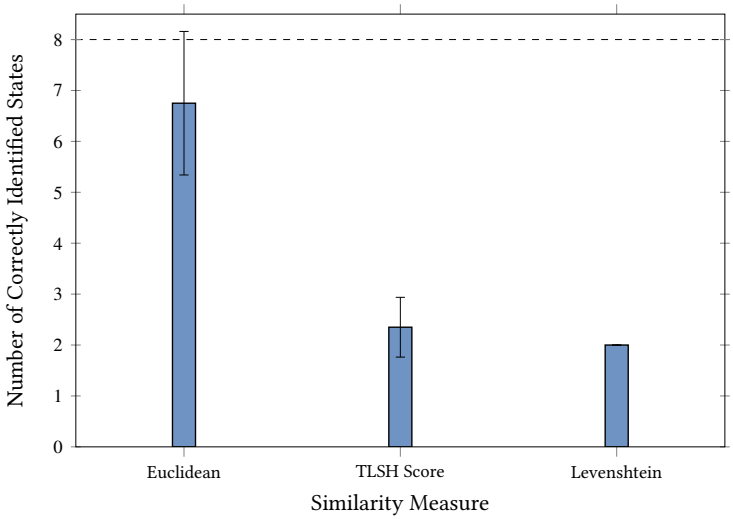


Figure 8.4: Mean number of correctly identified states for the different similarity measures (based on [*Bor23c*]). Using the Euclidean distance for all detectors leads to the highest number but also results in the highest standard deviation, visualized by the whiskers.

calculated on the 20 runs of the respective configuration, and the dashed line represents the number of states in the correct state machine. It shows that the configuration using all Euclidean distance results in a mean number of correctly identified states of 6.75, while TLSH score achieves 2.35, and using the Levenshtein distance for the `StateChangeDetector` leads to 2.00 correctly identified states. Moreover, it is apparent that the standard deviation of the results is the highest for the Euclidean distance, while being smaller for the TLSH score and zero for the Levenshtein distance.

In addition to the final count of correctly identified states, we also report the impact of the similarity measure for the different detectors in Table 8.4. We calculate this impact value as the difference between the minimum and the maximum mean number of correctly identified states for all configurations concerning this specific detector. Based on this value, the choice of the similarity measure is more important for the `StateChangeDetector` and the `StateDetector` than for the `EndpointDetector`.

Table 8.4: Impact of the choice regarding similarity measures for the different detectors (based on [*Bor23c*]). The similarity measures have the smallest impact on the `EndpointDetector`.

Detector	Impact
`StateChangeDetector`	2.30
`StateDetector`	2.46
`EndpointDetector`	0.08

8.3.4.3 Discussion of Results

The results presented above highlight the following three key findings.

1 The Euclidean similarity measure results in the highest number of correctly identified states.

2 The use of TLSH increases the standard deviation of the results.

3 The choice of similarity measure has more impact on the `StateDetector` and on the `StateChangeDetector` than on the `EndpointDetector`.

The observation that the Euclidean distance leads to better results than the TLSH score is contra-intuitive, since both approaches use TLSH as basis and one would expect that the distance measure especially developed for this hash would lead to a better performance. Moreover, the literature suggests that distance measures based on the Manhattan distance are more suitable for high dimensional data than the Euclidean distance [Agg01]. Based on the findings by Aggarwal et al. and given our 72-dimensional data, it is expected that TLSH score, approximating the Hamming distance, to perform better than the Euclidean distance. This discrepancy could be due to the differences in the data used by Aggarwal et al. and the data we used in our experiments. Aggarwal et al. use synthetic data drawn from a normal distribution and data from the UCI machine learning repository [Agg01], whereas our study uses TLSH as input for both similarity measure and clustering. Analyzing classification performance of the similarity measures on various data types could be a valuable direction for future research, thought it is beyond the scope of this doctoral work.

The second finding is that the usage of TLSH increases the standard deviation of the results. This is expected due to effects of the padding necessary for TLSH [*Bor23c*].

The third finding relates to the impact of the choice of similarity measures. It shows that the choice for the two detectors directly concerned with the states of the state machine has a greater impact on the number of correctly identified states. This supports the expected behavior, since these two detectors directly influence the quality of the state machine. The `EndpointDetector`, clustering similar endpoints, has a relatively high impact on the runtime and efficiency of the testing compared to its impact on the quality of the state machine.

8.3.5 Discussion

The following contextualizes the results and findings with respect to `SWaTEval` in Section 8.3.5.1, discusses known limitations in Section 8.3.5.2, and outlines possible directions for future work in Section 8.3.5.2. Note that the WA provided by BC_{ex} does not include an internal state, since it only presents status information on the bus coupler, without providing any means for user interaction. Thus, we did not apply `SWaTEval` to test BC_{ex}.

8.3.5.1 Implications

With `SWaTEval`, we publish the first modular evaluation framework with a special focus on SWAT and state machine inference. It allows for evaluation of individual design choices and approaches in this domain and builds the basis for future research. To further support future research building upon `SWaTEval`, we published the source code[1].

Our evaluation showed that `SWaTEval` fulfills the requirements for an evaluation framework for SWAT as defined in Sections 8.3.3.1 and 8.3.3.2. Moreover, we successfully utilized `SWaTEval` to gain new insights with respect to the

[1] https://github.com/SWaTEval

impact of similarity measures on the state inference performance. On the one hand, this shows how SWaTEval can be used for evaluations. On the other hand, it shows that the choice of similarity measures actually impacts the quality of the state machine inference, showing that this decision should be taken deliberately in TTs for stateful WAs.

8.3.5.2 Limitations

The current implementation of the target WA included with SWaTEval only incorporates HTML-based web pages and does not feature JavaScript-based content. While this still covers a relevant portion of the WAs used in the industrial domain, it would be interesting to extend the target WA in the future. Moreover, SWaTEval currently focuses on sever-side WA and does not support modern web technologies such as progressive web apps. This could also be an additional starting point to enhance the target WA.

8.3.5.3 Future Work

Based on the newly presented evaluation framework SWaTEval, several research directions could be explored.

On the one hand, SWaTEval could be used to analyze additional approaches to and design choices for state machine inference. For example, it could be used to evaluate the effectiveness of different state inference algorithms, such as the ones proposed by Raffelt et al. and Vaandrager et al. [Raf05, Vaa22]. Moreover, SWaTEval could be used to study how the quality of the inferred state machine affects fuzzer performance. Another possible direction for future research based on SWaTEval would be to optimize the feature selection for the representation of endpoints, interactions, and states.

On the other hand, the existing implementation of SWaTEval could be enhanced by adding new detector modules. For instance, anomaly detection capabilities of autoencoders, such as presented by Mirsky et al. [Mir18], could serve as a basis for new detectors. Furthermore, the basic state selection approach used in SWaTEval could be enhanced by using a more sophisticated

approach, such as the ones presented by Liu et al. [Liu22], or the ones presented in Section 7.6 [*Bor23a*]. With this, an evaluation of the impact of the state selection algorithm on a fuzzer's performance in the domain of WA testing could be conducted.

8.4 Stateless Fuzzing

As fuzzers are complex non-deterministic systems, solidly evaluating their performance and comparing them to other fuzzers is a challenging task. In this section, we examine which approaches exist for the evaluation of fuzzers and which dimensions of fuzzing evaluations are considered by literature. Based on this analysis, we identify the performance metrics used to assess the fuzzers as dimensions that received less attention in the past. Moreover, recent literature suggests that the relative assessment of fuzzers can vary depending on the metrics chosen [Fio22, Böh22]. Thus, we aim to evaluate the impact of both established and new fuzzing metrics on the performance evaluation of fuzzers. To this end, we extend the existing fuzzing framework MAGMA [Haz20] with new fuzzing metrics and evaluate how the selection of metrics influences the assessment of the fuzzers.

8.4.1 Related Work

Recently, numerous studies on fuzzing evaluations have been published, indicating the high relevance of this topic. The following provides an overview of the work in this area, including studies published after we started working on MEMA.

Probably the most recognized work in the domain of fuzzer evaluations is the one presented by Klees et al. [Kle18]. The authors formulate requirements for a sound fuzzing evaluation, such as a sufficient number of runs and an appropriate timeout for these runs. Then, they analyze the fuzzer evaluations conducted by 32 research papers and find that none of these evaluations

fulfills all of the requirements. Based on their findings, they formulate recommendations for future fuzzing evaluations. Klees et al. analyse the following dimensions of fuzzing: statistical sound comparisons, seed selection, timeouts, performance metrics, and targets. With respect to the performance metrics, the authors state that fuzzer evaluations should measure the performance of a fuzzer in terms of known bugs, while code coverage metrics could be used as a secondary metric. Their experiments show that heuristics for found bugs, such as code coverage or stack hashes, tend to over-count the number of bugs and thus are not a reliable heuristic. However, the authors do not discuss the impact or relevance of other performance metrics. In a recent study, Schloegel et al. [Sch24] analyze 150 fuzzing evaluations to investigate to what extend existing guidelines are implemented, including those by Klees et al. [Kle18]. They show that the guidelines are not fully implemented by the considered publications, especially with respect to statistical tests. Moreover, they revise existing fuzzing evaluation recommendations and formulate new recommendations. With respect to performance metrics, Schloegel et al. recommend to evaluate code coverage and bug-finding capabilities [Sch24]. Kim et al. present a study on evaluating *directed* fuzzers, highlighting requirements specific to this subdomain of fuzzing [Kim24].

Li et al. present UNIFUZZ, a fuzzing benchmark that focuses on implementing several performance metrics to analyze their impact on the fuzzer evaluation [Li21]. With this, their work has the same objective as MEMA. UNIFUZZ encompasses performance metrics from the following categories: quantity of unique bugs, quality of bugs, speed of finding bugs, stability of finding bugs, coverage, and overhead. MEMA also includes performance metrics with respect to the quantity and quality of bugs, the stability of bugs, and the overhead. However, the implementation and experiments presented by Li et al. exceed the ones conducted in the work presented in this section. Nevertheless, the fuzzers and targets chosen by Li et al. only have a small intersection with only one fuzzer and one target: Only AFL and *sqlite3* are evaluated by Li et al. and the study presented in this chapter. Consequently, MEMA accompanies and supports the findings by Li et al., since both studies show that the choice of metrics indeed influences the relative assessment of fuzzers and thus are to be considered a relevant factor in fuzzing evaluations.

The study presented by Böhme et al. is also concerned with performance metrics in fuzzing evaluations [Böh22]. The authors analyze the correlation between code coverage, often used as proxy metric, and the actual bug finding capabilities of fuzzers. Their experiments show a strong correlation between the coverage achieved and the number of bugs a fuzzer finds. However, they also show that even though this correlation exists, the relative ranking of fuzzer performance (*agreement*) shows only a moderate consensus between the two metrics. This work again highlights the importance and the impact of performance metrics in fuzzing evaluations.

Paaßen et al. analyze the impact of the stochastic properties of fuzzing on fuzzing evaluations [Paa21]. Specifically, they demonstrate the relevance of statistical tests for the evaluation of fuzzers. Additionally, the authors present a framework for fuzzing evaluations which provides statistical analysis methods for the evaluations. We include their findings on statistical tests for fuzzing evaluations in our analysis of the dimensions of fuzzing evaluations (see Section 8.4.2).

Wolff et al. examine how the results of a fuzzer evaluation conducted on a certain benchmark generalize [Wol22]. They show that the relative assessment of fuzzers depends on the benchmark and the seeds used for the evaluation. Moreover, they propose a methodology to explain the impact a specific evaluation setting has on the evaluation's results. Our analysis of the dimensions of fuzzing evaluations incorporates the findings of Wolff et al.

Gavrilov et al. and Gopinath et al. [Gav20, Gop22] propose new approaches to fuzzing evaluations. Gavrilov et al. propose to use several versions of a SuT to analyze whether the fuzzers are able to reveal the changes inbetween the different versions of the SuT [Gav20]. The authors argument that version updates, similar to bugs, are changes in the SuT that should be revealed by a fuzzer. They conduct preliminary experiments showing that the results of their evaluation is consistent with evaluations based on bug-based metrics.

Gopinath et al. propose to apply the concept of *mutation analysis* known from traditional software testing to fuzzing evaluations [Gop22]. Mutation analysis evaluates the performance of a TT by introducing small changes to the SuT and

then analyzing whether the evaluated TT is able to detect the change in the SuT. With this, it is based on a similar idea as the approach presented by Gavrilov et al. [Gav20], but does not rely on several real-world versions of the SuT. Instead, the mutations to the SuT are injected automatically. Gopinath et al. claim that using mutation analysis will improve fuzzing evaluations, especially due to the flexibility in injecting arbitrary faults into a SuT. Nevertheless, they also identify several challenges that need to be solved before mutation analysis can be used for fuzzing evaluations, such as a reduction of the computational cost [Gop22].

8.4.2 Analysis

As a basis for our implementation and experiments, we conduct an analysis of existing approaches to fuzzing evaluations. This analysis is driven by the following two questions:

1 What methodologies are utilized for the systematic evaluation of fuzzers?

2 Which dimensions are considered for the evaluation of fuzzers and what choices are considered for these dimensions?

With answering the first question, we aim to give an overview of the current state-of-the-art with respect to fuzzing evaluations. The second question aims to analyze how these fuzzing evaluations are conducted and which dimensions of the evaluation are considered. For example, one dimension for fuzzer evaluations could be the seeds that are chosen for the fuzzer. Moreover, we aim to analyze which choices for these dimensions are considered in literature. For example, for the dimension of the fuzzer's seed, one could imagine configurations such as (1) no seeds, (2) a set of valid and invalid seeds, (3) a minimized set of seeds.

Note that we concentrate on the evaluation of stateless graybox fuzzers, since these fuzzers did receive more attention in literature compared to blackbox fuzzers [Böh21, Mal23]. Nevertheless, we recognize that a similar analysis for the evaluation of stateful fuzzers and blackbox fuzzers would also be beneficial (see also Section 8.4.4.3).

8.4.2.1 Methodologies for Fuzzing Evaluations

Our analysis on the methodologies for fuzzing evaluations builds upon survey papers on fuzzing and system evaluations [Kle18, Böh22, Fio22, Haz20, Wol22, Kou19], as well as papers presenting new fuzzing approaches that include evaluations of this new fuzzer (such as [Fio20, Pha20, Pfr18]). Note that the latter set of papers is by no means exhaustive, as a significant number of fuzzing papers have been published recently [Sch24].

Benchmarking Systems

In general, evaluating or benchmarking systems should fulfill the following requirements: (1) completeness, (2) relevance, (3) correctness, and (IV) reproducibility [Kou19].

Completeness The evaluation of a system needs to consider all claims about the system. It also includes highlighting the limitations of the system.

Relevance The results presented in an evaluation need to be relevant to the objectives of the evaluation. For example, the evaluation setting should simulate a real environment such that the evaluation results are transferable to real-world deployments of the system.

Correctness The evaluation needs to be conducted correctly and the results need to be analyzed accurately. In the evaluation of fuzzers, this mainly concerns the calculation of statistical metrics as well as their solid interpretation.

Reproducibility For results to be verified, evaluations need to be reproducible. To ensure this, the evaluation setup needs to be fully described and important configurations need to be specified. Moreover, the code used for the evaluation could be published to ensure full reproducibility.

Evaluating Fuzzers

Meeting the requirements discussed above is essential for well-founded evaluations. However, there are some additional challenges to consider when evaluating fuzzers. On the one hand, fuzzing is a stochastic process and thus the results of a fuzzer vary from execution to execution. Thus, to obtain reliable results, fuzzers need to be executed multiple times, which increases the resources and time required, limits reproducibility, and makes correct analyses more demanding [Kle18]. On the other hand, the fuzzing process can be quite non-transparent. Many fuzzers use feedback obtained from the SuT during fuzzing and generate their inputs based on this feedback. These developments over time are difficult to comprehend from the outside. Furthermore, a variety of design decisions for the fuzzer and the evaluation influence the evaluation outcome [Paa21, Wol22].

Generally, fuzzers are empirically evaluated by comparing the new fuzzer with one or more reference fuzzers [Kle18, Sch24]. For this comparison, several fuzzing campaigns are run against a set of targets, and several metrics are collected during and after these campaigns (see Section 8.4.2.2 for more details). Theoretical evaluation approaches are used to infer and justify new fuzzing techniques (e.g. [Böh20]), but are only used to complement empirical evaluations. Some papers take the approach to compare various attributes and characteristics of different fuzzers (e.g. [Pfr18]). This provides an overview of the fuzzers, while not delivering a complete evaluation of the fuzzers' performance.

8.4.2.2 Dimensions of Fuzzing Evaluations

Our analysis of survey papers and fuzzing papers shows that empirical evaluations of fuzzers are the main approach for evaluating fuzzers. Nevertheless, as discussed in Section 8.4.1, we recognize that there are some studies proposing different approaches [Gop22, Gav20]. Thus, we detail the different dimensions of empirical fuzzing evaluations in the following. The dimensions and descriptions are mainly based on the work by Klees et al. [Kle18], Wolff et al. [Wol22], Li et al. [Li21], Paaßen et al. [Paa21], Li et al. [Li21], and Arcuri et al. [Arc11].

Reference Fuzzer The chosen reference fuzzer needs to be relevant so that insights can be gained from the comparison between the fuzzer being evaluated and the reference fuzzer. For this purpose, the reference fuzzer should be state-of-the-art. Furthermore, the comparison between the fuzzer being evaluated and the reference fuzzer should be fair, meaning that the fuzzers should be based on similar use cases and architectures. For example, comparing a whitebox fuzzer and a blackbox fuzzer can be misleading since these fuzzers are designed for different use cases. Although direct comparisons between different evaluations are challenging due to differing system specifications, version deviations, and numerous evaluation parameters [Li21], choosing a widely-used reference fuzzer such as AFL++ [Fio20] can facilitate comparisons.

Seeds Literature shows that the initial seed corpus has a significant impact on the evaluation results [Kle18, Paa21]. Existing seed corpora vary in terms of the number and size of the seeds, the average execution time of the seeds, and the code coverage achieved by the seeds. The seeds can be valid or invalid inputs, randomly generated, manually selected, or generated through previous fuzzing campaigns. Since all these variations can influence the evaluation, the seed files used in an evaluation should be specified. To mitigate the influence of individual characteristics of the seeds, diverse seed sets should be used. Moreover, fuzzers can be evaluated based on an empty initial corpus.

Timeout The timeout of a fuzzing campaign describes the duration of a fuzzing test run. Since fuzzers usually cannot fully test a program, a termination criterion needs to be established. In most cases, a fixed time duration is used for this purpose. The timeout affects the results of a fuzzing campaign, and,

consequently, the evaluation of a fuzzer [Kle18, Paa21]. A balance needs to be struck between the effort and the completeness of the evaluation. A common choice for the fuzzing timeout are 24 hours [Kle18].

Trials Since fuzzing usually includes non-deterministic behavior, the performance of different runs of the same fuzzer can result in different performance [Paa21]. Thus, a large number of fuzzing runs should be conducted to account for this non-deterministic behavior and to obtain statistically significant results. Similar to the timeout, a balance between the effort and the completeness needs to be struck. It is recommended to conduct at least 30 runs for each combination of fuzzer and target [Kle18].

Statistical Tests To be able to derive reliable interpretations, statistical tests should be used to assess the significance of the results [Arc11, Paa21]. The Mann-Whitney U test [Man47] and the Vargha-Delaney $\hat{A}_{12}$ measure [Var00] are suitable for this purpose [Kle18]. While the Mann-Whitney U test helps to decide whether the performance difference between two fuzzers is statistically significant, the $\hat{A}_{12}$ measure quantifies how big the performance difference is. In any case, statistical measures such as the mean, median, and standard deviation should be reported. This enables others to understand the results of the evaluation and form their own interpretations.

