L-SCALE: Locality-Sensitive Coverage for Automata LEarning

Mark Leon Giraud
Fraunhofer IOSB
Karlsruhe, Germany
mark.giraud@iosb.fraunhofer.de

Bastian Engel
Fraunhofer IOSB
Karlsruhe, Germany
bastian.engel@iosb.fraunhofer.de

Lea Nasarek
Fraunhofer IOSB
Karlsruhe, Germany
lea.nasarek@iosb.fraunhofer.de

Yannis Storrer
Fraunhofer IOSB
Karlsruhe, Germany
yannis.storrer@iosb.fraunhofer.de

Philipp Takacs
Fraunhofer IOSB
Karlsruhe, Germany
philipp.takacs@iosb.fraunhofer.de

Leon Philipp Wittemund
Fraunhofer IOSB
Karlsruhe, Germany
leon.philipp.wittemund@iosb.fraunhofer.de

Abstract

Approaches like AFLNet have successfully managed to improve upon stateless fuzzers by incorporating state information in the form of response codes or enum values in the source code. Most approaches use the state information directly instead of attempting to infer a state machine. However, having an explicit state model of the System under Test (SUT) is beneficial for designing more sophisticated fuzzing techniques, demonstrated by approaches that infer a state machine by using the responses the SUT sends.

In this work, we propose L-SCALE, an approach based on the L^*-Algorithm utilizing the coverage of individual messages as a measure of the behavior of the SUT. In contrast to other approaches that utilize only the responses of the SUT, this allows us insights into the internal state of the SUT, which is often not reflected in the responses it sends. Since coverage is a very fine-grained measure, we utilize TLSH, a Locality Sensitive Hash (LSH), in combination with a threshold value for the similarity of the hashes to categorize slightly different coverage measurements as the same observation.

We also contribute `mogi`, an emulator framework that provides deterministic execution and incremental snapshotting capabilities. `mogi` facilitates efficient exploration of deep states since we can reset to any point along the exploration path. This avoids having to replay all the messages along that path each time. We use `mogi` as a runtime environment for L-SCALE.

We evaluate L-SCALE on two real-world open-source targets: Redis, a widely used key-value store, representing the IT-domain, and open62541, an implementation of the OPC UA architecture, representing the Operational Technology (OT)-domain. Our results show that by using coverage as an observation measure, we can successfully identify detailed state representations of SUT, and that by utilizing LSH, one can control the coarseness of the learned state representation while still maintaining an accurate, albeit simplified, representation of the underlying state space. This enables testers to choose between fine-grained but complex, and coarse but simple state representations, depending on the use case. Our experiments also show that choosing a good threshold highly depends on the SUT. Overall, L-SCALE provides a means to automatically infer state machines from SUT binaries requiring minimal manual effort. As such, it poses great potential for future work on stateful fuzzing approaches that can utilize the learned states to improve fuzzing campaigns.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Networks** → *Protocol testing and verification*; • **Theory of computation** → *Active learning*; • **Security and privacy** → *Software and application security*.

Keywords

State Learning, State Inference, Active Automata Learning, L^*-Algorithm, Protocol Testing, Emulation, Locality Sensitive Hashing

ACM Reference Format:
Mark Leon Giraud, Bastian Engel, Lea Nasarek, Yannis Storrer, Philipp Takacs, and Leon Philipp Wittemund. 2026. L-SCALE: Locality-Sensitive Coverage for Automata LEarning. In *7th ACM/IEEE International Conference on Automation of Software Test (AST '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3793654.3793755

1 Introduction

Security testing and vulnerability discovery are critical in an era of increasing cyber threats. Greybox fuzzers like AFL++[14] and libAFL[15] are cornerstones of automated vulnerability discovery, but traditional approaches often struggle with complex, stateful systems that maintain internal state across interactions[31]. Meng et al.[31] emphasize that stateful fuzzing is a growing necessity.