Performance Metrics While Klees et al. note that a fuzzing evaluation should report the number of bugs identified by fuzzing as a performance metric, most scientific fuzzing evaluations use code coverage as a proxy metric [Böh22]. Li et al. state that most papers in literature use either the number of unique crashes, the number of unique bugs, or the code coverage [Li21]. Muench et al. focus on fuzzing embedded devices and show that measuring only the crashes of a SuT has the potential to not correctly identify and measure memory corruption bugs triggered by a fuzzer [Mue18].

Benchmarks for fuzzing usually also report different metrics. For example, MAGMA [Haz20] reports the number of bugs a fuzzer reached and the number of bugs a fuzzer triggered. A bug is considered as *reached* when the faulty line of code is executed, and *triggered* when the fault condition is satisfied. FuzzBench [Met21] reports code coverage and bug coverage, including a

differential coverage, showing the regions that were uniquely covered by one fuzzer when compared to the other fuzzers in the evaluation. However, before the start of the work presented in this section, no study analyzed the actual impact of the performance metric on the relative assessment of fuzzers.

Targets The choice of the fuzzers' targets, or *benchmark suite* [Kle18], used during the evaluation highly influences the performance of the evaluated fuzzers [Wol22]. If one fuzzer performs better on a specific target program, this does not necessarily indicate that the fuzzer will perform better in general [Kle18, Wol22]. To obtain a representative overview of a fuzzer's performance, it therefore needs to be evaluated on a diverse set of targets. For a relevant and realistic evaluation, fuzzers should be evaluated using real-world targets. However, the large amount of possible targets makes comparisons between evaluations difficult. If a target program is used for evaluations at two different points in time, bugs found in one evaluation might have been fixed in the meantime, or the structure of the program might have changed. Version differences also significantly hinder the reproducibility of evaluations [Haz20]. To approach this issue, several fuzzing benchmark suites have been proposed, such as FuzzBench [Met21], MAGMA [Haz20], UNIFUZZ [Li21], and ProFuzzBench [Nat21], which is specifically designed for stateful fuzzing.

8.4.2.3 Analysis Summary

Our literature review and analysis leads to the following conclusions.

1 The empirical evaluation of fuzzers is currently the most prevalent approach to fuzzing evaluations.

2 There are slight differences in the dimensions presented and investigated by literature, but no contradictions.

3 No study analyses the impact of the performance metrics on the relative ranking of fuzzers.

4 Recent work continues to support the existing recommendations for fuzzing evaluations, while highlighting that these recommendations are not implemented satisfactorily [Sch24].

8.4.3 Experiments

Based on our analysis, we identify the need to evaluate the impact of different performance metrics on the relative assessment of fuzzers. Thus, we extend the existing fuzzing benchmark MAGMA [Haz20] by adding additional metrics and an automated evaluation and comparison of these metrics. We choose MAGMA as basis, since it includes real-world targets with real bugs, and already provides the number of reached and triggered bugs as performance metrics. In addition to these metrics that are already included in MAGMA, we implement a means to calculate *stability*, *rare bugs*, *memory usage*, and *CPU usage*, which are described in the following. All these metrics are especially relevant for the practical application of fuzzers. Note that Li et al., whose work was published during the work on MEMA, consider similar types of performance metrics [Li21]. In their evaluation, the authors also examine the stability, the rareness of bugs, and the memory consumption of the fuzzers.

8.4.3.1 Metrics

We implement the following additional metrics for MAGMA. Firstly, we aggregate and contextualize the number of found bugs across multiple fuzzers and multiple runs measure rare errors and stability. Secondly, we collect additional system information to measure memory consumption and the CPU load.

Stability Unlike scientific fuzzing evaluations, which typically involve several runs of a fuzzer to account for its randomness, practical applications of fuzzers usually only include one or a few number of runs. Therefore, it is crucial for a fuzzer to demonstrate stable performance across multiple runs against the same target. We include the stability of a fuzzer's performance, as measured by the variance in the number of the bugs found by the fuzzer, in MEMA. Note that Wang et al. use the term *stability* of a fuzzer to describe performance consistency across different targets [Wan21c], and AFL++ the term with respect to the SuT [Heu24].

Rare Bugs Similar to our evaluations concerning WVSs [*Bor18*, *Pfr19b*], we aim to include a metric to compare the bugs found by the fuzzers. Specifically, we count the number of bugs that were found only by the fuzzer under consideration and not by any of the other fuzzers. To maintain consistent terminology, we refer to these bugs as *rare bugs*, in line with the terminology of the paper by Li et al. [Li21], which was presented during our work on MEMA.

Memory Consumption and CPU load Along with CPU load, memory consumption is a common metric used to evaluate the performance of software systems [Kou19]. In fuzzing evaluations, these metrics can quantify the resources required by a fuzzer during its runs, providing additional insights into the fuzzer's performance.

8.4.3.2 Implementation

Figure 8.5 shows an overview of our implementation of MEMA. In the figure, the rounded rectangles represent inputs or outputs of the process, while the rectangles show processing steps. As discussed above, we build our metric collection approach upon the fuzzing benchmarking framework MAGMA [Haz20]. An evaluation using MEMA starts with a configuration detailing the setting such as the fuzzers and the targets to be used during the evaluation. Then, the first run of MAGMA is started accordingly, while the parallel measurement of memory consumption and CPU load is conducted by MEMA. As a result, we receive the number of bugs found by the fuzzer as reported by MAGMA as well as the collected metrics concerning CPU load and memory consumption. Then,

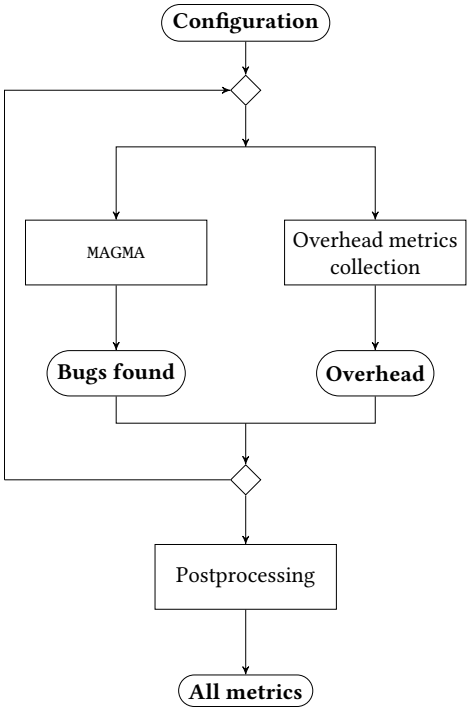


Figure 8.5: Overview of the implementation of MEMA, building upon the evaluation framework MAGMA [Haz20]. MEMA collects additional metrics during the runs of MAGMA, and performs a postprocessing of MAGMA's results to calculate additional performance metrics. This is conducted for each combination of fuzzer and target specified in the configuration.

this loop is repeated until all runs that were specified in the configuration are finalized. Based on the results and measurements from all these runs, MEMA performs several postprocessing steps to calculate the rare bugs found by the fuzzers and the stability of the fuzzers. Moreover, MEMA reports statistical values and the results of a Mann-Whitney U test and the Vargha-Delaney $\hat{A}_{12}$ measure. Finally, we receive a textual representation (JSON) and visual representation of the collected performance metrics of the fuzzers.

As MAGMA runs the fuzzers and targets in docker containers, MEMA measures the CPU load and memory consumption every ten seconds using the `docker stats` command. MEMA reports the mean values for each fuzzer for each target individually, and for each fuzzer aggregated over all targets. Moreover, MEMA reports the maximum value for the memory consumption that was observed during the fuzzing runs for each fuzzer and target.

8.4.3.3 Experimental Setting

We base our experiments on the fuzzers and targets included in MAGMA. Specifically, we use AFL [Zal16], `Entropic` [Böh20], and `FairFuzz` [Lem18] as fuzzers and run them against the following targets: *libpng*, *libtiff*, *libxml2*, *poppler*, and *sqlite3*. `FairFuzz` is an AFL-based fuzzer, while `Entropic` is based on `LibFuzzer` [LLV24]. For the fuzzing campaigns, we use the seed files given by MAGMA[1]. We repeat each combination of fuzzer and target for 24 times and run the campaigns for 24 hours. Note that we repeat each run only 24 times due to time and resource restrictions. As hardware, we use a Intel Xeon E5-1650 v3 processor with 3.5 GHz, six cores and two threads per core. The system uses Ubuntu 22.04 and has 64 gigabytes of RAM.

Our experiments aim to provide new insights into the following research question:

> *What influence does the choice of metrics have on the relative rating of the fuzzers?*

8.4.3.4 Results

In the following, we present the results that we collected during our experiments. For some of the results, we aggregate the information in order to give a condensed view on the results and the performance of the fuzzers, focussing on the contextualization of the results. More details can be found in the Bachelor's thesis by Tom Blankefort [*Bla23*].

[1] https://github.com/HexHive/magma (commit hash: 75d1ae7)

Reached and Triggered Bugs

First, we report the number of reached and triggered bugs as measured by MAGMA. MAGMA classifies a bug as *reached* if the corresponding line of code was executed during a fuzzing run, and classifies it as *triggered* if the specific fault condition was met [Haz20]. Table 8.5 shows the statistical metrics of these values, including the mean, the median, the standard deviation, and the coefficient of variance. AFL and FairFuzz reach a mean of 41.79 and 41.38 bugs, respectively, while Entropic reaches a mean of 39.21 bugs. The standard deviation of reached bugs is 3.51 for AFL, 3.17 for FairFuzz, and 1.14 for Entropic.

To allow for a more detailed comparison of the fuzzers' performance, conducted several statistical tests and report their results in Table 8.6. As suggested by literature [Kle18], we consider a significance level of $\alpha = 0.05$. With this, the differences in the number of bugs between AFL and FairFuzz are not significant, both for reached bugs (0.289447) and triggered bugs (0.237786), . With respect to the reached bugs, the differences found are significant both in

Table 8.5: Statistical metrics of the bugs reached (reach.) and triggered (trigg.) by the fuzzers over 24 runs: mean, median, standard deviation, and Coefficient of Variance (CV). The results are aggregated over the five targets.

Fuzzer	Mean		Median		Std. Dev.		CV	
	reach.	trigg.	reach.	trigg.	reach.	trigg.	reach.	trigg.
AFL	41.79	8.42	43	8	3.51	1.41	0.0823	0.1642
Entropic	39.21	8.00	39	8	1.98	1.14	0.0494	0.1398
FairFuzz	41.38	8.92	42	9	3.17	1.28	0.0751	0.1408

Table 8.6: Results of the statistical analyses based on the number of bugs reached and triggered by the fuzzers. We report p-values calculated using the Mann-Whitney U test, as well as the $\hat{A}_{12}$ values.

Fuzzers	p-value		$\hat{A}_{12}$	
	reached	triggered	reached	triggered
AFL vs. FairFuzz	0.289447	0.237786	0.5877	0.4028
AFL vs. Entropic	0.000005	0.284279	0.8394	0.5885
FairFuzz vs. Entropic	0.000537	0.016882	0.7899	0.6962

Table 8.7: Bugs that have only been found by one fuzzer (*rare* bugs), referenced by the bug identifiers as provided by MAGMA.

Fuzzer	Rare Bugs	
	reached	triggered
AFL	-	TIF001
Entropic	SQL013, SQL012, SQL006, PDF018	PDF018, PDF021, SQL013, XML003, SQL012, PNG006
FairFuzz	SQL020	SQL020

the comparison of AFL to Entropic (0.000005) and FairFuzz to Entropic (0.000537). However, concerning the triggered bugs, the difference between FairFuzz and Entropic is significant (0.016882), but the difference between AFL and Entropic is not (0.284279).

The Vargha-Delaney $\hat{A}_{12}$ values describes the probability that a value from one group is greater than a value from another group. $\hat{A}_{12}$ values in the interval [0.56,0.64) are interpreted as small effect sizes, values in [0.64,0.71) are interpreted as medium effect size, and values in [0.71,1] as large effect sizes [Var00]. With respect to the reached bugs, the effect size between AFL and FairFuzz is small (0.5877), while it is large between AFL and Entropic (0.8394), and FairFuzz and Entropic (0.7899). This is in line with the findings from the Mann-Whitney U test.

Rare Bugs

Based on the reached and triggered bugs as reported by MAGMA, MEMA calculates the rare bugs found by each fuzzer. *Rare* bugs are those bugs that are only found by one of the considered fuzzers. Table 8.7 shows the rare errors that have been found by the fuzzers, referenced by the MAGMA bug identifiers. It shows that AFL does not reach a rare bug, while Entropic reaches 4 rare bugs, and FairFuzz one. AFL and FairFuzz trigger one rare bug, while Entropic triggers 6.

Stability

We measure the stability of a fuzzer by calculating the Coefficient of Variance (CV) of the triggered bugs (see Table 8.5). With this, a lower absolute value for the CV and thus for the stability is to be interpreted as a better performance. AFL achieves a stability of 0.1642, FairFuzz achieves 0.1408, and Entropic 0.1398.

Memory Consumption

Figure 8.6 shows the memory consumption of the fuzzers for each of the targets in Mebibyte (MiB). The bars show the mean value, calculated over the 24 runs, while the additional circle marks the maximum memory consumption observed over the 24 runs. For example, for the target *libxml2*, AFL consumes a mean of 119.94 MiB, Entropic uses 394.87 MiB, and FairFuzz 190.47 MiB. Averaged over the five targets, AFL needs 85.07 MiB, Entropic uses 248.58 MiB, and FairFuzz 264.34 MiB. Entropic shows the highest maximum numbers for all five targets. For *libxml2*, Entropic exhibits a maximal memory consumption of 651.6 MiB.

CPU Load

MEMA also reports the CPU load observed during the fuzzers' runs. Figure 8.7 shows the mean CPU load for the fuzzers for each of the targets. The results show varying results for the CPU load. AFL exhibits the lowest CPU load for *libpng*, *libxml2*, and *poppler*, but the highest for *libtiff* (98.73%). Entropic and FairFuzz show similar CPU loads for most targets. However, for *sqlite3*, FairFuzz reaches a mean CPU load of 98.17%, while Entropic uses only 84.98%.

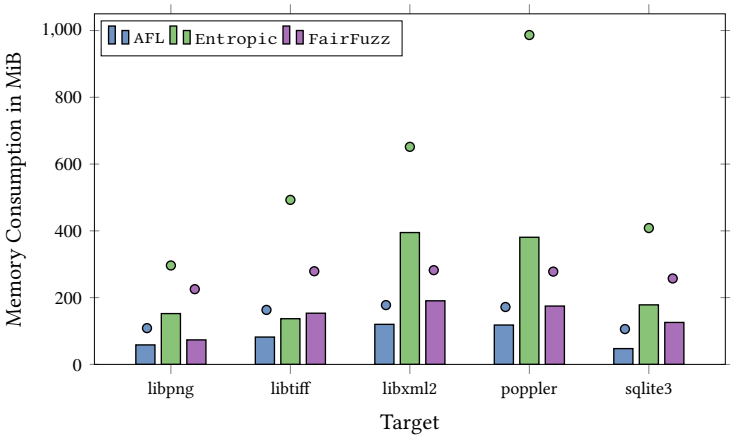


Figure 8.6: Memory consumption of the fuzzers for the five targets. The bars show the mean memory consumption in MiB over the 24 runs, and the additional marks show the maximum value that was observed during the runs.

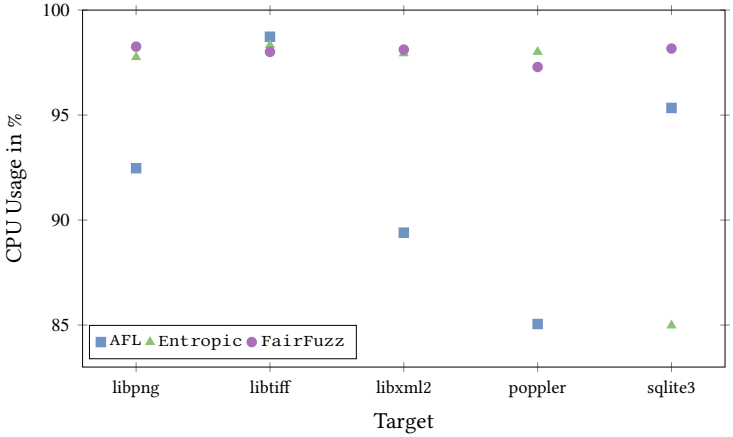


Figure 8.7: Mean CPU load shown by the fuzzers for the five targets. AFL shows the lowest CPU load for three of the targets.

Table 8.8: Overview of the aggregated performance metrics for the different fuzzers. *Rank* shows the relative ranking of each fuzzer for the corresponding performance metric. While AFL achieves the first rank for most of the metrics, it does not outperform the other two fuzzers in all metrics.

Metric	AFL		Entropic		FairFuzz	
	Value	Rank	Value	Rank	Value	Rank
Triggered Bugs	202	2	192	3	214	1
Reached Bugs	1003	1	941	3	993	2
Rare Bugs	1	2	6	1	1	2
Stability	0.1642	3	0.1398	1	0.1408	2
Memory Consumption	85 MiB	1	249 MiB	3	144 MiB	2
CPU Load	92.20%	1	95.40%	2	97.97%	3

8.4.3.5 Discussion of Results

We aggregate and assess the values for the different metrics as provided by MEMA in Table 8.8. The results indicate that the relative assessment of fuzzers varies depending on the considered metric. For each fuzzer, a metric exists for which it achieves rank 1, 2, or 3. For example, FairFuzz triggers the highest number of bugs (214), Entropic triggers the highest number of rare bugs (6), and AFL reaches the highest number of bugs (1003).

When considering only the number of reached bugs as the primary metric, AFL is rated better than Entropic, which a significant difference in performance. However, when the number of triggered bugs is considered, the difference between AFL and Entropic is no longer significant. Moreover, while FairFuzz reaches and triggers a higher number of bugs than Entropic, Entropic reveals a higher number of rare bugs. An evaluation that does not consider rare bugs as metric might overlook this aspect in the relative comparison between FairFuzz and Entropic. Thus, specifically the choice between the number of triggered, reached, and rare bugs, as well as the consideration of statistic calculations, affects the evaluation of the fuzzers in this context.

For both memory consumption and CPU load, AFL achieves smaller values than Entropic and FairFuzz. On average, Entropic consumes nearly three times more memory than AFL, with the mean maximum memory usage being

3.5 times higher. Therefore, in an evaluation deeming the used resources of the fuzzers an important metric, AFL could be favored over FairFuzz and Entropic. However, an alternative interpretation of these results could be that AFL does not fully utilize the available resources, and thus is less effective.

Overall, the experiments conducted with MEMA demonstrate that the choice of metrics impacts the relative ranking of fuzzers. The selection of the primary metric as well as the weighting of metrics influences the ranking. Taking multiple metrics into account allows for a more comprehensive and nuanced assessment of fuzzers than reducing the evaluation to a single metric and ranking.