Stateful targets, prevalent in Internet of Things (IoT) and Industrial Internet of Things (IIoT) systems, present significant attack surfaces [41, 43]. We evaluate two such targets: Redis [40] (IT sector) and open62541 [35] (OT sector). Redis [40], an in-memory key-value database and message broker, and open62541 [35], an implementation of the OPC UA protocol, which is being used increasingly in industrial control systems. The statefulness of these System under Tests (SUTs) means responses depend on both current input and previous sequences, creating opportunities for deep execution path bugs that are challenging for conventional testing.

Stateful fuzzers like AFLNet [36], StateAFL [33], SNPSfuzzer [26], RESTler [5], and IJON [3] demonstrate various approaches to overcoming the limitations of traditional fuzzing for stateful systems. These tools utilize various forms of state information, ranging from protocol specifications to runtime state variables, to guide their

AST '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2476-3/26/04
https://doi.org/10.1145/3793654.3793755

exploration of the target's state space more effectively. Some approaches use active automata learning [23] like TTT [24] to infer the state machine of the SUT. These approaches utilize the responses from the SUTs as feedback to the automata learning algorithm [11, 12, 16]. However, relying solely on the SUT's responses may miss internal state changes [6].

Building upon this foundation, we propose a novel state learning approach that leverages the target's coverage information as a behavioral indicator for the L^*-Algorithm [2]. Our method represents a departure from existing techniques by utilizing fine-grained execution traces rather than protocol responses or explicit state variables. This coverage-based approach provides detailed insights into the target's internal behavior, potentially revealing state distinctions that might be missed by approaches relying solely on observable outputs or predefined state indicators. The L^*-Algorithm requires a deterministic execution environment to function correctly.

Therefore, in addition to L-SCALE, we also contribute our emulator framework `mogi` that provides fully deterministic execution, functionality to create snapshots of the execution state, and enables detailed instrumentation of the target. We perform our instrumentation during emulation of the target. In contrast to the previously mentioned greybox approaches, L-SCALE only requires the target executable and no source code, making our approach applicable to more targets.

Since our approach introduces a very fine-grained observation measure, this can lead to an explosion of states, making it difficult to identify interesting states. To alleviate this issue, we also introduce a technique based on Locality Sensitive Hashing (LSH) to partition the fine-grained observation space into coarser compartments.

This work focuses exclusively on the state learning component of stateful fuzzing, intentionally decoupling it from the integration of learned states into fuzzing campaigns. This separation allows for a thorough investigation of the state learning methodology and provides a foundation for future integration efforts.

1.1 Contributions

The main contributions of this paper are:

- A novel approach to state learning of SUTs using coverage information as a metric for behavioral equivalence;
- An extensible emulator framework that enables fully deterministic and snapshottable execution of SUTs;
- An efficient observation space reduction technique using LSH to identify similar behaviors and a tunable threshold parameter to adjust the sensitivity;
- An empirical evaluation on two real world targets using only the compiled binary, demonstrating the effectiveness of our approach in discovering detailed state representations while managing state space complexity;

2 Background and Related Work

Fuzzing[28] in general has many challenges[7]. Specifically, the problem of stateful fuzzing can be divided into four aspects, each of which requires individual approaches. Ultimately, all of these aspects need to be combined to be able to fuzz stateful SUTs effectively. We identify the following aspects:

State Identification. In order to use state information, we first have to identify it. This can come in the form of enumeration values [6], manual annotations [3], or actual state inference by using available information [13, 16, 20, 36]. Fiterau-Brostean et al. [16] analyze DTLS implementations by using the TTT algorithm [24], and Marksteiner et al. [30] use various algorithms to learn NFC and BLE protocol behavior. They use the output of the target, i.e., the messages it responds with, as their observations. This feedback may not represent the full picture of what is happening to the SUT's state [32, 36]. The response codes of protocols can also be manually extended to include more detailed information on internal state by modifying the target executable. This is underlined by Giraud [22] who suggest that knowledge of internal state is advantageous. In contrast, we propose an approach that works without the need for state variables or modifications to the target. We also omit the necessity of a response parser as used in AFLNet[36]. Similar to our approach, Garhewal et al. [19] use taint analysis, but to reduce the number of queries needed, instead of as an observation.