8.4.4 Discussion

With MEMA, we analyzed existing approaches to fuzzer evaluations, included additional metrics to the fuzzer evaluation framework MAGMA, and evaluated the impact of the metrics on the relative ranking of fuzzers. The following discusses MEMA's implications in Section 8.4.4.1, details limitations in Section 8.4.4.2, and shows possible directions for future work in Section 8.4.4.3

8.4.4.1 Implications

The main finding of MEMA is that the choice and weighting of metrics for fuzzing evaluations influences the relative ranking of fuzzers. With this, it highlights the need for generally accepted metrics and a clear and justified choice of metrics in fuzzer evaluations. During the work on MEMA, Li et al. published a study targeting the same research question as MEMA [Li21], resulting in the same key finding. The authors approached the same research question, chose similar additional performance metrics (see also Section 8.4.1), and present a comprehensive evaluation of eight fuzzers using 20 targets. Out of these fuzzers and targets, one of each overlaps with the experiments chosen for MEMA: AFL and *sqlite3*. Thus, even with using different fuzzers and different targets, Li et al. come to the same results and conclusions as presented in this

chapter. With this, the study presented in this chapter complements the results presented by Li et al. [Li21]. However, their work exceeds the experiments presented in this chapter in terms of the evaluated metrics, fuzzers, and targets.

8.4.4.2 Limitations

Most fuzzing evaluations choose the code coverage as a metric to assess the performance of the fuzzers [Böh22]. Since MAGMA does not support this metric and as an integration of this metric would mean a significant amount of work (see Section 8.4.4.3), we refrained from including this metric in our experiments. However, the experiments presented in this section could be extended by including the code coverage as additional metric to assess the fuzzers. Moreover, one could extend the experiments by including more of the targets fuzzers supported by MAGMA. In total, MAGMA supports nine targets and 17 fuzzers at the time of this writing.

8.4.4.3 Future Work

The authors of MAGMA deliberately excluded coverage from the metrics supported by MAGMA, since they claim that the code coverage is only an approximation of the actual performance of the fuzzers [Haz]. However, for an evaluation targeting the impact of different performance metrics, including the coverage would be beneficial. To achieve this, it would be necessary to use the inputs generated by the fuzzers to measure the coverage achieved by these inputs after the actual runs of the fuzzers. To this end, one could create additional versions of the docker containers provided by MAGMA such that they include the tools necessary for code coverage measurement (e.g. `llvm-cov`[1]). With this, one could re-use the compilation instructions used, and thus achieve the same setup for the coverage analysis as has been used for fuzzing before.

[1] https://llvm.org/docs/CommandGuide/llvm-cov.html

The analyses and experiments presented in this chapter focus on the evaluation of stateless graybox fuzzers. On top of this, it would be interesting to further investigate and analyse existing approaches to evaluating blackbox fuzzers and stateful fuzzers. Especially for the latter, additional metrics such as the final state coverage could be evaluated [Guo24].

8.5 Summary

This chapter presented the contributions of this doctoral work to the domain of evaluating TTs. On the one hand, SWaTEval contributes to evaluating TTs for SWAT by providing an evaluation framework and an evaluation target. We show that SWaTEval can be utilized to yield reliable results by showing that we can use it to reproduce results from extensive studies in literature. Moreover, we utilized SWaTEval to evaluate whether the choice of similarity measures used in the automatic state inference influences the performance of the state machine inference. Our evaluation shows that (1) the Euclidean similarity measure leads to the highest number of correctly identified states, that (2) the use of TLSH increases the standard deviation of the results, and that (3) the choice of similarity measure is of different importance for the different parts of automatic state machine inference.

On the other hand, MEMA contributes to evaluating stateless graybox fuzzers by giving an overview of the different dimensions of fuzzing evaluations, and by adding additional metrics to the fuzzer evaluation framework MAGMA. Using MEMA, we evaluate the impact of choosing different performance metrics on the final ranking of fuzzers. Our experiments show that the choice of performance metrics indeed impacts the ranking and thus should be considered for future fuzzing evaluations, based on the concrete setting and requirements of the evaluation.

9 Discussion

This section consolidates the various approaches presented in the previous chapters by addressing the overarching research questions formulated in Section 1.3. It also discusses the impact and the limitations of this doctoral work. While this section emphasizes the broader contextualization of the results, each preceding chapter includes a section that discusses the respective approach in more detail.

9.1 Research Questions

The following addresses and discusses the research questions formulated in Section 1.3 and details how the newly presented approaches of this doctoral work contribute to the research questions.

Research Question 1. *What sources of information are available for blackbox security testing?*

While blackbox testing assumes no access to internal data of the System under Test (SuT), external and behavioral information on the SuT is still accessible during testing. We present the results of our analysis of approaches from the literature that utilize additional information in blackbox testing in Chapter 3.

Our analysis reveals that the information sources utilized in blackbox testing can be categorized along two dimensions: (1) time of accessibility, and (2) temporal variability. Regarding time of accessibility, we differentiate between information accessible prior to testing, and information that becomes accessible only during testing. Examples for information sources that are accessible before

testing are known vulnerabilities and previously recorded network traffic. One information source that is only accessible during testing is the behavior of the SuT in response to certain test cases, such as crashes or the SuT's responses.

The second dimension, temporal variability, differentiates between static and dynamic information sources. Static information, which remains constant over time, includes information on the communication endpoints that a SuT provides. In contrast, dynamic information, potentially changing over time, encompasses the network traffic which can be observed during testing.

In Chapter 3, we also classify the newly introduced approaches according to the information sources they utilize. It shows that the newly presented approaches do not rely on new information sources but instead utilize known information sources in new ways. Chapter 10 explores additional information sources that could potentially be leveraged in future research.

Research Question 2. *How can information that is accessible prior to the actual test be used to improve a blackbox security test?*

With respect to information sources that are accessible before testing, our analysis shows that existing Test Tools (TTs) utilize information on known vulnerabilities, expert knowledge in terms of heuristics for test case generation, and network traffic (see Chapter 3). In this doctoral work, we present three new means to utilize this information.

Amongst other information, `HitM` uses additional information on the SuT known prior to the test, such as login details, to improve the performance of Web Vulnerability Scanners (WVSs) transparently. Our experiments show that utilizing `HitM` increases the number of true positive reports by the WVSs and an increased Uniform Resource Locator (URL) coverage. For $\texttt{BC}_{\text{ex}}$, `HitM` revealed one previously unknown vulnerability [CVE18].

With `ClusterCrash`, we present an approach to represent the knowledge from existing vulnerabilities systematically using Vulnerability Anti-Patterns (VAPs) (see Chapter 5 and [*Bor22*]). From these VAPs, we derive blackbox fuzzing scripts leveraging the knowledge on known vulnerabilities, which have been revealed by whitebox and graybox testing. With this, we make the

information on known vulnerabilities directly accessible for blackbox TTs. We utilize the derived fuzzing scripts to test eight Operational Technology (OT) components and report eleven findings, showing that our scripts can successfully utilize the information from known vulnerabilities.

Additionally, we present `Palpebratum`, which infers a protocol-agnostic Hidden Markov Model (HMM) from network traffic to determine the interestingness of a test case based on the inferred model (see Section 7.5). As a basis, we additionally propose an approach to preprocess the network traffic to make it accessible for the HMM, called `NeDaP`. We train the HMM on user data including 140 traces of File Transfer Protocol (FTP) traffic. During fuzzing, this HMM is queried based on the observed network traffic. This information is then used to determine the interestingness of a test case. Our experiments show that these models can be utilized by a blackbox fuzzer to generate test cases that are more efficient than those generated by a traditional blackbox fuzzer. Nevertheless, the final coverage of such a fuzzer is outperformed by a true blackbox fuzzer. One possible explanation for this is the overhead introduced by the HMMs which results in a reduced number of test cases that can be sent during a fuzzing campaign.

Research Question 3. *How can models of the SuT be derived from information sources during a blackbox security test?*

In this doctoral work, we present three new approaches to derive a model from information sources accessible before and during testing. Details on these approaches, including a detailed discussion of related work, can be found in Chapter 7. First, we present `Smevolution`, an approach to combine evolutionary fuzzing with a Machine Learning (ML) model which is trained during the course of the fuzzing campaign. This model utilizes information on the SuT's crashes and approximates a function mapping a test case to the services of the SuT that are expected to crash in response to this test case. We implement this approach with three different ML models, namely a Decision Tree (DT), a Neural Network (NN), and an Support Vector Machine (SVM). Our experiments indicate that the DT and the NN are able to learn relevant

information on the behavior of the SuT which can be leveraged to improve the performance of a blackbox fuzzer in terms of found vulnerabilities, while the fuzzer based on the SVM does not perform better than the baselines.

Second, as discussed with respect to the previous research question, `Palpebratum` leverages network traffic accessible before and during testing to train an HMM which is then used to approximate the behavior of the SuT.

Third, with `StateBandit`, we evaluate the applicability of Reinforcement Learning (RL) to the state selection problem in fuzzing. For our experiments, we apply different Multi-armed Bandit (MaB) agents to the state selection problem of graybox fuzzing. These agents interact with the SuT and build an implicit model of the SuT during this interaction. Our experiments show that the fuzzers based on these agents are not able to outperform the current state-of-the-art. This observation is in line with the results by Li et al., which show that none of the state selection approaches the authors considered has lead to a distinguishable fuzzer performance [Li21]. We chose to conduct our experiments with respect to graybox fuzzing to evaluate whether MaB agents are able to improve fuzzing based on graybox information. However, given the results, we decided not to transfer this approach to blackbox testing, in which the information that is accessible for the agents is even more limited.

Research Question 4. *How can test approaches be evaluated and compared to competing approaches?*

While each evaluation of a TT should follow the generic requirements for evaluating systems by providing complete, relevant, correct, and reproducible results [Kou19], the specific evaluation approach depends on the domain of the TT. For this doctoral work, we focus on blackbox Stateful Web Application Testing (SWAT) and stateless graybox fuzzing.

Literature includes several evaluations of general WVSs (see Section 4.3.2), but evaluating *stateful* WVS has not received as much attention. Thus, we provide the modular framework for evaluating stateful WVSs `SWaTEval`, including a target Web Application (WA) with a known state machine [*Bor23c*]. With this, we allow for detailed analyses of the impact of different testing approaches

and design choices. We show this by (1) implementing different approaches for the distance measure necessary for the automatic state machine inference, and by (2) evaluating their impact. Our results imply that the choice of this measure indeed has an impact on the quality of the inferred state machine, and thus should be considered while designing stateful WVSs.

With respect to stateless graybox fuzzing, we analyze existing evaluation approaches and guidelines, and describe the different dimensions of a fuzzing evaluation in Section 8.4.2. Our analysis suggests that the guidelines presented by Klees et al. [Kle18] are considered to be the state-of-the-art for fuzzing evaluations. This was also confirmed by the more recent work presented by Schloegel et al. [Sch24], which was published after we conducted our analysis. Nevertheless, some authors suggest considering other evaluation approaches based on mutation testing [Gav20, Gop22].

Moreover, our analysis shows that the impact of the choice of performance metrics on the relative assessment of fuzzers has not received much attention in the past. However, more recent literature suggests that the choice of metrics may have a significant impact on the relative assessment [Fio22]. Thus, we extend the fuzzing framework MAGMA [Haz20] with new performance metrics and evaluate their impact, calling our approach MEMA. We show that the choice and weighting of performance metrics influence the relative ranking of fuzzers by evaluating three fuzzers using five targets. With this result, our experiments supplement the findings by Li et al., who conducted a similar study after we started our work [Li21]. The authors used mostly different fuzzers and targets, with only one fuzzer and one target being the same for their evaluation and our evaluation. Nevertheless, Li et al. come to the same conclusions as our work, which supports the general claim that the choice of performance metrics is important for the evaluation of stateless fuzzers.

9.2 Implications

All approaches developed, implemented, and evaluated in this doctoral work contribute to the objective to allow for effective testing of OT components. The experiments including actual OT components and artificial vulnerabilities show that the proposed approaches `HitM`, `Smevolution`, `ClusterCrash` are able to improve blackbox testing. `Palpebratum` and `StateBandit` further explore how ML techniques could be used to utilize the information accessible in blackbox and graybox fuzzing. Moreover, we contributed to the research on evaluating TTs by providing `SWaTEval`, a framework to evaluate stateful WVS, and by evaluating the impact of the choice of performance metrics on fuzzer evaluations using `MEMA`.

9.2.1 Reported Vulnerabilities

Several of the approaches presented in this doctoral work have been executed against OT components, including BC_{ex}, to assess their practical applicability. During these experiments and tests, we followed a responsible disclosure policy and disclosed all our findings to the respective manufacturers.

BC_{ex} Findings

The PROFINET bus coupler BC_{ex} is used as a running example throughout this doctoral work. By applying several of the newly presented approaches, we revealed anomalies and vulnerabilities in BC_{ex} which are summarized in Table 9.1. In total, the approaches revealed one configuration error and six Denial of Service (DoS) vulnerabilities within BC_{ex}. Of these DoS vulnerabilities, four are to be considered as critical based on their Common Vulnerability Scoring System (CVSS) score. Details on the vulnerabilities are provided in the sections referenced in Table 9.1. These findings illustrate how the different approaches can be applied to an OT component, and which vulnerabilities can potentially be found within an OT component.

Additional Findings

In addition to BC_{ex}, we tested several other OT components during the experiments as presented in this doctoral work. Table 9.2 aggregates the vulnerabilities of other OT components that have been confirmed and published during this doctoral work, while additionally showing anonymized data on those vulnerabilities that have been confirmed, but not yet published by the manufacturer. The following gives an overview of those vulnerabilities that have been confirmed by the affected manufacturer and that received a Common Vulnerabilities and Exposures (CVE) identifier.

During this doctoral work, nine vulnerabilities were reported and confirmed by the corresponding manufacturer. For example, we reported a DoS vulnerability of a safety controller manufactured by SICK. This vulnerability was confirmed and closed by the manufacturer. We confirmed this fix by re-testing the updated version of the controller's software.

All except one of the vulnerabilities shown in Table 9.2 have been fixed by the respective manufacturer, although not all have been assigned a CVE identifier yet. The remaining unaddressed vulnerability is the DoS identified in the industrial I/O device. This attack is only feasible shortly after the startup of the affected OT component. According to the manufacturer, this scenario represents a highly unlikely attack vector, which is why they opted not to address this vulnerability.

Table 9.1: Vulnerabilities and anomalies of BC_{ex} revealed by the novel approaches presented in this doctoral work.

CVSS	Approach	Vulnerability	Reference
0.0	`HitM`	Missing security headers	Section 4.4
5.3	`HitM`	DoS via TCP flood	Section 4.4
5.3	`Smevolution`	DoS via IPv4 (source address)	Section 7.7
7.5	`HitM`	DoS via parallel TCP connections	Section 4.4
7.5	`HitM`	DoS via HTTP [CVE18]	Section 4.4
7.5	`ClusterCrash`	DoS via IPv4 (length field)	Section 5.5

Table 9.2: Vulnerabilities revealed during this doctoral work in OT components other than BC_{ex} that have been confirmed by the respective manufacturer. A dash (-) in the CVE column implies that the respective vulnerability was not yet assigned a CVE ID. Most vulnerabilities are DoS vulnerabilities. CVSS scores indicating a high severity are highlighted in red, while scores indicating a medium severity are highlighted in orange.

CVSS	CVE	Tested OT component	Vulnerability
7.5	CVE-2019-14753	Safety controller by SICK	DoS via UDP
5.3	CVE-2021-21003	Industrial switch by Phoenix Contact	DoS via TCP
7.4	CVE-2021-21004	Industrial switch by Phoenix Contact	Cross-site scripting via LLDP
7.5	CVE-2021-21005	Industrial switch by Phoenix Contact	DoS via TCP
5.3	-	Industrial I/O device	DoS via PROFINET
8.6	-	Temperature sensor	DoS via DHCP
7.7	-	Temperature sensor	DoS via DNS
6.5	-	OPC UA network stack	DoS via OPC UA

As shown in Table 9.2, most of the vulnerabilities are DoS vulnerabilities. Since the availability is a crucial security requirement for OT components [Sto23], it is of special importance to find and close DoS vulnerabilities. This doctoral work contributed to the goal of finding and closing vulnerabilities in OT components by finding and reporting eleven vulnerabilities of which eight have been closed by the manufacturers. Of these eight vulnerabilities, six have a high severity as defined by the CVSS score, and two have a medium severity.

9.2.2 Data and Code Availability

In addition to the descriptions provided in this doctoral work, further content related to the contributions of this doctoral work has been published. Table 9.3 summarizes these additional materials.

Table 9.3: Overview of the information that has been published for the different contributions of this doctoral work in addition to the descriptions presented in this dissertation.

Contribution	Reference	Additional Content
`HitM`	Section 4.3	Paper [*Bor20*] Underlying work [*Wei19*] `mitmproxy` add-ons[a, b]
`ClusterCrash`	Section 5.4	Paper [*Bor22*] VAPs[c] Code upon request
`NeDaP`	Section 6.4	Preliminary implementation[d]
`Smevolution`	Section 7.4	Paper [*Bor23b*] Underlying work [*Mor21*] Code[e] Experimental data[f]
`Palpebratum`	Section 7.5	Underlying work [*Här23*]
`StateBandit`	Section 7.6	Paper [*Bor23a*] Underlying work [*Fit22*]
`SWaTEval`	Section 8.3	Paper [*Bor23c*] Underlying work [*Zim21*, *Hof21*] Code[g] Experimental data[h] Documentation[i]
`MEMA`	Section 8.4	Underlying work [*Bla23*]

[a] https://github.com/mitmproxy/mitmproxy/pull/3961
[b] https://github.com/mitmproxy/mitmproxy/pull/3962
[c] https://github.com/anneborcherding/vulnerability-anti-patterns
[d] https://github.com/anneborcherding/network-packets-preprocessor
[e] https://github.com/anneborcherding/Smarter-Evolution
[f] https://fordatis.fraunhofer.de/handle/fordatis/345
[g] https://github.com/SWaTEval
[h] https://github.com/SWaTEval/evaluation-data
[i] https://swateval.github.io/

Specifically, papers have been published on `HitM`, `ClusterCrash`, `Smevolution`, `StateBandit`, and `SWaTEval`, while publications for `NeDaP` and `Palpebratum` are planned. Since Li et al. published a study similar to our experiments with `MEMA` during the course of our research [Li21], we have opted not to publish these results separately.

We have published the code for several contributions of this doctoral work. This includes the code for the `mitmproxy` add-ons that were developed with `HitM`, the code that we used to define the ML models used for `Smevolution` as well as the code used to interpret the experimental data produced during the experiments with respect to `Smevolution`. Moreover, we published the code of the evaluation framework `SWaTEval`, including the implementation of the target WA and a documentation of the framework.

For `HitM`, `Smevolution`, and `ClusterCrash`, we based our fuzzer implementation on ISuTest®. However, as the process of making the core of ISuTest® open source is still ongoing, the enhancements done during this doctoral work could not yet be published. Despite this, we extracted the test scripts created with `ClusterCrash`, which now can be run without an ISuTest® installation. While the concrete vulnerabilities found during the experiments with `ClusterCrash` have been fixed, these scripts have the potential to crash OT components and other embedded systems. Thus, we opted not to publish them but provide them upon request.

The preliminary results presented with `StateBandit` were based on a preliminary implementation, which is set to be improved upon in future work. Mark Giraud will continue this research direction by developing the RL-based fuzzing approaches using an emulator that offers a deterministic and restorable execution environment, and provides more data and possible actions to the agent Thereby, it has the potential to enhance the agent's decision-making process. Thus, we chose not to publish the preliminary implementation, but to publish the improved and more reliable and flexible implementation in the future.

Our experiments for `SWaTEval` and `Smevolution` are based on artificial vulnerabilities, allowing us to publish our experimental results without exposing unpatched or unpublished vulnerabilities in OT components. For `SWaTEval`, we additionally published the full source code of the framework and the target WA.

We have made the effort to ensure that the approaches and experiments are as transparent and as accessible as possible. Our goal is to provide other researchers and interested parties with the resources to understand, utilize, and further enhance our research.