Sequence Mutation. To discover more and deeper states, the fuzzer needs to send mutated input sequences to the target [36, 37]. This usually consists of inserting messages, swapping their order, removing messages, or even mutating some of the messages. Crafting effective sequence mutations requires knowing the state space, since, for example, inserting messages after a disconnect message usually makes little sense.

State Selection. When performing the actual fuzzing, the fuzzer needs to decide in which state of the SUT it wants to perform mutations. This requires having identified states from which we can select. One has to devise a weighting mechanism akin to the seed weights used in AFL++[14]. Borcherding et al., Liu et al. [8, 29] discuss various state selection strategies and their impact.

State-Aware Mutation. Lastly, the fuzzer must mutate and apply messages to the SUT in a certain state. Mutations can be guided by state information, helping select inputs likely to trigger interesting behavior in the state they are applied. It is, for example, more effective to fuzz connection-establishing messages in the disconnected state, since messages requiring an active connection are likely to be discarded. There are also multiple approaches that attempt to associate input information with state information in the SUT, such as REDQUEEN [4] and STEELIX [27].

Many stateful fuzzing approaches attempt to alleviate all of these problems at once. In many cases, it is not entirely clear which part of the proposed solution contributes how much to the overall efficacy of the approach, if at all. Thus, we focus solely on the *State Identification* aspect in our work and iterate on our findings in future work.

2.1 L^*-Algorithm

We base our approach on the active automata learning algorithm L^* devised by Angluin [2]. L^* uses a so-called *minimally adequate teacher* to learn a Deterministic Finite Automaton (DFA) by querying the teacher. The teacher provides an *observation function* $\text{obs} : \Sigma^* \to \mathbf{O}$ that maps a sequence of inputs into the SUT to the observation under this given sequence, where Σ is the input

alphabet of the SUT and $\mathbf{O}$ is the observation space of the SUT. In the case of the classic L^*-Algorithm, this corresponds to $\mathbf{O} = \{0, 1\}$, since it was originally designed to learn regular languages.

The algorithm essentially works by constructing a state transition table T that differentiates states by their observable behavior under different inputs. The rows of the table consist of prefixes $\mathbf{p} \in \mathbf{P} \subseteq \Sigma^*$ that lead to a specific state, and the columns consist of suffixes $\mathbf{s} \in \mathbf{S} \subseteq \Sigma^*$ applied in that state that serve as differentiators for the states. For the algorithm to work, the set $\mathbf{P}$ needs to be prefix-closed, meaning that for each $\mathbf{p} \in \mathbf{P}$, all of its prefixes are also part of $\mathbf{P}$. The set $\mathbf{S}$ needs to be suffix-closed, meaning that for each $\mathbf{s} \in \mathbf{S}$, all of its suffixes are also part of $\mathbf{S}$. The observation table is defined as follows:

Definition 2.1 (Observation Table). The observation table T can be split into an upper table $T_\uparrow$ and a lower table $T_\downarrow$. The rows in the table are labeled with the prefixes and their extensions $\hat{\mathbf{P}} = \mathbf{P} \cup \mathbf{P} \cdot \Sigma$, and the columns are labeled with the suffixes $\mathbf{S}$.

$$T : \hat{\mathbf{P}} \times \mathbf{S} \to \mathbf{O}$$
$$T_\uparrow : \mathbf{P} \times \mathbf{S} \to \mathbf{O}$$
$$T_\downarrow : \mathbf{P} \cdot \Sigma \times \mathbf{S} \to \mathbf{O}$$

The contents of the cells of T are the observations obtained when providing the teacher with the prefix and suffix identifying the row and column: $T(\mathbf{p}, \mathbf{s}) = \text{obs}(\mathbf{p} \cdot \mathbf{s})$.

The function $\text{row} : \hat{\mathbf{P}} \to \mathbf{S} \to \mathbf{O}$ is a partial application of T on only the prefix. For a given prefix $\mathbf{p} \in \hat{\mathbf{P}}$, the function $\text{row}(\mathbf{p})$ is defined as:

$$(\text{row}(\mathbf{p}))(\mathbf{s}) = T_\uparrow(\mathbf{p}, \mathbf{s}) \quad \forall \mathbf{s} \in \mathbf{S}$$

In essence, $\text{row}(\mathbf{p})$ encapsulates all the observations for the prefix $\mathbf{p}$ across all suffixes in $\mathbf{S}$. Two prefixes $\mathbf{p}_1$ and $\mathbf{p}_2$ are considered to exhibit identical behavior if their rows are equal, i.e., $\text{row}(\mathbf{p}_1) = \text{row}(\mathbf{p}_2)$.