9.2.3 Applicability

As this doctoral work is located in the domain of testing OT components, we needed to strike a balance between a controlled environment and realistic SuTs for the experiments presented in this dissertation. Our experiments regarding `HitM`, `ClusterCrash` used OT components as SuTs, while `NeDaP`, `Palpebratum`, `StateBandit`, and `MEMA` used network stacks as SuT. As `StateBandit` and `MEMA` are based on a graybox approach, a direct application of those two approaches to OT components is not feasible. Nevertheless, both contribute relevant insights to the security testing domain in general. `Smevolution` and `SWaTEval` were evaluated based on artificial SuTs which aim to represent real SuTs as closely as possible. Both approaches are designed for a blackbox test setting and can be applied to testing OT components, especially `Smevolution` since it is integrated into the blackbox testing framework ISuTest®.

In summary, all blackbox approaches presented in this work have either already been applied to OT component testing, or provide the foundation to apply the new approach to OT component testing. With this, we allow for a practical application of the approaches presented in this dissertation and thus allow for improved security testing of OT components in practice.

9.2.4 Transferability

While the domain of this doctoral work is testing OT components, the results could be transferred to other domains, such as building automation and Internet of Things (IoT).

Building Automation First, the approaches could be transferred to the domain of IoT for building automation. This domain also shares similar challenges [Gra22], as, for example, specialized or proprietary network protocols need to be tested [Mor24]. However, testing approaches such as fuzzing have not yet been applied to this domain extensively [Mor24]. Thus, the approaches presented in this doctoral work could serve as a starting point to improve the security testing of building automation devices.

Internet of Things Furthermore, the approaches presented in this dissertation could be transferred to the general domain of testing embedded systems, such as IoT devices. Testing, especially fuzzing, these devices shares challenges similar to those presented in Section 1.3. For example, the source code for an embedded device's firmware is typically not accessible and cannot be recompiled [Yun22]. In addition, detecting crashes or anomalies in the behavior of an embedded device is challenging [Mue18, Yun22], and resetting the SuT to a known state is time-consuming since it needs to be restarted. Since restarting an embedded device may need several seconds [Mey13], this delays the fuzzing process. Thus, the approaches presented in this doctoral work could be transferred to testing embedded systems.

Software In contrast, transferring the approaches targeting OT components to general software testing might yield different results, as software testing faces different challenges. Especially, the time needed to process a single test case is smaller for software testing than for testing of OT components. For blackbox OT components, the test case needs to be sent via the network interface, which takes longer than providing a software with a specific input [And24]. Moreover, monitoring the current status of the SuT to detect crashes and anomalies generally takes more time for OT components, since the different services of the SuT need to be monitored for a crash. Furthermore, resetting an OT component to a known state includes restarting the whole OT component, while software can be reset to a known state more easily. Considering all of this, the importance of effective test cases is higher for testing OT components, and more overhead to generate the test cases is acceptable. In contrast, experiments by Nicolae et al. [Nic23] with respect to graybox software fuzzing suggest

that spending time budget on the mutation strategy by AFL or AFL++ might lead to better overall results than spending time on more complex approaches such as NEUZZ [She19].

Blackbox Fuzzing Two of the presented approaches, `StateBandit` and MEMA, focus on graybox fuzzing. The question arises whether these approaches could be transferred to blackbox fuzzing. As `StateBandit` aims to utilize information accessible during testing by giving it to a MaB agent, we first analyzed whether graybox fuzzing would provide enough information for such an agent to be successful. However, our experiments show that the MaB-based fuzzers perform significantly worse than the baseline graybox fuzzer. Based on these results, we refrained from attempting to transfer this approach to blackbox fuzzing, where an agent would have even more limited information. Since MEMA analyzes additional performance metrics specific to graybox testing, its implementation and results do not directly apply to blackbox testing. However, future work could analyze which additional performance metrics could be used in blackbox fuzzing and how they would influence the relative assessment of blackbox fuzzers. We would expect the performance metrics to have an impact on the relative assessment in blackbox fuzzing as well.

9.3 Limitations

While the approaches presented in this dissertation contribute to improving blackbox security testing for OT components, there are certain limitations that need to be acknowledged. The following section discusses general limitations, whereas the corresponding sections in previous chapters address limitations specific to each approach.

Evaluation Target

One limitation relates to the targets used in our experiments. When evaluating blackbox TTs for OT component testing, one can choose from three types of targets: (1) a true blackbox OT component, (2) a graybox target, or (3) an artificial target. Each choice presents distinct advantages and disadvantages.

Blackbox OT Component Using a true blackbox OT component provides the most realistic assessment of a TT's performance. However, the lack of access to ground truth data on the OT component's vulnerabilities poses a significant challenge. Moreover, we cannot rely on proxy metrics such as code coverage to assess a TT, since we cannot measure these metrics. Furthermore, there are no established blackbox OT component SuTs that would enable comparisons between different studies.

Graybox Target Graybox targets allow the measurement of proxy metrics, which allows for a more detailed assessment of the TT's performance. Furthermore, SuTs supported by fuzzing benchmarks such as `FuzzBench` [Met21] are widely used, enabling some degree of comparability between studies using these SuTs. However, many graybox SuTs do not provide a ground truth for their bugs, and precisely controlling the included vulnerabilities is difficult.

Artificial SuT An artificial SuT offers a controlled experimental environment where vulnerabilities can be predefined and controlled. However, the use of artificial targets can limit the transferability of results to real-world blackbox OT component testing.

In our experiments, we selected the targets that best suited the specific use case. For example, `HitM` especially addresses the limitations of WVSs in their applicability to OT components, so we evaluated `HitM`'s performance using actual OT components. Conversely, for `Smevolution`, our goal was to gain a detailed understanding of which vulnerabilities different model-based fuzzers could identify. Therefore, we opted for an artificial target where vulnerabilities could be easily defined. While this choice facilitated controlled experiments, it does affect the transferability of the results to blackbox OT component testing.

Target Complexity

Given that most OT components currently incorporate simple, Hypertext Markup Language (HTML)-based WAs, we chose these for our evaluation of `HitM` and `SWaTEval`. However, as future OT components may adopt more modern approaches to WAs, the applicability of `HitM` and `SWaTEval` to these newer WAs will need further evaluation and adaption. Additionally, for `NeDaP` and `Palpebratum`, we focused on the text-based communication protocol FTP. The transferability of these results to binary communication protocols, such as OPC UA, remains uncertain and needs further experiments.

Network Stack Implementation

The majority of the approaches presented in this dissertation focus on test case generation for blackbox testing. These approaches assume access to a communication interface that implements the network protocols used by the SuT. For our practical implementations, we use e.g. the implementation provided by `scapy`[1], which also supports industrial protocols such as PROFINET. However, access to such an implementation might not always be available in a real-world testing scenario, especially if proprietary protocols by third parties need to be tested. This could affect the practical usability of our approaches. A potential mitigation of this limitation could involve combining our methods with techniques that automatically learn a representation of the network protocol communication (e.g. [Gas15, Zhe22]).

[1] https://scapy.net/

10 Future Work

The approaches presented in this dissertation contribute to improving black-box testing of Operational Technology (OT) components, while also revealing opportunities for future investigation. This chapter highlights potential directions for future research, providing suggestions on how the current work could be extended. While this chapter discusses general research opportunities, the corresponding sections in previous chapters focus on future research specific to each approach.

Distribution of Vulnerabilities

The primary objective of Test Tools (TTs), particularly fuzzers, is to identify inputs within the input space that expose vulnerabilities, bugs, or anomalies in the System under Test (SuT). Although various random and deterministic approaches have been developed to approach this objective, we identify a research gap regarding the distribution of these triggering inputs across the input space. To the best of our knowledge, the only empirical study in this area is the work by Bishop presented in 1993, which explores failure regions within a program's continuous input space [Bis93]. This study reveals contiguous failure regions in two Fortran programs' input spaces and the analyses suggest that such contiguous failure regions are likely to exist in any program. With respect to coverage in a graybox test setting, Zhang et al. show that for some programs, 1% of the input bytes contribute up to 90% of the covered edges, also suggesting relative locality of the coverage findings [Zha24a].

While acknowledging that this is highly time-consuming, we suggest conducting a comprehensive analysis of the input space of several SuTs. This would provide valuable insights into how bugs and vulnerabilities are distributed within the input space. Based on these insights, TTs could be designed to leverage this knowledge effectively, as, e.g., discussed by Chen et al. [Che10].

Recent Machine Learning Approaches

Recent advances in Machine Learning (ML) offer new possibilities for blackbox testing of OT components. For instance, transformer models [Vas17] could be employed to identify efficient encodings for test cases and network traffic, which can then be utilized by downstream ML models for test case generation.

Transformer models could also be applied directly for generating test cases in blackbox OT component testing. In other domains, several studies have explored the potential of Large Language Models (LLMs) for test case generation, with promising results (e.g. [Liu24b, Isa24, Wan24, Xia24, Den24]). Other studies evaluate the use of LLMs for automated unit test generation (e.g. [Oué24, Yan24]), and for generating test cases from natural language descriptions, bug reports, or specifications (e.g. [Ras24, Xue24, Ple24, Ma24]). Although these studies address different use cases, they highlight the potential for leveraging LLMs in automated testing.

Additional Information Sources

Beyond the information sources discussed in Chapter 3, other information sources could be utilized for blackbox testing. For example, techniques used in asset discovery within industrial networks could provide more information on the SuT (e.g. [Cas13, Wed15, Sam22]). Additionally, information explicitly stored about an OT component, such as that found in the asset administration shell [Ye19], could be utilized to tailor test cases to the specific features of the SuT. For instance, since many OT components share the same network stack,

and thus may exhibit similar vulnerabilities [*Pfr19a*], a TT could first identify the network stack in use, and then focus on testing for known vulnerability patterns associated with it first.

Software Testing

Research on software testing in general as well as security testing was conducted with respect to several domains, but the relationship between these two fields has not been discussed in depth [Awa23]. Awalurahman et al. present a systematic literature review with the goal of providing new insights into this relationship. While it gives an overview of different approaches and aspects of software testing and security testing, a more in-depth analysis might be necessary to fully understand the implications and potentials of the relationship between software testing and security testing [Awa23]. Nevertheless, the relationship and also a possible convergence of software testing and security testing shows in some publications. For example, fuzzing was historically primarily used for security testing, but is also used for general software testing nowadays [Man19]. In addition, Gopinath et al. suggest to apply mutation analysis, an approach from software testing, to improve fuzzing [Gop22]. Future work could encompass a more in-depth analysis of the relationship between these two research domains, which could provide insights into possible synergies.

System Identification

Rooted in control theory, system identification focuses on building a mathematical model of a system based on observed inputs and outputs (see e.g. [Lju10, Kee11]). One possible direction for future research is to explore whether techniques from system identification could be applied to construct a model of the SuT. Such a model could then be used to guide and improve the blackbox testing of OT components.

11 Summary

The primary objective of this doctoral work was to improve blackbox testing for Operational Technology (OT) components by utilizing information accessible before and during testing. Through various contributions that consider different Systems under Test (SuTs) and utilize different information sources, this doctoral work advances the state-of-the-art in this research area. The approaches presented in this doctoral work demonstrate how accessible information can be utilized to improve blackbox testing of OT components. Their practical application uncovered several previously unknown vulnerabilities in OT components. These vulnerabilities were disclosed responsibly, and a total of nine vulnerabilities were confirmed by the respective manufacturers (see Section 9.2.1). Among these, five were assigned Common Vulnerabilities and Exposures (CVE) identifiers [CVE18, CVE19, CVE21a, CVE21b, CVE21c]. With this, this doctoral work contributes to the security of OT components and the production environments in which they are deployed. Thus, this doctoral work advances the field of blackbox testing of OT components and has a tangible impact on the practical security of critical infrastructure.

Analysis Our analysis demonstrates that a variety of static and dynamic information sources can be utilized to improve blackbox testing (see Chapter 3). These information sources are accessible either prior to or during testing. The contributions newly presented in this doctoral work utilize these information sources in novel ways to further improve the performance of blackbox Test Tools (TTs).

`HitM` We introduced `HitM`, a proxy-based solution that transparently improves the performance of blackbox Web Vulnerability Scanners (WVSs) when testing OT components (see Chapter 4 and [*Bor20*]). `HitM` addresses several challenges specific to OT components, such as their limited resources, which often result

in early crashes. Our experiments, conducted using six WVSs and five OT components, demonstrate that HitM increases the number of true positive reports and improves Uniform Resource Locator (URL) coverage. However, this improvement comes at the cost of increased runtime per test case and a higher false positive rate. Notably, one of the vulnerabilities identified during the evaluation of HitM was assigned a CVE identifier [CVE18].

ClusterCrash ClusterCrash utilizes information on known vulnerabilities that is accessible prior to testing a concrete SuT (see Chapter 5 and [*Bor22*]). This approach aggregates and structures information on vulnerabilities identified using graybox and whitebox tests by using Vulnerability Anti-Patterns (VAPs). From these VAPs, we derive blackbox test scripts tailored to identify similar vulnerabilities in a blackbox test setting. In our experiments, we applied these newly developed test scripts to eight OT components, resulting in eleven findings. Of these findings, three vulnerabilities were assigned CVE identifiers [CVE21c, CVE21a, CVE21b]. Additionally, our findings demonstrate that VAPs are effective in identifying similar vulnerabilities across different protocols and device classes.

NeDaP With NeDaP, we introduced a tool to preprocess another information source that is accessible both before and during testing: network traffic (see Chapter 6). We propose a preprocessing pipeline that includes a dimensionality reduction to make the information more accessible by downstream Machine Learning (ML) models. We evaluated the performance of three different approaches to dimensionality reduction, highlighting that each approach has distinct advantages and disadvantages. Thus, the decision on the dimensionality reduction approach depends on the downstream ML model which builds upon the preprocessed network packets.

Palpebratum With Palpebratum, we proposed a novel approach to modeling the behavior of the SuT based on network traffic generated during testing (see Section 7.5). The network traffic is first preprocessed using the NeDaP pipeline, and then utilized to query an Hidden Markov Model (HMM) that approximates the behavior of the SuT. This HMM is employed to estimate the *interestingness* of a test case, and thus to guide the test case generation. Our experiments, using a File Transfer Protocol (FTP) implementation as target,

demonstrated that the test cases generated by an HMM-based fuzzer are more effective than those generated by a traditional blackbox fuzzer. However, the final coverage achieved by a blackbox fuzzer significantly outperforms the coverage achieved by a HMM-based fuzzer. One reason for this could be the overhead the HMM introduces.

`Smevolution` `Smevolution` employs an ML model to approximate the behavior of the SuT based on the services of the SuT that crash in response to a specific test cases (see Section 7.4 and [*Bor23b*]). This model is embedded into an evolutionary test case generation algorithm in order to provide additional insights for mutating and selecting the test case individuals. We evaluated this approach using three different ML models: a Decision Tree (DT), a Neural Network (NN), and an Support Vector Machine (SVM). Experiments conducted on an artificial SuT reveal that the DT-based fuzzer significantly outperforms the baseline fuzzer in terms of bugs detected. This result suggests that the DT was particularly effective in approximating the SuT's behavior, leading to improved testing performance.

`StateBandit` With `StateBandit`, we explored the use of a Multi-armed Bandit (MaB) agent to approach the state selection problem of stateful network fuzzing (see Section 7.6 and [*Bor23a*]). To gain initial insights, we evaluated `StateBandit` in a graybox test scenario. Our experiments, using an OPC Unified Architecture (OPC UA) network stack as target, revealed that the state-of-the-art fuzzer `AFLNet` significantly outperforms the MaB-based fuzzers. Due to these results, we decided not to transfer this approach to a blackbox test scenario, where the agents would have even less information to build their decisions on.

`SWaTEval` In addition to proposing new approaches to utilize the information accessible in blackbox testing, this doctoral work also contributes to the field of evaluating TTs (see Section 8.3 and [*Bor23c*]). `SWaTEval` is a novel framework designed to evaluate stateful WVSs. This framework includes an artificial Web Application (WA) that serves as an evaluation target. Using `SWaTEval` we investigated the impact of different similarity measures on the

performance of the state machine inference employed by stateful WVS. Our experiments demonstrate that the choice of similarity measures indeed affects the performance on the state machine inference.

MEMA MEMA enhances an existing fuzzer benchmark by introducing additional performance metrics to be measured during testing, allowing for an analysis of their impact on the relative assessment of fuzzers (see Section 8.4). We focus on stateless graybox fuzzers, and we conducted our evaluation using five graybox targets. Our experiments indicate that the selection of performance metrics indeed influences the relative ranking of fuzzers, highlighting the importance of choosing appropriate metrics for accurate fuzzer evaluations.

Bibliography

[Agg01] Aggarwal, Charu; Hinneburg, Alexander and Keim, Daniel: "On the surprising behavior of distance metrics in high dimensional space". In: *Database theory—ICDT 2001: 8th international conference London, UK, January 4–6, 2001 proceedings 8*. Springer. 2001, pp. 420–434.

[Aic21] Aichernig, Bernhard; Muškardin, Edi and Pferscher, Andrea: "Learning-based fuzzing of IoT message brokers". In: *14th Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 47–58.

[Ala12] Alassmi, Shafi; Zavarsky, Pavol; Lindskog, Dale; Ruhl, Ron; Alasiri, Ahmed and Alzaidi, Muteb: "An analysis of the Effectiveness of Black-box Web Application Scanners in Detection of Stored XSSI Vulnerabilities". In: *International Journal of Information Technology and Computer Science* 4.1 (2012), pp. 40–48.

[Ala22] Alazmi, Suliman and De Leon, Daniel Conte: "A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners". In: *IEEE Access* 10 (2022), pp. 33200–33219.

[Ali19] Alidoosti, Mitra; Nowroozi, Alireza and Nickabadi, Ahmad: "BLProM: A Black-Box Approach for Detecting Business-Layer Processes in the Web Applications". In: *Journal of Computing & Security* 6.2 (2019), pp. 65–80.

[Amu23] Amusuo, Paschal; Méndez, Ricardo Andrés Calvo; Xu, Zhongwei; Machiry, Aravind and Davis, James: "Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 926–938.

[And24] Andarzian, Seyed Behnam; Daniele, Cristian and Poll, Erik: "On the (in) efficiency of fuzzing network protocols". In: *Annals of Telecommunications* (2024), pp. 1–10.

[Ani21] Ani, Uchenna P Daniel; Watson, Jeremy; Green, Benjamin; Craggs, Barnaby and Nurse, Jason: "Design considerations for building credible security testbeds: Perspectives from industrial control system use cases". In: *Journal of Cyber Security Technology* 5.2 (2021), pp. 71–119.

[Ant12] Antonello, Rafael; Fernandes, Stenio; Kamienski, Carlos; Sadok, Djamel; Kelner, Judith; Godor, Istvan; Szabo, Geza and Westholm, Tord: "Deep packet inspection tools and techniques in commodity platforms: Challenges and trends". In: *Journal of Network and Computer Applications* 35.6 (2012), pp. 1863–1878.

[Ant21] Anton, Simon Daniel Duque; Fraunholz, Daniel; Krohmer, Daniel; Reti, Daniel; Schneider, Daniel and Schotten, Hans Dieter: "The Global State of Security in Industrial Control Systems: An Empirical Analysis of Vulnerabilities Around the World". In: *IEEE Internet of Things Journal* 8.24 (2021), pp. 17525–17540.

[App18] Appelt, Dennis; Nguyen, Cu; Panichella, Annibale and Briand, Lionel: "A machine-learning-driven evolutionary approach for testing web application firewalls". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 733–757.

[Arc11] Arcuri, Andrea and Briand, Lionel: "A practical guide for using statistical tests to assess randomized algorithms in software engineering". In: *Proceedings of the 33rd international conference on software engineering*. 2011, pp. 1–10.

[Arr18] Arrieta, Aitor; Wang, Shuai; Arruabarrena, Ainhoa; Markiegi, Urtzi; Sagardui, Goiuria and Etxeberria, Leire: "Multi-objective black-box test case selection for cost-effectively testing simulation models". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2018, pp. 1411–1418.

[Atl19] Atlidakis, Vaggelis; Godefroid, Patrice and Polishchuk, Marina: "Restler: Stateful rest api fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 748–758.

[Aud07] Audibert, Jean-Yves; Munos, Rémi and Szepesvári, Csaba: "Tuning bandit algorithms in stochastic environments". In: *International conference on algorithmic learning theory*. Springer. 2007, pp. 150–165.

[Aue02a] Auer, Peter; Cesa-Bianchi, Nicolo and Fischer, Paul: "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47 (2002), pp. 235–256.

[Aue02b] Auer, Peter; Cesa-Bianchi, Nicolo; Freund, Yoav and Schapire, Robert E: "The nonstochastic multiarmed bandit problem". In: *SIAM journal on computing* 32.1 (2002), pp. 48–77.

[Avc23] Avci, Deniz Imge: "Using Hidden Markov Models for Blackbox Fuzzing". Supervised by Anne Borcherding and Mark Giraud. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[Awa23] Awalurahman, Halim Wildan; Witsqa, Ibrahim Hafizhan; Raharjana, Indra Kharisma and Basori, Ahmad Hoirul: "Security Aspect in Software Testing Perspective: A Systematic Literature Review". In: *Journal of Information Systems Engineering & Business Intelligence* 9.1 (2023), pp. 95–107.

[Ba22] Ba, Jinsheng; Böhme, Marcel; Mirzamomen, Zahra and Roychoudhury, Abhik: "Stateful greybox fuzzing". In: *31st USENIX Security Symposium*. 2022, pp. 3255–3272.

[Bae17] Baezner, Marie and Robin, Patrice: Stuxnet. Tech. rep. ETH Zurich, 2017.

[Bar14] Bartz-Beielstein, Thomas; Branke, Jürgen; Mehnen, Jörn and Mersmann, Olaf: "Evolutionary algorithms". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4.3 (2014), pp. 178–195.

[Bau10] Bau, Jason; Bursztein, Elie; Gupta, Divij and Mitchell, John: "State of the art: Automated black-box web application vulnerability testing". In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2010, pp. 332–345.

[Bin23] Binosi, Lorenzo; Rullo, Luca; Polino, Mario; Carminati, Michele; Zanero, Stefano et al.: "Rainfuzz: Reinforcement-Learning Driven Heat-Maps for Boosting Coverage-Guided Fuzzing". In: *13th International Conference on Pattern Recognition Applications and Methods (ICPRAM)*. 2023, pp. 39–50.

[Bis93] Bishop, Peter G: "The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail)". In: *The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*. IEEE. 1993, pp. 98–107.

[Biß23] Biss, Klaus; Kippe, Jörg and Karch, Markus: "Device discovery and identification in industrial networks: Geräteerkennung und-identifizierung in industriellen Netzen". In: *at-Automatisierungstechnik* 71.9 (2023), pp. 726–735.

[Bla23] Blankefort, Tom: "Systematische Evaluation von Fuzzern". Supervised by Anne Borcherding and Mark Giraud. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[Böh11] Böhr, Frank: "Model based statistical testing of embedded systems". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 18–25.

[Böh16] Böhme, Marcel; Pham, Van-Thuan and Roychoudhury, Abhik: "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1032–1043.

[Böh17] Böhme, Marcel; Pham, Van-Thuan; Nguyen, Manh-Dung and Roychoudhury, Abhik: "Directed greybox fuzzing". In: *2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[Böh20] Böhme, Marcel; Manès, Valentin and Cha, Sang Kil: "Boosting fuzzer efficiency: An information theoretic perspective". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 678–689.

[Böh21] Böhme, Marcel; Cadar, Cristian and Roychoudhury, Abhik: "Fuzzing: Challenges and Reflections". In: *IEEE Software* 38.3 (2021), pp. 79–86.

[Böh22] Böhme, Marcel; Szekeres, László and Metzman, Jonathan: "On the reliability of coverage-based fuzzer benchmarking". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1621–1633.

[Bor18] Borcherding, Anne: "Security-Testing für Webserver auf industriellen Automatisierungskomponenten". MA thesis. Karlsruhe Institute of Technology (KIT), 2018.

[Bor19] Borcherding, Anne; Goerke, Niklas and Klamroth, Jonas: Prüfmethoden für Security. German. Tech. rep. Kooperationsprojekt Cyberprotect, 2019.

[Bor20] Borcherding, Anne; Pfrang, Steffen; Haas, Christian; Weiche, Albrecht and Beyerer, Jürgen: "Helper-in-the-Middle: Supporting Web Application Scanners Targeting Industrial Control Systems". In: *Proceedings of the 17th International Conference on Security and Cryptography (SECRYPT / ICETE 2020)*. SCITEPRESS - Science and Technology Publications, 2020, pp. 27–38.

[Bor21] Borcherding, Anne; Takacs, Philipp and Beyerer, Jürgen: Vulnerability Anti-Patterns. Dec. 16, 2021. url: https://github.com/anneborcherding/vulnerability-anti-patterns (visited on 04/20/2024).

[Bor22] Borcherding, Anne; Takacs, Philipp and Beyerer, Jürgen: "Cluster Crash: Learning from Recent Vulnerabilities in Communication Stacks". In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*. 2022, pp. 334–344.

[Bor23a] Borcherding, Anne; Giraud, Mark; Fitzgerald, Ian and Beyerer, Jürgen: "The Bandit's States: Modeling State Selection for Stateful Network Fuzzing as Multi-armed Bandit Problem". In: *1st International Workshop on Re-design Industrial Control Systems with Security (RICSS)*. 2023, pp. 345–350.

[Bor23b] Borcherding, Anne; Morawetz, Martin and Pfrang, Steffen: "Smarter Evolution: Enhancing Evolutionary Black Box Fuzzing with Adaptive Models". In: *Sensors* 23.18 (2023).

[Bor23c] Borcherding, Anne; Penkov, Nikolay; Giraud, Mark and Beyerer, Jürgen.: "SWaTEval: An Evaluation Framework for Stateful Web Application Testing". In: *Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP,* INSTICC. SCITEPRESS - Science and Technology Publications, 2023, pp. 430–441.

[Bor23d] Borcherding, Anne; Pfrang, Steffen and Haas, Christian: "Test assembly and method for testing a test object, and computer program for carrying out the method". WO2023144158 A1. 2023.

[Böt18] Böttinger, Konstantin; Godefroid, Patrice and Singh, Rishabh: "Deep reinforcement fuzzing". In: *Security and Privacy Workshops (SPW)*. IEEE. 2018, pp. 116–122.

[Bou20] Bouneffouf, Djallel; Rish, Irina and Aggarwal, Charu: "Survey on applications of multi-armed and contextual bandits". In: *Congress on Evolutionary Computation (CEC)*. IEEE. 2020, pp. 1–8.

[Byt23] Bytes, Andrei; Rajput, Prashant Hari Narayan; Doumanidis, Constantine; Maniatakos, Michail; Zhou, Jianying and Tippenhauer, Nils Ole: "FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023, pp. 499–512.

[Cab23] Cabello-Solorzano, Kelsy; Ortigosa de Araujo, Isabela; Peña, Marco; Correia, Luís and Tallón-Ballesteros, Antonio: "The Impact of Data Normalization on the Accuracy of Machine Learning Algorithms: A Comparative Analysis". In: *International Conference on Soft Computing Models in Industrial and Environmental Applications*. Springer. 2023, pp. 344–353.

[Cad22] Cady, Field: Training Hidden Markov Models - The Baum-Welch and Forward-Backward Algorithms. Jan. 31, 2022. url: https://towardsdatascience.com/training-hidden-markov-models-831c1bdec27d (visited on 05/02/2024).

[Cas13] Caselli, Marco; Hadžiosmanović, Dina; Zambon, Emmanuele and Kargl, Frank: "On the feasibility of device fingerprinting in industrial control systems". In: *Critical Information Infrastructures Security: 8th International Workshop (CRITIS 2013)*. Springer. 2013, pp. 155–166.

[Cas16] Caselli, Marco and Kargl, Frank: "A security assessment methodology for critical infrastructures". In: *Critical Information Infrastructures Security: 9th International Conference, CRITIS 2014, Limassol, Cyprus, October 13-15, 2014, Revised Selected Papers 9*. Springer. 2016, pp. 332–343.

[Cer20] Cervantes, Jair; Garcia-Lamont, Farid; Rodríguez-Mazahua, Lisbeth and Lopez, Asdrubal: "A comprehensive survey on support vector machine classification: Applications, challenges and trends". In: *Neurocomputing* 408 (2020), pp. 189–215.

[Cer23] Cerquozzi, Renzo et al.: Certified Tester Foundation Level Syllabus v4.0. 2023.

[Cha24] Chafjiri, Sadegh Bamohabbat; Legg, Phil; Hong, Jun and Tsompanas, Michail-Antisthenis: "Vulnerability detection through machine learning-based fuzzing: A systematic review". In: *Computers & Security* (2024).

[Che10] Chen, Tsong Yueh; Kuo, Fei-Ching; Merkel, Robert and Tse, T.H.: "Adaptive random testing: The art of test case diversity". In: *Journal of Systems and Software* 83.1 (2010), pp. 60–66.

[Che18] Chen, Shay: Evaluation of Web Application Vulnerability Scanners in Modern Pentest/SSDLC Usage Scenarios. 2018. url: https://sectooladdict.blogspot.com/ (visited on 08/27/2024).

[Che19] Chen, Yuqi; Poskitt, Christopher; Sun, Jun; Adepu, Sridhar and Zhang, Fan: "Learning-guided network fuzzing for testing cyber-physical system defences". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 962–973.

[Che20] Chen, Yaohui; Ahmadi, Mansour; Wang, Boyu; Lu, Long et al.: "MEUZZ: Smart seed scheduling for hybrid fuzzing". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 2020, pp. 77–92.

[Chi20] Chiu, Kai-Cheng; Liu, Chien-Chang and Chou, Li-Der: "CAPC: packet-based network service classifier with convolutional autoencoder". In: *IEEE Access* 8 (2020), pp. 218081–218094.

[Cho21] Chouchen, Moataz; Ouni, Ali; Kula, Raula Gaikovina; Wang, Dong; Thongtanunam, Patanamon; Mkaouer, Mohamed Wiem and Matsumoto, Kenichi: "Anti-patterns in modern code review: Symptoms and prevalence". In: *IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2021, pp. 531–535.

[CVE18] CVE: CVE-2018-16994. Denial of service in several fieldbus couplers (severity 7.5). 2018. URL: https://www.cve.org/CVERecord?id=CVE-2018-16994.

[CVE19] CVE: CVE-2019-14753. Buffer overflow in a safety PROFINET gateway (severity 7.5). 2019. URL: https://www.cve.org/CVERecord?id=CVE-2019-14753.

[CVE21a] CVE: CVE-2021-21003. Denial of service in industrial switch (severity 5.3). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21003.

[CVE21b] CVE: CVE-2021-21004. Cross-site scripting vulnerability in industrial switch (severity 7.4). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21004.

[CVE21c] CVE: CVE-2021-21005. Denial of service in industrial switch (severity 7.5). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21005.

[Dai08] Dainotti, Alberto; Pescapé, Antonio; Rossi, Pierluigi Salvo; Palmieri, Francesco and Ventre, Giorgio: "Internet traffic modeling by means of Hidden Markov Models". In: *Computer Networks* 52.14 (2008), pp. 2645–2662.

[Dan24] Daniele, Cristian; Andarzian, Seyed Behnam and Poll, Erik: "Fuzzers for stateful systems: Survey and Research Directions". In: *ACM Computing Surveys* 56.9 (2024), pp. 1–23.

[Dee18] Deepa, Ganesan; Thilagam, Santhi; Praseed, Amit and Pais, Alwyn: "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications". In: *Journal of Network and Computer Applications* 109 (2018), pp. 89–109.

[DeM07] DeMott, Jared; Enbody, Richard and Punch, William F: "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing". In: *BlackHat and Defcon* (2007).

[Den24] Deng, Yinlin; Xia, Chunqiu Steven; Yang, Chenyuan; Zhang, Shizhuo Dylan; Yang, Shujing and Zhang, Lingming: "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–13.

[Dil20] Dillmann, Joerg: "Systematisch unterstützte Durchführung und Auswertung von Security-Tests mit ISuTest". Supervised by Steffen Pfrang and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[DiP18] DiPinto, Alessandro; Dragoni, Younes and Carcano, Andrea: "TRITON: The first ICS cyber attack on safety instrument systems". In: *Black Hat USA* 2018 (2018), pp. 1–26.

[Dou10] Doupé, Adam; Cova, Marco and Vigna, Giovanni: "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2010, pp. 111–131.

[Dou12] Doupé, Adam; Cavedon, Ludovico; Kruegel, Christopher and Vigna, Giovanni: "Enemy of the state: A state-aware black-box web vulnerability scanner". In: *21st USENIX Security Symposium*. 2012, pp. 523–538.

[Dou23] Doumanidis, Constantine; Xie, Yongyu; Rajput, Prashant; Pickren, Ryan; Sahin, Burak; Zonouz, Saman and Maniatakos, Michail: "Dissecting the Industrial Control Systems Software Supply Chain". In: *IEEE Security & Privacy* (2023).

[Dra20] Drakonakis, Kostas; Ioannidis, Sotiris and Polakis, Jason: "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1953–1970.

[Dra22] Dragos: ICS/OT Cybersecurity Year in Review 2022. Tech. rep. Dragos, Inc., 2022.

[Dra23a] Dragos: ICS/OT Cybersecurity Year in Review 2023. Tech. rep. Dragos, Inc., 2023.

[Dra23b] Drakonakis, Kostas; Ioannidis, Sotiris and Polakis, Jason: "ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning". In: *Network and Distributed System Security (NDSS) Symposium*. 2023.

[Duc14] Duchene, Fabien; Rawat, Sanjay; Richier, Jean-Luc and Groz, Roland: "KameleonFuzz: evolutionary fuzzing for black-box XSS detection". In: *4th ACM conference on Data and application security and privacy*. 2014, pp. 37–48.

[Dwi23] Dwivedi, Rudresh; Dave, Devam; Naik, Het; Singhal, Smiti; Omer, Rana; Patel, Pankesh; Qian, Bin; Wen, Zhenyu; Shah, Tejal; Morgan, Graham et al.: "Explainable AI (XAI): Core ideas, techniques, and solutions". In: *ACM Computing Surveys* 55.9 (2023), pp. 1–33.

[Eis23] Eisele, Max; Ebert, Daniel; Huth, Christopher and Zeller, Andreas: "Fuzzing Embedded Systems Using Debug Interfaces". In: *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. 2023.

[Esp18] Esposito, Damiano; Rennhard, Marc; Ruf, Lukas and Wagner, Arno: "Exploiting the potential of web application vulnerability scanning". In: *ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018*. IARIA. 2018, pp. 22–29.

[Fan18a] Fan, Rong and Chang, Yaoyao: "Machine learning for black-box fuzzing of network protocols". In: *Information and Communications Security: 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings 19*. Springer. 2018, pp. 621–632.

[Fan18b] Fang, Kaiming and Yan, Guanhua: "Emulation-instrumented fuzz testing of 4g/lte android mobile devices guided by reinforcement learning". In: *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*. Springer. 2018, pp. 20–40.

[Fed22] Federal Office for Information Security: Industrial Control System Security – Top 10 threats and countermeasures 2022. Tech. rep. German Federal Office for Information Security (BSI), 2022.

[Fel16a] Felderer, Michael; Büchler, Matthias; Johns, Martin; Brucker, Achim; Breu, Ruth and Pretschner, Alexander: "Security testing: A survey". In: *Advances in Computers*. Vol. 101. Elsevier, 2016, pp. 1–51.

[Fel16b] Felderer, Michael; Zech, Philipp; Breu, Ruth; Büchler, Matthias and Pretschner, Alexander: "Model-based security testing: a taxonomy and systematic classification". In: *Software testing, verification and reliability* 26.2 (2016), pp. 119–148.

[Fel23] Felício, Duarte; Simão, José and Datia, Nuno: "Rapitest: Continuous black-box testing of restful web apis". In: *Procedia Computer Science* 219 (2023), pp. 537–545.

[Fer11] Ferreira, Alexandre Miguel and Kleppe, Harald: "Effectiveness of automated application penetration testing tools". MA thesis. University of Amsterdam, 2011.

[Fer22] Fernandez, Leon; Karlsson, Gunnar and Hübinette, Daniel: A framework for feedback-enabled Blackbox fuzzing using context-free grammars. Tech. rep. KTH Royal Institute of Technology, and Infinera, 2022.

[Fet02] Fetzer, Christof and Xiao, Zhen: "An automated approach to increasing the robustness of C libraries". In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE. 2002, pp. 155–164.

[Fie99] Fielding, Roy; Gettys, James; Mogul, Jeffrey; Frystyk, Henrik; Masinter, Larry; Leach, Paul and Berners-Lee, Tim: Hypertext Transfer Protocol – HTTP/1.1. STD. RFC Editor, June 1999. url: http://www.rfc-editor.org/rfc/rfc2616.txt.

[Fio20] Fioraldi, Andrea; Maier, Dominik; Eißfeldt, Heiko and Heuse, Marc: "AFL++ combining incremental steps of fuzzing research". In: *14th USENIX Conference on Offensive Technologies*. 2020, pp. 10–10.

[Fio22] Fioraldi, Andrea; Maier, Dominik Christian; Zhang, Dongjia and Balzarotti, Davide: "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1051–1065.

[Fit22] Fitzgerald, Ian: "Zustandsauswahl für zustandsbehaftetes Fuzzing als Multi-Armed Bandit Problem: Modellierung und Evaluation". Supervised by Anne Borcherding and Mark Giraud. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2022.

[Flu22] Fluchs, Sarah; Drath, Rainer and Fay, Alexander: "A security decision base: How to prepare security by design decisions for industrial control systems". In: *17. EKA (Entwurf Komplexer Automatisierungssysteme)*. 2022.

[Fon07a] Fong, Elizabeth and Okun, Vadim: "Web application scanners: definitions and functions". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE. 2007, 280b–280b.

[Fon07b] Fonseca, Jose; Vieira, Marco and Madeira, Henrique: "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks". In: *13th Pacific Rim international symposium on dependable computing (PRDC 2007)*. IEEE. 2007, pp. 365–372.

[For10] Forouzan, Behrouz: TCP/IP protocol suite – 4th Edition. 2010.

[Gal10] Galán, Eduardo; Alcaide, Almudena; Orfila, Agustín and Blasco, Jorge: "A multi-agent scanner to detect stored-XSS vulnerabilities". In: *2010 International Conference for Internet Technology and Secured Transactions*. IEEE. 2010, pp. 1–6.

[Gas15] Gascon, Hugo; Wressnegger, Christian; Yamaguchi, Fabian; Arp, Daniel and Rieck, Konrad: "Pulsar: Stateful black-box fuzzing of proprietary network protocols". In: *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015*. Springer. 2015, pp. 330–347.

[Gau21] Gauthier, François; Hassanshahi, Behnaz; Selwyn-Smith, Benjamin; Mai, Trong Nhan; Schlüter, Max and Williams, Micah: "Backrest: A model-based feedback-driven greybox fuzzer for web applications". In: *arXiv preprint arXiv:2108.08455* (2021).