The observation table then induces a graph $G = (V, E)$ as follows:

$$V = \left\{ \left\{ \text{row}(\mathbf{p}') \;\middle|\; \mathbf{p}' \in \mathbf{P} : \text{row}(\mathbf{p}') = \text{row}(\mathbf{p}) \right\} \;\middle|\; \mathbf{p} \in \mathbf{P} \right\}$$
$$E = \left\{ (v_1, v_2) \;\middle|\; \begin{array}{l} \text{row}(\mathbf{p}) \in v_1 \text{ and } \text{row}(\mathbf{p} \cdot \sigma) \in v_2 \\ v_1, v_2 \in V, \mathbf{p} \in \mathbf{P}, \sigma \in \Sigma \end{array} \right\}$$

All equal rows in the upper table are considered to be the same state, and the states are the sets of equivalent rows. Appending a symbol to a prefix in the upper table gives a row that corresponds to possibly the same or a different state, which constitutes a transition labeled by the symbol.

The teacher also provides an equivalence oracle in the form of a function $\text{equiv} : G \to \{\sigma \in \Sigma^*, \bot\}$. This function essentially either outputs a counterexample if the provided hypothesis automaton did not match the ground truth automaton, or it outputs $\bot$, signaling that the hypothesis is correct.

Whenever new prefixes or suffixes are added by the learner due to counterexamples, or when the observations in the table are filled, the algorithm needs to ensure the following two properties hold:

Definition 2.2 (Closed Property). The table T is considered *closed* if and only if for all rows in the lower table there exists a row in the upper table that is equal:

$$\forall \mathbf{p} \in \mathbf{P}, \forall \sigma \in \Sigma : \exists \mathbf{p}' \in \mathbf{P} : \text{row}(\mathbf{p} \cdot \sigma) = \text{row}(\mathbf{p}')$$

If a violation of this property is found, it can be fixed by adding the prefix of the row in the lower table to the upper table.

Definition 2.3 (Consistent Property). The table T is considered *consistent* if and only if for two rows, their successors given the same word σ exhibit the same behavior:

$$\forall \mathbf{p}_0, \mathbf{p}_1 \in \mathbf{P} \times \mathbf{P} : \forall \sigma \in \Sigma :$$
$$\text{row}(\mathbf{p}_0) = \text{row}(\mathbf{p}_1) \implies \text{row}(\mathbf{p}_0 \cdot \sigma) = \text{row}(\mathbf{p}_1 \cdot \sigma)$$

If a violation of this property is found, it can be fixed by extending the suffix where the difference occurred by σ and adding it as a new suffix to the table, thus effectively differentiating $\text{row}(\mathbf{p}_0)$ from $\text{row}(\mathbf{p}_1)$ and fixing the property.

The original L^*-Algorithm requires all observations to be filled out. However, it is also possible to have missing observations and still uphold all required properties. To this end, we extend the table mapping to $T : \hat{\mathbf{P}} \times \mathbf{S} \to \mathbf{O} \cup \{\bot\}$. When using missing observations, we define two observations to be undetermined if one or both are missing. If both observations are determined and not equal, they can be considered different, otherwise they are considered equal. Using this approach, we can check the closed property by verifying if there exists a row in the lower table that is not equal to any row in the upper table. If this is the case, we can add this distinct row to the upper table. Consistency checks can be performed in a similar manner by checking if there exist two equal rows for which their successor rows reached by σ are not equal. The found violation can then be fixed as previously mentioned. When using missing observations, the induced graph will contain multiple transitions to different states with the same word σ, since they cannot yet be differentiated. This constitutes a non-deterministic finite automaton. These non-deterministic transitions can be eliminated by further exploring the observations and filling out the table.