[Gav20] Gavrilov, Miroslav; Dewey, Kyle; Groce, Alex; Zamanzadeh, Davina and Hardekopf, Ben: "A practical, principled measure of fuzzer appeal: A preliminary study". In: *20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2020, pp. 510–517.

[Gei20] Geismann, Johannes and Bodden, Eric: "A systematic literature review of model-driven security engineering for cyber–physical systems". In: *Journal of Systems and Software* 169 (2020), p. 110697.

[Gér22] Géron, Aurélien: Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, Inc., 2022.

[Gho19] Ghosh, Angkush Kumar; Ullah, AMM Sharif and Kubo, Akihiko: "Hidden Markov model-based digital twin construction for futuristic manufacturing systems". In: *Ai Edam* 33.3 (2019), pp. 317–331.

[Gir20] Giraud, Mark Leon: "Design and evaluation of methods for efficient fuzzing of stateful software". Supervised by Christian Haas and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[God20] Godefroid, Patrice: "Fuzzing: Hack, art, and science". In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[Goh24] Gohil, Vasudev; Kande, Rahul; Chen, Chen; Sadeghi, Ahmad-Reza and Rajendran, Jeyavijayan: "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[Gon19] Gonzalez, Danielle; Alhenaki, Fawaz and Mirakhorli, Mehdi: "Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities". In: *International Conference on Software Architecture (ICSA)*. IEEE. 2019, pp. 31–40.

[Gop20] Gopinath, Rahul; Bendrissou, Bachir; Mathis, Björn and Zeller, Andreas: "Fuzzing with fast failure feedback". In: *arXiv preprint arXiv:2012.13516* (2020).

[Gop22] Gopinath, Rahul; Görz, Philipp and Groce, Alex: "Mutation analysis: Answering the fuzzing challenge". In: *arXiv preprint arXiv:2201.11303* (2022).

[Gra22] Graveto, Vitor; Cruz, Tiago and Simões, Paulo: "Security of Building Automation and Control Systems: Survey and future research directions". In: *Computers & Security* 112 (2022), p. 102527.

[Gre23] Greenbone: Our Solutions in Comparison – Greenbone Community Edition, Greenbone Enterprise Appliances and Greenbone Cloud Service. Tech. rep. Greenbone, 2023.

[Guo24] Guo, Jiaxing; Gu, Chunxiang; Chen, Xi; Zhang, Xieli; Tian, Kai and Li, Ji: "Stateful black-box fuzzing for encryption protocols and its application in IPsec". In: *Computer Networks* (2024), p. 110605.

[Haj21] Hajda, Janusz; Jakuszewski, Ryszard and Ogonowski, Szymon: "Security Challenges in Industry 4.0 PLC Systems". In: *Applied Sciences* 11.21 (2021), p. 9785.

[Här23] Häring, Johannes: "Enlightening the Dark: Estimating Code Coverage during Blackbox Fuzzing". Supervised by Anne Borcherding. Seminar "KI Systems Engineering". Karlsruhe Institute of Technology (KIT), 2023.

[Has22] Hassanshahi, Behnaz; Lee, Hyunjun and Krishnan, Paddy: "Gelato: Feedback-driven and guided security analysis of client-side web applications". In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 618–629.

[Haz] Hazimeh, Ahmad; Herrera, Adrian and Payer, Mathias: Magma: A Ground-Truth Fuzzing Benchmark. url: https://hexhive.epfl.ch/magma/docs/faq.html (visited on 07/08/2024).

[Haz20] Hazimeh, Ahmad; Herrera, Adrian and Payer, Mathias: "Magma: A ground-truth fuzzing benchmark". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[Hec15] Hecht, Geoffrey; Rouvoy, Romain; Moha, Naouel and Duchien, Laurence: "Detecting antipatterns in android apps". In: *2015 2nd ACM international conference on mobile software engineering and systems*. IEEE. 2015, pp. 148–149.

[Hel24] Helin, Aki: radamsa. Feb. 27, 2024. url: https://gitlab.com/akihe/radamsa (visited on 08/08/2024).

[Heu24] Heuser, Marc; Eissfeld, Heiko; Fioraldi, Andrea and Maier, Dominik: AFL++ Documentation. July 1, 2024. URL: https://aflplus.plus/ (visited on 07/05/2024).

[Hir09] Hirvonen, Matti and Laulajainen, Jukka-Pekka: "Two-phased network traffic classification method for quality of service management". In: *13th International Symposium on Consumer Electronics*. IEEE. 2009, pp. 962–966.

[Hof21] Hofmann, Felix: "Black-Box-Sicherheitsanalyse von zustandsbehafteten Webanwendungen". Supervised by Anne Borcherding and Florian Patzer. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[Hol11] Holm, Hannes; Sommestad, Teodor; Almroth, Jonas and Persson, Mats: "A quantitative evaluation of vulnerability scanning". In: *Information Management & Computer Security* (2011).

[Hou12] Householder, Allen and Foote, Jonathan: Probability-based parameter selection for black-box fuzz testing. Tech. rep. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2012-TN-019, 2012.

[Idr17] Idrissi, Siham El; Berbiche, Naoual; Guerouate, Fatima and Shibi, Mohamed: "Performance evaluation of web application security scanners for prevention and protection against vulnerabilities". In: *International Journal of Applied Engineering Research* 12.21 (2017), pp. 11068–11076.

[Idr19] Idrissi, Omar El; Mezrioui, Abdellatif and Belmekki, Abdelhamid: "Cyber security challenges and issues of industrial control systems–some security recommendations". In: *International Smart Cities Conference (ISC2)*. IEEE. 2019, pp. 330–335.

[Idr21] Idrissi, Omar El; Mezrioui, Abdellatif and Belmekki, Abdelhamid: "Interactions between cyber security and safety in the ICS context". In: *Journal of Information Assurance & Security* 16.2 (2021).

[Int10] International Electrotechnical Commission: IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Standard. Geneva, CH: International Electrotechnical Commission, 2010.

[Int19] International Electrotechnical Commission: IEC 62443 Security for Industrial Automation and Control Systems. Standard. Geneva, CH: International Electrotechnical Commission, 2019.

[Int22] International Electrotechnical Commission: IEC 61139-2 Industrial networks - Single-drop digital communication interface - Part 2: Functional safety extensions. Standard. Geneva, CH: International Electrotechnical Commission, 2022.

[Isa24] Isaku, Erblin; Laaber, Christoph; Sartaj, Hassan; Ali, Shaukat; Schwitalla, Thomas and Nygård, Jan F: "LLMs in the Heart of Differential Testing: A Case Study on a Medical Rule Engine". In: *arXiv preprint arXiv:2404.03664* (2024).

[Jea19] Jeannotte, Bryer and Tekeoglu, Ali: "Artorias: IoT security testing framework". In: *2019 26th International Conference on Telecommunications (ICT)*. IEEE. 2019, pp. 233–237.

[Jia24] Jiang, Shihao; Zhang, Yu; Li, Junqiang; Yu, Hongfang; Luo, Long and Sun, Gang: "A Survey of Network Protocol Fuzzing: Model, Techniques and Directions". In: *arXiv preprint arXiv:2402.17394* (2024).

[Jua85] Juang, Biing-Hwang and Rabiner, Lawrence: "A probabilistic distance measure for hidden Markov models". In: *AT&T technical journal* 64.2 (1985), pp. 391–408.

[Jul13] Julisch, Klaus: "Understanding and overcoming cyber security anti-patterns". In: *Computer Networks* 57.10 (2013), pp. 2206–2211.

[Kee11] Keesman, Karel J: System identification: an introduction. Springer Science & Business Media, 2011.

[Kha21] Khalifa, Yassin; Mandic, Danilo and Sejdić, Ervin: "A review of Hidden Markov models and Recurrent Neural Networks for event detection and localization in biomedical signals". In: *Information Fusion* 69 (2021), pp. 52–72.

[Kho11] Khoury, Nidal; Zavarsky, Pavol; Lindskog, Dale and Ruhl, Ron: "An analysis of black-box web application security scanners against stored SQL injection". In: *Third International Conference on Privacy, Security, Risk and Trust and Third International Conference on Social Computing*. IEEE. 2011, pp. 1095–1101.

[Kim20] Kim, SungJin; Cho, Jaeik; Lee, Changhoon and Shon, Taeshik: "Smart seed selection-based effective black box fuzzing for IIoT protocol". In: *The Journal of Supercomputing* 76 (2020), pp. 10140–10154.

[Kim24] Kim, Tae Eun; Choi, Jaeseung; Im, Seongjae; Heo, Kihong and Cha, Sang Kil: "Evaluating Directed Fuzzers: Are We Heading in the Right Direction?" In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 316–337.

[Kle18] Klees, George; Ruef, Andrew; Cooper, Benji; Wei, Shiyi and Hicks, Michael: "Evaluating fuzz testing". In: *2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 2123–2138.

[Kna14] Knapp, Eric and Langill, Joel Thomas: Industrial Network Security: Securing critical infrastructure networks for smart grid, SCADA, and other Industrial Control Systems. Syngress, 2014.

[Kna24] Knapp, Eric D: Industrial Network Security: Securing critical infrastructure networks for smart grid, SCADA, and other Industrial Control Systems. Elsevier, 2024.

[Koh20a] Kohl, Moshe and Oberman, Shlomi: Ripple 20 - CVE-2020-11896 RCE CVE-2020-11898 Info Leak. Tech. rep. JSOF Research Lab, 2020. URL: https://www.jsof-tech.com/wp-content/uploads/2020/06/JSOF_Ripple20_Technical_Whitepaper_June20.pdf (visited on 05/31/2021).

[Koh20b] Kohl, Moshe; Schön, Ariel and Oberman, Shlomi: Ripple 20 - CVE-2020-11901. Tech. rep. JSOF Research Lab, 2020. URL: https://www.jsof-tech.com/wp-content/uploads/2020/08/Ripple20_CVE-2020-11901-August20.pdf (visited on 05/31/2021).

[Kou19] Kouwe, Erik van der; Heiser, Gernot; Andriesse, Dennis; Bos, Herbert and Giuffrida, Cristiano: "SoK: Benchmarking flaws in systems security". In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 310–325.

[Küh23] Kühn, Maximilian: "Anwendungsfälle von maschinellem Lernen für das Fuzzing". Supervised by Anne Borcherding and Mark Giraud. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[Lae18] Laeufer, Kevin; Koenig, Jack; Kim, Donggyu; Bachrach, Jonathan and Sen, Koushik: "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[Lan21] Lan, Huiying and Sun, Yanbin: "Review on fuzz testing for protocols in industrial control systems". In: *Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE. 2021, pp. 433–438.

[Lem18] Lemieux, Caroline and Sen, Koushik: "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 2018, pp. 475–485.

[Li16] Li, Wei: "Automatic generation of attack vectors for stored-xss". In: *Revista Ibérica de Sistemas e Tecnologias de Informação* E13 (2016), p. 1.

[Li18a] Li, Yin; Sun, Zhi-Guang and Jiang, Ting-Ting: "An automated test suite generating approach for stateful web services". In: *Software Analysis, Testing, and Evolution: 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings 8*. Springer. 2018, pp. 185–201.

[Li18b] Li, Yuling and Guldenmund, Frank W: "Safety management systems: A broad overview of the literature". In: *Safety science* 103 (2018), pp. 94–123.

[Li21] Li, Yuwei; Ji, Shouling; Chen, Yuan; Liang, Sizhuang; Lee, Wei-Han; Chen, Yueyao; Lyu, Chenyang; Wu, Chunming; Beyah, Raheem; Cheng, Peng et al.: "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers". In: *30th USENIX Security Symposium*. 2021, pp. 2777–2794.

[Lim19] Lim, Hyun-Kyo; Kim, Ju-Bong; Heo, Joo-Seong; Kim, Kwihoon; Hong, Yong-Geun and Han, Youn-Hee: "Packet-based network traffic classification using deep learning". In: *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. IEEE. 2019, pp. 046–051.

[Lin06] Lin, Zhenjiang; King, Irwin and Lyu, Michael R: "Pagesim: A novel link-based similarity measure for the world wide web". In: *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06)*. IEEE. 2006, pp. 687–693.

[Lin17] Lin, Jun-Wei; Wang, Farn and Chu, Paul: "Using semantic similarity in crawling-based web application testing". In: *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 138–148.

[Lin21] Lin, Pei-Yi; Tien, Chia-Wei; Huang, Ting-Chun and Tien, Chin-Wei: "ICPFuzzer: proprietary communication protocol fuzzing by using machine learning and feedback strategies". In: *Cybersecurity* 4 (2021), pp. 1–15.

[Liu10] Liu, Yang; Park, Jeehae; Dahmen, Karin; Chemla, Yann and Ha, Taekjip: "A comparative study of multivariate and univariate hidden Markov modelings in time-binned single-molecule FRET data analysis". In: *The Journal of Physical Chemistry B* 114.16 (2010), pp. 5386–5403.

[Liu22] Liu, Dongge; Pham, Van-Thuan; Ernst, Gidon; Murray, Toby and Rubinstein, Benjamin IP: "State selection algorithms and their impact on the performance of stateful network protocol fuzzing". In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 720–730.

[Liu24a] Liu, Hangtian; Gan, Shuitao; Zhang, Chao; Gao, Zicong; Zhang, Hongqi; Wang, Xiangzhi and Gao, Guangming: "LABRADOR: Response Guided Directed Fuzzing for Black-box IoT Devices". In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society. 2024, pp. 127–127.

[Liu24b] Liu, Kaibo; Liu, Yiyang; Chen, Zhenpeng; Zhang, Jie; Han, Yudong; Ma, Yun; Li, Ge and Huang, Gang: "LLM-Powered Test Case Generation for Detecting Tricky Bugs". In: *arXiv preprint arXiv:2404.10304* (2024).

[Liy24] Liyanage, Danushka; Lee, Seongmin; Tantithamthavorn, Chakkrit and Böhme, Marcel: "Extrapolating Coverage Rate in Greybox Fuzzing". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–12.

[Lju10] Ljung, Lennart: "Perspectives on system identification". In: *Annual Reviews in Control* 34.1 (2010), pp. 1–12.

[LLV24] LLVM project: libFuzzer – a library for coverage-guided fuzz testing. Aug. 7, 2024. url: https://llvm.org/docs/LibFuzzer.html (visited on 07/08/2024).

[Lot20] Lotfollahi, Mohammad; Jafari Siavoshani, Mahdi; Shirali Hossein Zade, Ramin and Saberian, Mohammdsadegh: "Deep packet: A novel approach for encrypted traffic classification using deep learning". In: *Soft Computing* 24.3 (2020), pp. 1999–2012.

[Lv21] Lv, Wanyou; Xiong, Jiawen; Shi, Jianqi; Huang, Yanhong and Qin, Shengchao: "A deep convolution generative adversarial networks based fuzzing framework for industry control protocols". In: *Journal of Intelligent Manufacturing* 32 (2021), pp. 441–457.

[Ma24] Ma, Xiaoyue; Luo, Lannan and Zeng, Qiang: "From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices". In: *33rd USENIX Security Symposium*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4783–4800.

[Mae22] Maesschalck, Sam; Giotsas, Vasileios; Green, Benjamin and Race, Nicholas: "Don't get stung, cover your ICS in honey: How do honeypots fit within industrial control system security". In: *Computers & Security* 114 (2022), p. 102598.

[Mag17] Maghraby, Reham Taher El; Abd Elazim, Nada Mostafa and Bahaa-Eldin, Ayman M: "A survey on deep packet inspection". In: *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. IEEE. 2017, pp. 188–197.

[Mah20] Mahesh, Batta: "Machine learning algorithms-a review". In: *International Journal of Science and Research (IJSR)* 9.1 (2020), pp. 381–386.

[Mah21] Mahbooba, Basim; Timilsina, Mohan; Sahal, Radhya and Serrano, Martin: "Explainable artificial intelligence (XAI) to enhance trust management in intrusion detection systems using decision tree model". In: *Complexity* 2021.1 (2021), p. 6634811.

[Mai14] Maier, Alexander: "Online passive learning of timed automata for cyber-physical production systems". In: *12th International Conference on Industrial Informatics (INDIN)*. IEEE. 2014, pp. 60–66.

[Mak15] Makino, Yuma and Klyuev, Vitaly: "Evaluation of web vulnerability scanners". In: *8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 1. IEEE. 2015, pp. 399–402.

[Mal23] Mallissery, Sanoop and Wu, Yu-Sung: "Demystify the fuzzing methods: A comprehensive survey". In: *ACM Computing Surveys* 56.3 (2023), pp. 1–38.

[Man19] Manès, Valentin; Han, HyungSeok; Han, Choongwoo; Cha, Sang Kil; Egele, Manuel; Schwartz, Edward and Woo, Maverick: "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[Man47] Mann, Henry and Whitney, Donald: "On a test of whether one of two random variables is stochastically larger than the other". In: *The annals of mathematical statistics* (1947), pp. 50–60.

[Mao16] Mao, Xiaojiao; Shen, Chunhua and Yang, Yu-Bin: "Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections". In: *Advances in neural information processing systems* 29 (2016).

[Mar19] Marali, Mounesh; Sudarsan, Sithu and Gogioneni, Ashok: "Cyber security threats in industrial control systems and protection". In: *2019 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. IEEE. 2019, pp. 1–7.

[Mar96] Mari, Joao Fernando; Fohr, Dominique and Junqua, Jean-Claude: "A second-order HMM for high performance word and phoneme-based continuous speech recognition". In: *International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*. Vol. 1. IEEE. 1996, pp. 435–438.

[McA08] McAllister, Sean; Kirda, Engin and Kruegel, Christopher: "Leveraging user interactions for in-depth testing of web applications". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2008, pp. 191–210.

[Mes08] Mesbah, Ali; Bozdag, Engin and Van Deursen, Arie: "Crawling Ajax by inferring user interface state changes". In: *2008 eighth international conference on web engineering*. IEEE. 2008, pp. 122–134.

[Met21] Metzman, Jonathan; Szekeres, László; Simon, Laurent; Sprabery, Read and Arya, Abhishek: "Fuzzbench: an open fuzzer benchmarking platform and service". In: *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2021, pp. 1393–1403.

[Met23] Metasploit, Rapid7 -: The world's most used penetration testing framework. Dec. 22, 2023. url: https://www.metasploit.com/ (visited on 08/27/2024).

[Mey13] Meyer, Joachim; Noguera, Juanjo; Hübner, Michael; Stewart, Rodney and Becker, Juergen: "Embedded systems start-up under timing constraints on modern FPGAs". In: *Embedded Systems Design with FPGAs* (2013), pp. 149–172.

[Mil20] Miller, Barton; Zhang, Mengxiao and Heymann, Elisa R: "The relevance of classic fuzz testing: Have we solved this one?" In: *IEEE Transactions on Software Engineering* 48.6 (2020), pp. 2028–2039.

[Mir18] Mirsky, Yisroel; Doitshman, Tomer; Elovici, Yuval and Shabtai, Asaf: "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection". In: *Network and Distributed Systems Security (NDSS) Symposium*. 2018.

[MIT18] MITRE: CAPEC-540: Overread Buffers. July 31, 2018. URL: https://capec.mitre.org/data/definitions/540.html (visited on 05/02/2024).

[MIT24] MITRE: CWE-130: Improper Handling of Length Parameter Inconsistency. Feb. 29, 2024. URL: https://cwe.mitre.org/data/definitions/130.html (visited on 05/02/2024).

[Mit97] Mitchell, Tom: Machine learning. Vol. 1. 9. McGraw-hill New York, 1997.

[Moc87a] Mockapetris, Paul: Domain names - concepts and facilities. STD 13. RFC Editor, Nov. 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt.