Since our approach in this work is implemented in Rust and for performance reasons, we decided not to use the established automata learning library LearnLib [39], but instead re-implemented the algorithm in Rust directly. This also allowed us to integrate the aforementioned missing observations approach.

2.2 Locality Sensitive Hashing

Cryptographic hash functions are designed such that small input variations produce uncorrelated outputs. In contrast, LSH preserves input similarity by assigning closely related inputs to similar outputs, enabling efficient nearest-neighbor search and dimensionality reduction of inputs. We use TLSH [34], which is the state-of-the-art for LSH, and provides a distance metric that can be applied to the hashes. We use the properties of TLSH to reduce large coverage maps to smaller hashes that retain the similarity of their inputs.

3 Approach

There are approaches that successfully leverage the outputs or state variables as observations, as discussed in section 2. To our knowledge, however, there is no approach that considers the use of coverage information for state inference. We thus propose learning a DFA that represents an approximation of the SUT's state space by using the L^*-Algorithm in combination with coverage information as observations. This DFA will only ever be an approximation, since

the input of most SUTs will conform to a context-sensitive grammar, which cannot be represented by a DFA. To approximate the SUT's state space, we use LSH and a comparison threshold to abstract away fine-grained details in the observation space of the SUT.

3.1 Input behavior

Recall that the learner needs a function obs : $\Sigma^* \to \mathbf{O}$ that maps a sequence of inputs to an observation. In our case, an input consists of any number of data streams or control commands that are applied in one step. This may, for example, be one or more Transmission Control Protocol (TCP) streams, standard input data, or other means of communicating with the target. In this work, we focus only on TCP and limit the number of concurrent connections to one.

To define our observations, we first define two coverage measurement techniques. The first one, called `dense`, uses a map that is as large as the code segment of the SUT. If a basic block in the SUT is executed, each element in the map that corresponds to an address that is part of the block is incremented by one, wrapping around if it overflows. The second coverage measurement technique, `sparse`, is very similar. It also uses the same map, but only increments the first address of the basic block. The reasoning is that the first address of a block is enough to know which following addresses have been executed.

With the two measurement techniques `sparse` and `dense`, we can now further define our observation space. For a given input $\mathbf{p} \cdot \sigma$, where $\mathbf{p} \in \Sigma^*$ and $\sigma \in \Sigma$, we measure the coverage that σ produces after having applied the prefix $\mathbf{p}$ to the SUT. This coverage measurement effectively captures what code is executed when the input s is processed, given that the SUT was set to some state by the prefix. Since the resulting coverage map is quite large, we then apply LSH to the coverage map, thus greatly reducing the size of the observation. This already introduces a reduction of the observation space. We then further utilize the distance metric provided by Oliver et al. [34] to define a threshold under which we consider the hashed coverage measurements equal.

We assign a unique ID to each coverage measurement that lies outside the threshold of all other measurements. If a measurement lies within the neighborhood of an existing measurement, we use two different approaches to pick an existing ID to assign. The first approach, `closest`, looks for the closest measurement and assigns the ID of that measurement to the new one. The second approach, `cutoff`, looks for the first discovered measurement and assigns the ID of that measurement to the new one. This effectively results in the coverage measurements being sorted into different buckets, reducing the observation space.

We can thus define four different observation spaces: $\mathbf{O}_{\text{CutD}}$, $\mathbf{O}_{\text{CutS}}$, $\mathbf{O}_{\text{CloD}}$, and $\mathbf{O}_{\text{CloS}}$. $\mathbf{O}_{\text{CutD}}$ uses the `cutoff` bucketing technique in combination with the `dense` coverage, and $\mathbf{O}_{\text{CutS}}$ uses the `sparse` coverage measurement. $\mathbf{O}_{\text{CloD}}$ uses the `closest` bucketing technique in combination with the `dense` coverage, and $\mathbf{O}_{\text{CloS}}$ uses the `sparse` coverage measurement.