[Moc87b] Mockapetris, Paul: Domain names - implementation and specification. STD 13. RFC Editor, Nov. 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt.

[Mor21] Morawetz, Martin: "Adaptive Black Box Fuzzing of Industrial Network Protocols". Supervised by Anne Borcherding and Steffen Pfrang. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[Mor24] Morales-Gonzalez, Christopher; Harper, Matthew; Cash, Michael; Luo, Lan; Ling, Zhen; Sun, Qun and Fu, Xinwen: "On Building Automation System security". In: *High-Confidence Computing* (2024), p. 100236.

[Mou23] Moura, Giovane and Heidemann, John: "Vulnerability Disclosure Considered Stressful". In: *ACM SIGCOMM Computer Communication Review* 53.2 (2023), pp. 2–10.

[Mue18] Muench, Marius; Stijohann, Jan; Kargl, Frank; Francillon, Aurélien and Balzarotti, Davide: "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices". In: *Network and Distributed System Security (NDSS) Symposium*. 2018.

[Mum21] Mumper, Robert: "Analyse von modellbasierten Methoden zum Zweck der Effizienzsteigerung von Blackbox-Fuzzing". Supervised by Anne Borcherding and Mark Giraud. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[Naf18] Nafees, Tayyaba; Coull, Natalie; Ferguson, Ian and Sampson, Adam: "Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities)". In: *24th Conference on Pattern Languages of Programs*. The Hillside Group. 2018, p. 23.

[Nat21] Natella, Roberto and Pham, Van-Thuan: "Profuzzbench: A benchmark for stateful protocol fuzzing". In: *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 2021, pp. 662–665.

[Nat22] Natella, Roberto: "Stateafl: Greybox fuzzing for stateful network servers". In: *Empirical Software Engineering* 27.7 (2022), p. 191.

[Nic23] Nicolae, Maria-Irina; Eisele, Max and Zeller, Andreas: "Revisiting neural program smoothing for fuzzing". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 133–145.

[Oli13] Oliver, Jonathan; Cheng, Chun and Chen, Yanggui: "TLSH–a locality sensitive hash". In: *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE. 2013, pp. 7–13.

[OPC22] OPC Foundation: OPC UA 10000. Standard. Scottsdale, AZ, USA: OPC Foundation, 2022.

[Öst20] Österlund, Sebastian; Razavi, Kaveh; Bos, Herbert and Giuffrida, Cristiano: "ParmeSan: Sanitizer-guided greybox fuzzing". In: *29th USENIX Security Symposium*. 2020, pp. 2289–2306.

[Oué24] Ouédraogo, Wendkûuni; Kaboré, Kader; Tian, Haoye; Song, Yewei; Koyuncu, Anil; Klein, Jacques; Lo, David and Bissyandé, Tegawendé: "Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation". In: *arXiv preprint arXiv:2407.00225* (2024).

[OWA20] OWASP: Overview of Vulnerability Scanning Tools. 2020. URL: https://owasp.org/www-community/Vulnerability_Scanning_Tools (visited on 03/07/2024).

[OWA21] OWASP: OWASP Top 10 - 2021. Tech. rep. OWASP, 2021. URL: https://owasp.org/Top10/.

[OWA24a] OWASP: OWASP Benchmark Project. 2024. URL: https://owasp.org/www-project-benchmark/ (visited on 08/27/2024).

[OWA24b] OWASP: Vulnerability Disclosure Cheat Sheet. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html (visited on 08/27/2024).

[Paa21] Paassen, David; Surminski, Sebastian; Rodler, Michael and Davi, Lucas: "My fuzzer beats them all! developing a framework for fair evaluation and comparison of fuzzers". In: *26th European Symposium on Research in Computer Security (ESORICS 2021)*. Springer. 2021, pp. 173–193.

[Pac18] Pacheco, Fannia; Exposito, Ernesto; Gineste, Mathieu; Baudoin, Cedric and Aguilar, Jose: "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey". In: *IEEE Communications Surveys & Tutorials* 21.2 (2018), pp. 1988–2014.

[Pan16] Panzner, Maximilian and Cimiano, Philipp: "Comparing hidden markov models and long short term memory neural networks for learning action representations". In: *Machine Learning, Optimization, and Big Data: Second International Workshop (MOD 2016)*. Springer. 2016, pp. 94–105.

[Pet20] Pettersen Gould, Kenneth and Bieder, Corinne: "Safety and security: the challenges of bringing them together". In: *The Coupling of Safety and Security: Exploring Interrelations in Theory and Practice* (2020), pp. 1–8.

[Pfe22] Pferscher, Andrea and Aichernig, Bernhard K: "Stateful black-box fuzzing of bluetooth devices using automata learning". In: *NASA Formal Methods Symposium*. Springer. 2022, pp. 373–392.

[Pfr17] Pfrang, Steffen; Meier, David and Kautz, Valentin: "Towards a modular security testing framework for industrial automation and control systems: Isutest". In: *22nd International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2017, pp. 1–5.

[Pfr18] Pfrang, Steffen; Meier, David; Friedrich, Michael and Beyerer, Jürgen: "Advancing Protocol Fuzzing for Industrial Automation and Control Systems". In: *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*. 2018, pp. 570–580.

[Pfr19a] Pfrang, Steffen and Borcherding, Anne: "Security Testing für industrielle Automatisierungskomponenten: Ein Framework, sein Einsatz und Ergebnisse am Beispiel von Profinet-Buskopplern". In: *IT-Sicherheit als Voraussetzung für eine erfolgreiche Digitalisierung - Tagungsband zum 16. Deutschen IT-Sicherheitskongress*. SecuMedia Verlag, 2019.

[Pfr19b] Pfrang, Steffen; Borcherding, Anne; Meier, David and Beyerer, Jürgen: "Automated security testing for web applications on industrial automation and control systems". In: *at-Automatisierungstechnik* 67.5 (2019), pp. 383–401.

[Pfr19c] Pfrang, Steffen; Giraud, Mark; Borcherding, Anne; Meier, David and Beyerer, Jürgen: "Design of an Example Network Protocol for Security Tests Targeting Industrial Automation Systems." In: *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*. 2019, pp. 727–738.

[Pfr23] Pfrang, Steffen and Borcherding, Anne: ISuTest®: Automated vulnerability assessment for industrial automation components. Dec. 19, 2023. url: https://www.iosb.fraunhofer.de/en/projects-and-products/isutest.html (visited on 05/07/2024).

[Pha20] Pham, Van-Thuan; Böhme, Marcel and Roychoudhury, Abhik: "AFLNet: a greybox fuzzer for network protocols". In: *13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 460–465.

[Pha24] Pham, Van-Hau; Chuong, Nguyen Phuc; Thai, Pham Thanh; Duy, Phan The et al.: "A Coverage-guided Fuzzing Method for Automatic Software Vulnerability Detection using Reinforcement Learning-enabled Multi-Level Input Mutation". In: *IEEE Access* (2024).

[Pin21] Pintelas, Emmanuel; Livieris, Ioannis and Pintelas, Panagiotis: "A convolutional autoencoder topology for classification in high-dimensional noisy image datasets". In: *Sensors* 21.22 (2021).

[Ple24] Plein, Laura; Ouédraogo, Wendkûuni; Klein, Jacques and Bissyandé, Tegawendé: "Automatic generation of test cases based on bug reports: a feasibility study with large language models". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024, pp. 360–361.

[Pon22] Poncelet, Clément; Sagonas, Konstantinos and Tsiftes, Nicolas: "So Many Fuzzers, So Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay) Stack". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[Pop16] Popescu, Doru Anastasiu and Nicolae, Dragoș: "Determining the similarity of two web applications using the edit distance". In: *Proceedings of the 6th International Workshop Soft Computing Applications (SOFA 2014)*. Springer. 2016, pp. 681–690.

[Pos81a] Postel, Jon: Internet Protocol. STD. RFC Editor, 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[Pos81b] Postel, Jon: Transmission Control Protocol. STD. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[Pos85] Postel, Jon and Reynolds, Joyce: File Transfer Protocol. STD. RFC Editor, Oct. 1985. URL: http://www.rfc-editor.org/rfc/rfc959.txt.

[Pot23] Pottebaum, Jens; Rossel, Jost; Somorovsky, Juraj; Acar, Yasemin; Fahr, René; Cabarcos, Patricia Arias; Bodden, Eric and Grässler, Iris: "Re-Envisioning Industrial Control Systems Security by Considering Human Factors as a Core Element of Defense-in-Depth". In: *European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2023, pp. 379–385.

[Rab86] Rabiner, Lawrence and Juang, Biinghwang: "An introduction to hidden Markov models". In: *IEEE ASSP magazine* 3.1 (1986), pp. 4–16.

[Raf05] Raffelt, Harald; Steffen, Bernhard and Berg, Therese: "Learnlib: A library for automata learning and experimentation". In: *10th international workshop on Formal methods for industrial critical systems*. 2005, pp. 62–71.

[Ras24] Rasool, Zafaryab; Barnett, Scott; Willie, David; Kurniawan, Stefanus; Balugo, Sherwin; Thudumu, Srikanth and Abdelrazek, Mohamed: "LLMs for Test Input Generation for Semantic Applications". In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 2024, pp. 160–165.

[Raw17] Rawat, Sanjay; Jain, Vivek; Kumar, Ashish; Cojocar, Lucian; Giuffrida, Cristiano and Bos, Herbert: "VUzzer: Application-aware evolutionary fuzzing". In: *Network and Distributed System Security (NDSS) Symposium*. Vol. 17. 2017, pp. 1–14.

[Ren19] Rennhard, Marc; Esposito, Damiano; Ruf, Lukas and Wagner, Arno: "Improving the effectiveness of web application vulnerability scanning". In: *International Journal on Advances in Internet Technology* 12.1/2 (2019), pp. 12–27.

[Ric16] Rice, Randall et al.: Certified Tester Advanced Level Syllabus – Security Tester version 2016. 2016.

[Rui15] Ruiter, Joeri De and Poll, Erik: "Protocol state fuzzing of TLS implementations". In: *24th USENIX Security Symposium*. 2015, pp. 193–206.

[Rus16] Russell, Stuart and Norvig, Peter: Artificial intelligence: a modern approach. Pearson, 2016.

[Saa24] Saad, Elie and Mitchell, Rick: OWASP Web Security Testing Guide. 2024. URL: https://owasp.org/www-project-web-security-testing-guide/ (visited on 08/27/2024).

[Sal20] Salls, Christopher; Machiry, Aravind; Doupe, Adam; Shoshitaishvili, Yan; Kruegel, Christopher and Vigna, Giovanni: "Exploring abstraction functions in fuzzing". In: *Conference on Communications and Network Security (CNS)*. IEEE. 2020, pp. 1–9.

[Sal21] Salem, Hamad Al and Song, Jia: "Grammar-based Fuzzing Tool Using Markov Chain Model to Generate New Fuzzing Inputs". In: *International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE. 2021, pp. 1924–1930.

[Sam22] Samanis, Emmanouil; Gardiner, Joseph and Rashid, Awais: "Sok: A taxonomy for contrasting industrial control systems asset discovery tools". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 2022, pp. 1–12.

[San21a] Santos, Daniel dos; Dashevskyi, Stanislav; Amri, Amine; Wetzels, Jos; Karas, Asaf; Menashe, Shachar and Vozniuk, Denys: INFRA:HALT - Jointly discovering and mitigating large-scale OT vulnerabilities. Tech. rep. Forescout Research Labs and JFrog Security Research, 2021.

[San21b] Santos, Daniel dos; Dashevskyi, Stanislav; Amri, Amine; Wetzels, Jos; Oberman, Shlomi and Kol, Moshe: NAME:WRECK - Breaking and Fixing DNS Implementations. Tech. rep. Forescout Research Labs and JSOF, 2021.

[San21c] Santos, Daniel dos; Dashevskyi, Stanislav; Wetzels, Jos and Amri, Amine: Amnesia:33 - How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices. Tech. rep. Forescout Research Labs, 2021. URL: https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-iniot-ot-and-it-devices/ (visited on 05/31/2021).

[Sas21] Sasi, Ashwathi; Hariprasad, K.V.; Cherian, Season; Sharma, Ayushi; Narayanan, Jayasree and Pavithran, Vipin: "R0fuzz: A Collaborative Fuzzer for ICS Protocols". In: *12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE. 2021, pp. 1–5.

[Sca08] Scarfone, Karen; Souppaya, Murugiah; Cody, Amanda and Orebaugh, Angela: SP 800-115. Technical Guide to Information Security Testing and Assessment. Tech. rep. National Institute of Standards and Technology, 2008.

[Sch24] Schloegel, Moritz; Bars, Nils; Schiller, Nico; Bernhard, Lukas; Scharnowski, Tobias; Crump, Addison; Ale-Ebrahim, Arash; Bissantz, Nicolai; Muench, Marius and Holz, Thorsten: "SoK: Prudent Evaluation Practices for Fuzzing". In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society. 2024, pp. 137–137.

[Ser19] Seri, Ben; Vishnepolsky, Gergory and Zusman, Dor: Urgent/11 - Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. Tech. rep. ARMIS, 2019. url: https://info.armis.com/rs/645-PDC-047/images/Urgent11%20Technical%20White%20Paper.pdf (visited on 05/31/2021).

[Sha21] Shang, Wenli; Zhang, Guanyu; Wang, Tianyu and Zhang, Rui: "A test cases generation method for industrial control protocol test". In: *Scientific Programming* 2021.1 (2021), p. 6611732.

[She19] She, Dongdong; Pei, Kexin; Epstein, Dave; Yang, Junfeng; Ray, Baishakhi and Jana, Suman: "Neuzz: Efficient fuzzing with neural program smoothing". In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2019, pp. 803–817.

[Sli19] Slivkins, Aleksandrs et al.: "Introduction to multi-armed bandits". In: *Foundations and Trends® in Machine Learning* 12.1-2 (2019), pp. 1–286.

[Smi20] Smith, Connie U: "Software performance antipatterns in cyber-physical systems". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 173–180.

[Spe19] Sperl, Philip and Böttinger, Konstantin: "Side-channel aware fuzzing". In: *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security*. Springer. 2019, pp. 259–278.

[Spe24] Specht, Felix and Otto, Jens: "Efficient Machine Learning-based Security Monitoring and Cyberattack Classification of Encrypted Network Traffic in Industrial Control Systems". In: *29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2024.

[Spr21] Spring, Jonathan; Hatleback, Eric; Householder, Allen; Manion, Art and Shick, Deana: "Time to Change the CVSS?" In: *IEEE Security & Privacy* 19.2 (2021), pp. 74–78.

[Sto23] Stouffer, Keith; Stouffer, Keith; Pease, Michael; Tang, CheeYee; Zimmerman, Timothy; Pillitteri, Victoria; Lightman, Suzanne; Hahn, Adam; Saravia, Stephanie; Sherule, Aslam et al.: Guide to operational technology (ot) security. US Department of Commerce, National Institute of Standards and Technology, 2023.

[Su24] Su, Kai; Giraud, Mark; Borcherding, Anne; Krautter, Jonas; Nenninger, Philipp and Tahoori, Mehdi: "Fuzz Wars: The Voltage Awakens–Voltage-Guided Blackbox Fuzzing on FPGAs". In: *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE. 2024, pp. 1–7.

[Suc15] Sucar, Luis Enrique: "Probabilistic graphical models". In: *Advances in Computer Vision and Pattern Recognition* 10.978 (2015).

[Sut13] Suteva, Natasa; Zlatkovski, Dragi and Mileva, Aleksandra: "Evaluation and testing of several free/open source web vulnerability scanners". In: *Proceedings of the Tenth International Conference on Informatics and Information Technology* (2013).

[Sut16] Suthaharan, Shan: "Support Vector Machine". In: *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*. Boston, MA: Springer US, 2016, pp. 207–235.

[Sut18] Sutton, Richard and Barto, Andrew G: Reinforcement learning: An introduction. MIT press, 2018.

[Tad20] Tadayon, Manie and Pottie, Greg: "Comparative analysis of the hidden markov model and LSTM: A simulative approach". In: *arXiv preprint arXiv:2008.03825* (2020).

[Ten23] Tenable: SCADA Family for Nessus. Dec. 22, 2023. url: https://www.tenable.com/plugins/nessus/families/SCADA (visited on 08/27/2024).

[Tri23] Trickel, Erik; Pagani, Fabio; Zhu, Chang and Dresel, Lukas: "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities". In: IEEE. 2023.

[Tru18] Trubiani, Catia; Bran, Alexander; Hoorn, André van; Avritzer, Alberto and Knoche, Holger: "Exploiting load testing and profiling for performance antipattern detection". In: *Information and Software Technology* 95 (2018), pp. 329–345.

[Tum19] Tuma, Katja; Hosseini, Danial; Malamas, Kyriakos and Scandariato, Riccardo: "Inspection guidelines to identify security design flaws". In: *Proceedings of the 13th European Conference on Software Architecture-Volume 2*. 2019, pp. 116–122.

[Urb22] Urbano, Luigi; Perrone, Gaetano and Romano, Simon Pietro: "Reinforced wavsep: a benchmarking platform for web application vulnerability scanners". In: *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*. IEEE. 2022, pp. 1–6.

[Vaa22] Vaandrager, Frits; Garhewal, Bharat; Rot, Jurriaan and Wissmann, Thorsten: "A new approach for active automata learning based on apartness". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 223–243.

[Var00] Vargha, András and Delaney, Harold D: "A critique and improvement of the CL common language effect size statistics of McGraw and Wong". In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[Vas17] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan; Kaiser, Łukasz and Polosukhin, Illia: "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[Veg17] Vega, Esteban Alejandro Armas; Orozco, Ana Lucila Sandoval and Villalba, Luis Javier García: "Benchmarking of pentesting tools". In: *International Journal of Computer and Information Engineering* 11.5 (2017), pp. 602–605.

[Vii08] Viide, Joachim; Helin, Aki; Laakso, Marko; Pietikäinen, Pekka; Seppänen, Mika; Halunen, Kimmo; Puuperä, Rauli and Röning, Juha: "Experiences with Model Inference Assisted Fuzzing". In: *WOOT* 2 (2008), pp. 1–2.

[Vin10] Vincent, Pascal; Larochelle, Hugo; Lajoie, Isabelle; Bengio, Yoshua; Manzagol, Pierre-Antoine and Bottou, Léon: "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion". In: *Journal of machine learning research* 11.12 (2010).

[Wan17] Wang, Junjie; Chen, Bihuan; Wei, Lei and Liu, Yang: "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *IEEE Symposium on Security and Privacy (S&P)*. 2017, pp. 579–594.

[Wan19] Wang, Jinghan; Duan, Yue; Song, Wei; Yin, Heng and Song, Chengyu: "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019, pp. 1–15.

[Wan20a] Wang, Yan; Jia, Peng; Liu, Luping; Huang, Cheng and Liu, Zhonglin: "A systematic review of fuzzing based on machine learning techniques". In: *PloS one* 15.8 (2020), e0237749.

[Wan20b] Wang, Yanhao; Jia, Xiangkun; Liu, Yuwei; Zeng, Kyle; Bao, Tiffany; Wu, Dinghao and Su, Purui: "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". In: *Network and Distributed System Security (NDSS) Symposium*. 2020.

[Wan21a] Wang, Haomin and Li, Wei: "DDosTC: A transformer-based network attack detection hybrid mechanism in SDN". In: *Sensors* 21.15 (2021), p. 5047.

[Wan21b] Wang, Jinghan; Song, Chengyu and Yin, Heng: "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: *Network and Distributed System Security (NDSS) Symposium*. 2021.