3.2 Execution Environment

To facilitate running our SUTs in a deterministic and resettable manner, we provide our emulator framework `mogi`. `mogi` consists of the components depicted in fig. 1. The entire state of the system is

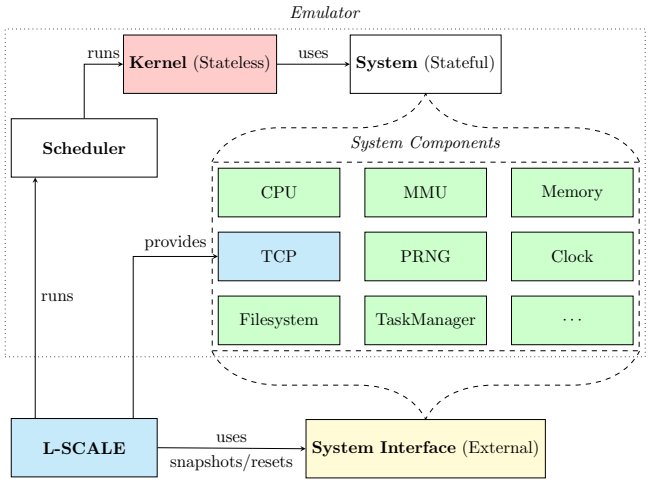


Figure 1: The structure of the `mogi` emulator framework and how L-SCALE interacts with it. Yellow indicates an interface, green indicates resettable system components. Red indicates stateless components, and blue is used for components associated with L-SCALE.

isolated in the system components (colored green) and implemented with copy-on-write data structures so that copying the state is cheap. The kernel implementation, i.e., the logic like syscall handling, is implemented in the kernel structure (colored red) and is entirely stateless. This allows us to rapidly reset the state in order to explore different paths through the SUT's state space.

The `CPU` and `MMU` components are implemented using the unicorn framework [38], for which we implemented and upstreamed an incremental memory snapshotting mechanism. The `Memory` component manages the memory mappings of the kernel to the emulated virtual address space while also keeping track of the emulated physical memory that the virtual addresses are mapped to. An in-memory filesystem that supports incremental snapshots is implemented by the `Filesystem` component. To facilitate deterministic execution, a pseudorandom number generator is implemented in the `PRNG` component and provides reproducible randomness for the emulated binary. The `Clock` component implements a clock that advances proportionally to the executed code block size.

The `System Interface` (colored yellow) is used by L-SCALE to interact with the emulator. L-SCALE uses this interface to fetch the memory area where the executable code is mapped. Additionally, L-SCALE registers basic block hooks to collect the coverage information it uses as observations.

L-SCALE also provides the `TCP` system component, which manages the interaction between L-SCALE and the emulated SUT. Whenever there is no more input available in the `TCP` component, the emulator will yield to L-SCALE, which can then decide on the next course of action. This avoids the need for timeouts required by approaches like AFLNet [36] that communicate directly over the network. Our approach is thus more efficient and also avoids spurious executions of the main loop during the timeout, which introduces non-deterministic behavior since the target may continue running for different amounts of time during the timeout.

During development of `mogi`, another emulator framework—`ICICLE`—was published by Chesser et al. [10]. We evaluated its performance and found that it was orders of magnitudes slower than `mogi`, both overall and when using just its CPU emulator. Hence, we decided to use our own framework for further development. However, `ICICLE`'s use of the `SLEIGH` [21] processor specification language from `GHIDRA` is favorable for making maintenance easier and facilitating fuzzing for different architectures, we have started work on investigating if and how their approach can be improved.

3.3 State Learning

To facilitate learning, we use our emulator `mogi` to execute the SUT. We use a depth-first approach to utilize the fact that our emulator supports incremental snapshots. This enables us to apply an input, reset to the previous state, and then proceed with the next input. Alternatively, we can stay in the state without resetting and search from that point onward. In the following we describe the core learning procedure of L-SCALE, depicted in Algorithm 1.

Learning continues as long as unobserved suffixes remain. Since our emulator currently only supports a linear history, we fill observations in a depth-first manner while tracking the current prefix path in $\mathbf{p}^{\text{cur}}$. In line 6, we search for any extension $\mathbf{p'}$ starting at $\mathbf{p}^{\text{cur}}$ that leads to rows with unobserved suffixes. To enable backtracking, we take snapshots of all intermediate prefixes in line 9. Snapshotting also allows us to efficiently apply suffixes with missing observations in line 12. Once all prefixes of a path are exhausted, we backtrack by restoring and discarding the latest snapshot and removing the last word from $\mathbf{p}^{\text{cur}}$ in line 20. We then restore consistency and closedness as described in line 2.1.