[Wan21c] Wang, Mingzhe; Liang, Jie; Zhou, Chijin; Chen, Yuanliang; Wu, Zhiyong and Jiang, Yu: "Industrial Oriented Evaluation of Fuzzing Techniques". In: *14th Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 306–317.

[Wan21d] Wang, Wenhui: "Finding and Exploiting Vulnerabilities in Embedded TCP/IP Stacks". MA thesis. University of Twente, 2021.

[Wan24] Wang, Jincheng; Yu, Le and Luo, Xiapu: "Llmif: Augmented large language model for fuzzing iot devices". In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society. 2024, pp. 196–196.

[Wed15] Wedgbury, Adam and Jones, Kevin: "Automated asset discovery in industrial control systems-exploring the problem". In: *3rd International Symposium for ICS & SCADA Cyber Security Research 2015 (ICS-CSR 2015)*. BCS Learning & Development. 2015.

[Wei19] Weiche, Albrecht: "Entwicklung eines Proxys zur Integration und Verbesserung von Web Security Scannern für ISuTest 2". Supervised by Steffen Pfrang and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2019.

[Wei24] Wei, Xiaoyan; Yan, Zheng and Liang, Xueqin: "A survey on fuzz testing technologies for industrial control protocols". In: *Journal of Network and Computer Applications* (2024), p. 104020.

[Wer14] Wert, Alexander; Oehler, Marius; Heger, Christoph and Farahbod, Roozbeh: "Automatic detection of performance anti-patterns in inter-component communications". In: *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*. 2014, pp. 3–12.

[Wol22] Wolff, Dylan; Böhme, Marcel and Roychoudhury, Abhik: "Explainable fuzzer evaluation". In: *arXiv preprint arXiv:2212.09519* (2022).

[Woo13] Woo, Maverick; Cha, Sang Kil; Gottlieb, Samantha and Brumley, David: "Scheduling black-box mutational fuzzing". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 511–522.

[Wri10] Wright, Charles; Connelly, Christopher; Braje, Timothy; Rabek, Jesse; Rossey, Lee and Cunningham, Robert K: "Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security". In: *Recent Advances in Intrusion Detection: 13th International Symposium (RAID 2010)*. Springer. 2010, pp. 218–237.

[Wu17] Wu, Jianxin: "Introduction to convolutional neural networks". In: *National Key Lab for Novel Software Technology. Nanjing University. China* 5.23 (2017), p. 495.

[Wu22a] Wu, Mingyuan; Jiang, Ling; Xiang, Jiahong; Huang, Yanwei; Cui, Heming; Zhang, Lingming and Zhang, Yuqun: "One fuzzing strategy to rule them all". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1634–1645.

[Wu22b] Wu, Zihan; Zhang, Hong; Wang, Penghai and Sun, Zhibo: "RTIDS: A robust transformer-based approach for intrusion detection system". In: *IEEE Access* 10 (2022), pp. 64375–64387.

[Xav22] Xavier, Midhun; Dubinin, Victor; Patil, Sandeep and Vyatkin, Valeriy: "Process mining in industrial control systems". In: *20th International Conference on Industrial Informatics (INDIN)*. IEEE. 2022, pp. 1–6.

[Xia24] Xia, Chunqiu Steven; Paltenghi, Matteo; Le Tian, Jia; Pradel, Michael and Zhang, Lingming: "Fuzz4all: Universal fuzzing with large language models". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[Xue24] Xue, Zhiyi; Li, Liangguo; Tian, Senyue; Chen, Xiaohong; Li, Pingping; Chen, Liangyu; Jiang, Tingting and Zhang, Min: "LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2024).

[Yam21] Yamamoto, Koji: "Efficient penetration of API sequences to test stateful RESTful services". In: *IEEE International Conference on Web Services (ICWS)*. 2021, pp. 734–740.

[Yan20] Yandrapally, Rahulkrishna; Stocco, Andrea and Mesbah, Ali: "Near-duplicate detection in web app model inference". In: *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 2020, pp. 186–197.

[Yan24] Yang, Lin; Yang, Chen; Gao, Shutao; Wang, Weijing; Wang, Bo; Zhu, Qihao; Chu, Xiao; Zhou, Jianyi; Liang, Guangtai; Wang, Qianxiang et al.: "An Empirical Study of Unit Test Generation with Large Language Models". In: *arXiv preprint arXiv:2406.18181* (2024).

[Ye19] Ye, Xun and Hong, Seung Ho: "Toward industry 4.0 components: Insights into and implementation of asset administration shells". In: *IEEE Industrial Electronics Magazine* 13.1 (2019), pp. 13–25.

[Yin23] Yin, Zijing; Xu, Yiwen; Ma, Fuchen; Gao, Haohao; Qiao, Lei and Jiang, Yu: "Scanner++: Enhanced Vulnerability Detection of Web Applications with Attack Intent Synchronization". In: *ACM Transactions on Software Engineering and Methodology* 32.1 (2023), pp. 1–30.

[Yue20] Yue, Tai; Wang, Pengfei; Tang, Yong; Wang, Enze; Yu, Bo; Lu, Kai and Zhou, Xu: "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit". In: *29th USENIX Security Symposium*. 2020, pp. 2307–2324.

[Yun22] Yun, Joobeom; Rustamov, Fayozbek; Kim, Juhwan and Shin, Youngjoo: "Fuzzing of embedded systems: A survey". In: *ACM Computing Surveys* 55.7 (2022), pp. 1–33.

[Zal16] Zalewski, Michał: American Fuzzy Lop. Tech. rep. 2016.

[Zan17] Zander, Justyna; Schieferdecker, Ina and Mosterman, Pieter J: Model-based testing for embedded systems. CRC press, 2017.

[Zen19] Zeng, Yi; Gu, Huaxi; Wei, Wenting and Guo, Yantao: "$Deep - Full - Range$: a deep learning based network encrypted traffic classification and intrusion detection framework". In: *IEEE Access* 7 (2019), pp. 45182–45190.

[Zha19] Zhao, Hui; Li, Zhihui; Wei, Hansheng; Shi, Jianqi and Huang, Yanhong: "SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective". In: *12th Conference on software testing, validation and verification (ICST)*. IEEE. 2019, pp. 59–67.

[Zha23a] Zhang, Ao; Zhang, Yiying; Xu, Yao; Wang, Cong and Li, Siwei: "Machine Learning-based Fuzz Testing Techniques: A Survey". In: *IEEE Access* (2023).

[Zha23b] Zhang, Tianxiang; Huang, Hui; Lu, Yuliang; Zhu, Kailong and Zhao, Jiazhen: "State-Sensitive Black-Box Web Application Scanning for Cross-Site Scripting Vulnerability Detection". In: *Applied Sciences* 13.16 (2023), p. 9212.

[Zha24a] Zhang, Kunpeng; Zhu, Xiaogang; Xiao, Xi; Xue, Minhui; Zhang, Chao and Wen, Sheng: "SHAPFUZZ: Efficient Fuzzing via Shapley-Guided Byte Selection". In: *Network and Distributed System Security (NDSS) Symposium* (2024).

[Zha24b] Zhang, Xiaohan; Zhang, Cen; Li, Xinghua; Du, Zhengjie; Li, Yuekang; Zheng, Yaowen; Li, Yeting; Mao, Bing; Liu, Yang and Deng, Robert H: "A Survey of Protocol Fuzzing". In: *arXiv preprint arXiv:2401.01568* (2024).

[Zhe22] Zheng, Yaowen and Sun, Limin: "IPSpex: Enabling Efficient Fuzzing via Specification Extraction on ICS Protocol". In: *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*. Vol. 13269. Springer Nature. 2022, p. 356.

[Zhu22] Zhu, Xiaogang; Wen, Sheng; Camtepe, Seyit and Xiang, Yang: "Fuzzing: a survey for roadmap". In: *ACM Computing Surveys (CSUR)* 54.11s (2022), pp. 1–36.

[Zim21] Zimmermann, Simon: "Einsatz eines Zustandserkennungsverfahrens zur Verbesserung der Schwachstellensuche bei Webanwendungen". Supervised by Anne Borcherding and Florian Patzer. Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

Own Publications

This section contains a complete list of publications produced during the course of this doctoral work, as well as the Master's thesis completed prior to this doctoral work. The publications [1], [4], [6] focus on web security with a special focus on OT components, while [11] is concerned with general web security. In [10], [9], [8], [13] various aspects of blackbox and graybox testing of OT components and of Field Programmable Gate Arrays (FPGAs) are addressed. [3] presents a study on vulnerabilities in OT components using ISuTest®, which is presented in more detail on the project's website [12]. The technical report [2] presents an overview of general approaches to security testing.

Beyond the primary focus of this doctoral work, [5], [7] address topics in network protocol design and network intrusion detection, respectively.

[1] Borcherding, Anne: "Security-Testing für Webserver auf industriellen Automatisierungskomponenten". MA thesis. Karlsruhe Institute of Technology (KIT), 2018.

[2] Borcherding, Anne; Goerke, Niklas and Klamroth, Jonas: Prüfmethoden für Security. German. Tech. rep. Kooperationsprojekt Cyberprotect, 2019.

[3] Pfrang, Steffen and Borcherding, Anne: "Security Testing für industrielle Automatisierungskomponenten: Ein Framework, sein Einsatz und Ergebnisse am Beispiel von Profinet-Buskopplern". In: *IT-Sicherheit als Voraussetzung für eine erfolgreiche Digitalisierung - Tagungsband zum 16. Deutschen IT-Sicherheitskongress*. SecuMedia Verlag, 2019.

[4] Pfrang, Steffen; Borcherding, Anne; Meier, David and Beyerer, Jürgen: "Automated security testing for web applications on industrial automation and control systems". In: *at-Automatisierungstechnik* 67.5 (2019), pp. 383–401.

[5] Pfrang, Steffen; Giraud, Mark; Borcherding, Anne; Meier, David and Beyerer, Jürgen: "Design of an Example Network Protocol for Security Tests Targeting Industrial Automation Systems." In: *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*. 2019, pp. 727–738.

[6] Borcherding, Anne; Pfrang, Steffen; Haas, Christian; Weiche, Albrecht and Beyerer, Jürgen: "Helper-in-the-Middle: Supporting Web Application Scanners Targeting Industrial Control Systems". In: *Proceedings of the 17th International Conference on Security and Cryptography (SECRYPT / ICETE 2020)*. SCITEPRESS - Science and Technology Publications, 2020, pp. 27–38.

[7] Borcherding, Anne; Feldmann, Lukas; Karch, Markus; Meshram, Ankush and Beyerer, Jürgen: "Towards a Better Understanding of Machine Learning Based Network Intrusion Detection Systems in Industrial Networks". In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*. 2022.

[8] Borcherding, Anne; Takacs, Philipp and Beyerer, Jürgen: "Cluster Crash: Learning from Recent Vulnerabilities in Communication Stacks". In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*. 2022, pp. 334–344.

[9] Borcherding, Anne; Giraud, Mark; Fitzgerald, Ian and Beyerer, Jürgen: "The Bandit's States: Modeling State Selection for Stateful Network Fuzzing as Multi-armed Bandit Problem". In: *1st International Workshop on Re-design Industrial Control Systems with Security (RICSS)*. 2023, pp. 345–350.

[10] Borcherding, Anne; Morawetz, Martin and Pfrang, Steffen: "Smarter Evolution: Enhancing Evolutionary Black Box Fuzzing with Adaptive Models". In: *Sensors* 23.18 (2023).

[11] Borcherding, Anne; Penkov, Nikolay; Giraud, Mark and Beyerer, Jürgen.: "SWaTEval: An Evaluation Framework for Stateful Web Application Testing". In: *Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP*, INSTICC. SCITEPRESS - Science and Technology Publications, 2023, pp. 430–441.

[12] Pfrang, Steffen and Borcherding, Anne: ISuTest®: Automated vulnerability assessment for industrial automation components. Dec. 19, 2023. url: https://www.iosb.fraunhofer.de/en/projects-and-products/isutest.html (visited on 05/07/2024).

[13] Su, Kai; Giraud, Mark; Borcherding, Anne; Krautter, Jonas; Nenninger, Philipp and Tahoori, Mehdi: "Fuzz Wars: The Voltage Awakens–Voltage-Guided Blackbox Fuzzing on FPGAs". In: *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE. 2024, pp. 1–7.

Patents

This section contains a patent that was developed during this doctoral work. The patent [1], focused on model-based testing, was created in collaboration with Steffen Pfrang and Christian Haas.

[1] Borcherding, Anne; Pfrang, Steffen and Haas, Christian: "Test assembly and method for testing a test object, and computer program for carrying out the method". WO2023144158 A1. 2023.

Published Vulnerabilities

Throughout the course of this doctoral work, several vulnerabilities in OT components were discovered responsibly disclosed. Depending on the vulnerability handling process of the affected manufactures, these vulnerabilities were assigned Common Vulnerabilities and Exposures (CVE) identifiers and subsequently published. The following list includes all vulnerabilities were disclosed and published in this manner upon completion of this dissertation.

[1] CVE: CVE-2018-16994. Denial of service in several fieldbus couplers (severity 7.5). 2018. URL: https://www.cve.org/CVERecord?id=CVE-2018-16994.

[2] CVE: CVE-2019-14753. Buffer overflow in a safety PROFINET gateway (severity 7.5). 2019. URL: https://www.cve.org/CVERecord?id=CVE-2019-14753.

[3] CVE: CVE-2021-21003. Denial of service in industrial switch (severity 5.3). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21003.

[4] CVE: CVE-2021-21004. Cross-site scripting vulnerability in industrial switch (severity 7.4). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21004.

[5] CVE: CVE-2021-21005. Denial of service in industrial switch (severity 7.5). 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-21005.

Supervised Student Theses

This section lists the student theses supervised during this doctoral work. The theses [1], [5], [9] focus on web security, while [2], [4], [7], [8], [10], [11], [12], [13], [14], [15], [16], [17] are concerned with fuzzing. Beyond the primary subject of this doctoral work, [6], [3] are located in the domain of intrusion detection.

[1] Weiche, Albrecht: "Entwicklung eines Proxys zur Integration und Verbesserung von Web Security Scannern für ISuTest 2". Supervised by Steffen Pfrang and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2019.

[2] Dillmann, Joerg: "Systematisch unterstützte Durchführung und Auswertung von Security-Tests mit ISuTest". Supervised by Steffen Pfrang and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[3] Feldmann, Lukas: "A Systematic Approach to Predict the Intrusion Detection Model in Industrial Networks". Supervised by Anne Borcherding, Markus Karch, and Ankush Meshram. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[4] Giraud, Mark Leon: "Design and evaluation of methods for efficient fuzzing of stateful software". Supervised by Christian Haas and Anne Borcherding. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[5] Hofmann, Felix: "Black-Box-Sicherheitsanalyse von zustandsbehafteten Webanwendungen". Supervised by Anne Borcherding and Florian Patzer. Master's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[6] Laly, Yorick: “Analyse der elektrischen Signale zur Angriffserkennung in industriellen Systemen basierend auf maschinellem Lernen”. Supervised by Markus Karch, Ankush Meshram, and Anne Borcherding. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[7] Morawetz, Martin: “Adaptive Black Box Fuzzing of Industrial Network Protocols”. Supervised by Anne Borcherding and Steffen Pfrang. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[8] Mumper, Robert: “Analyse von modellbasierten Methoden zum Zweck der Effizienzsteigerung von Blackbox-Fuzzing”. Supervised by Anne Borcherding and Mark Giraud. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[9] Zimmermann, Simon: “Einsatz eines Zustandserkennungsverfahrens zur Verbesserung der Schwachstellensuche bei Webanwendungen”. Supervised by Anne Borcherding and Florian Patzer. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[10] Fitzgerald, Ian: “Zustandsauswahl für zustandsbehaftetes Fuzzing als Multi-Armed Bandit Problem: Modellierung und Evaluation”. Supervised by Anne Borcherding and Mark Giraud. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2022.

[11] Avci, Deniz Imge: “Using Hidden Markov Models for Blackbox Fuzzing”. Supervised by Anne Borcherding and Mark Giraud. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[12] Blankefort, Tom: “Systematische Evaluation von Fuzzern”. Supervised by Anne Borcherding and Mark Giraud. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[13] Häring, Johannes: “Enlightening the Dark: Estimating Code Coverage during Blackbox Fuzzing”. Supervised by Anne Borcherding. Seminar "KI Systems Engineering". Karlsruhe Institute of Technology (KIT), 2023.

[14] Hartstern, Daniel: “Design and Evaluation of Methods for Improving Graybox Fuzzing by Utilizing Implicit Program State”. Supervised by Mark Giraud and Anne Borcherding. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[15] Kühn, Maximilian: “Anwendungsfälle von maschinellem Lernen für das Fuzzing”. Supervised by Anne Borcherding and Mark Giraud. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2023.

[16] Lautner, Andreas: “Finding States in Black-Box Fuzzing”. Supervised by Christopher Huth, Mark Giraud, and Anne Borcherding. Master’s Thesis. University of Applied Sciences Esslingen, 2023.

[17] Su, Kai: “Using Voltage Drops as a Feedback Metric in Coverage-Guided Hardware Fuzzing on FPGAs”. Supervised by Jonas Krautter, Mark Giraud, and Anne Borcherding. Master’s Thesis. University of Applied Sciences Karlsruhe, 2023.

List of Figures

List of Tables

Listings

Acronyms

AE	Autoencoder
CAE	Convolutional Autoencoder
CAPEC	Common Attack Pattern Enumeration and Classification
CERT	Computer Emergency Response Team
CNN	Convolutional Neural Network
CPS	Cyber-Physical System
CRC	Cyclic Redundancy Check
CV	Coefficient of Variance
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DFA	Deterministic Finite Automaton
DoS	Denial of Service
DT	Decision Tree
EA	Evolutionary Algorithm
FPGA	Field Programmable Gate Array

FTP	File Transfer Protocol
GAN	Generative Adversarial Network
HMM	Hidden Markov Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IIoT	Industrial Internet of Things
IoT	Internet of Things
IP	Internet Protocol
ISTQB	International Software Testing Qualification Board
IT	Information Technology
LLM	Large Language Model
LSTM	Long Short-Term Memory Network
MaB	Multi-armed Bandit
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
NN	Neural Network
OPC UA	OPC Unified Architecture
OT	Operational Technology
OWASP	Open Worldwide Application Security Project

PC	Principal Component
PCA	Principal Component Analysis
PLC	Programmable Logic Controller
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SAE	Stacked Autoencoder
SCADA	Supervisory Control and Data Acquisition
SIL	Safety Integrity Level
SNMP	Simple Network Management Protocol
SuT	System under Test
SVM	Support Vector Machine
SWAT	Stateful Web Application Testing
TCP	Transmission Control Protocol
TD	Test Device
TLSH	Trend Micro Locality Sensitive Hash
TR	Tool Requirement
TT	Test Tool
URL	Uniform Resource Locator
VAP	Vulnerability Anti-Pattern
VDE	German Verband der Elektrotechnik, Elektronik und Informationstechnik

WA	Web Application
WR	Web Application Requirement
WVS	Web Vulnerability Scanner
XSS	Cross-site Scripting

Glossary

DHCP	Dynamic Host Configuration Protocol
DNP3	Distributed Network Protocol 3
DNS	Domain Name System
DOM	Document Object Model
HTTPS	Hypertext Transfer Protocol Secure, an application layer communication protocol used, for example, to communicate with web applications.
LLDP	Link Layer Discovery Protocol
LLMR	Link-Local Multicast Name Resolution
Modbus	An industrial request-response communication protocol.
PROFINET	An industrial communication protocol building upon industrial Ethernet.
SQL	Structured Query Language
UDP	User Datagram Protocol