In line 23, we add additional prefixes if no empty observations remain and the `refinement` mode is enabled. The prefixes are constructed by randomly picking an existing prefix and then either inserting one or more messages chosen at random, swapping two random messages, or deleting one or more random messages. This essentially corresponds to a very basic equivalence oracle, since if we find sequences that produce inconsistencies in our table, we can refine the learned state machine. In future work, this may be combined with existing fuzzing approaches to explore the sequences in a more guided manner.

3.4 Implementation and Data Availability

We endorse open science and make all our code and data available in our GitHub organization *mogikai*[1] and invite interested individuals to cooperate on improving our work. Both the implementation of `mogi`, which currently consists of about 17 000 lines of Rust, as well as the implementation of L-SCALE, which currently consists of about 3 500 lines of Rust, are licensed under the MIT or Apache-2.0 license. We release the binary application for L-SCALE under the GPL-2.0 license. We include our raw data in a separate repository in the organization, but it is also reproducible by running the code we publish, since it is entirely deterministic.

Additionally, we include a preliminary implementation of a new CPU emulator inspired by ICICLE [10] with the goal of further improving execution times in the future. This CPU emulator is licensed under MIT or Apache-2.0 and consists of approximately 47 000 lines of Rust.

[1] https://github.com/mogikai

Algorithm 1: State learning procedure

```
1  Initialize T with test corpus as prefixes and suffixes
2  p^cur ← ()
3  Set up the emulator and run until first input is requested
4  emulator.take_snapshot()
5  while ⊥ ∈ T do
6      while ∃p' ∈ Σ* : (p^cur · p' ∈ P̂ ∧ ⊥ ∈ row(p^cur · p')) do
7          for σ ∈ p' do
8              emulator.run(σ)
9              emulator.take_snapshot()
10         end
11         p^cur ← p^cur · p'
12         while ∃s ∈ S : (T(p^cur, s) = ⊥) do
13             emulator.run(s)
14             Gather, reduce, and bucket observation o
15             T(p, s) ← o
16             emulator.restore_snapshot()
17         end
18     end
19     emulator.restore_and_discard_snapshot()
20     p^cur ← (p_1^cur, ..., p_{|p^cur|-1}^cur)
21     Make consistent and closed
22     if ∀(p, s) ∈ P × S : T(p, s) ≠ ⊥ and do_refinement
        then
23         Add random prefixes to T and make prefix closed
24     end
25 end
26 Export state machine
```

4 Evaluation

To evaluate L-SCALE, we pose the following research questions:

- RQ1 Does the LSH-based approach reduce the state space while preserving relevant information?
- RQ2 Are the learned state models an accurate representation of the SUT's state space?
- RQ3 How do different SUT's affect the behavior of L-SCALE?

Since there is no ground truth state machine for either Redis or open62541, we construct an *expert graph* to the best of our knowledge and compare it to the learned graphs. We present one learned graph that most closely resembles the expert graph for both Redis and open62541 and highlight the differences between the learned and expert graphs.

4.1 Hardware

We run our experiments on a server with an `AMD Ryzen Threadripper PRO 5975WX` CPU with 32 physical cores and 64 threads with 128GB of RAM. All of the emulated system binaries as well as the commands used to start our evaluations are contained within our repository (see section 3.4).

4.2 Targets

For our evaluation, we chose two real-world targets. One target – Redis – from the IT-domain, and one target – open62541 – from the OT-domain. We chose these targets for the following three reasons: First, they are open source, making it easy to acquire source code to reason about their internals and construct an expert graph. Second, they use TCP as their communication protocol, which is currently the only protocol stack `mogi` supports[2]. Third, they are epoll-based and as such do not rely on multithreading, which is currently unsupported by `mogi`[3].

The corpora for the targets were all constructed by either manually crafting the packets or by interacting with the target and logging exchanged messages with Wireshark and extracting them afterward. Just recently, fandango was published [42], which could be used to craft more sophisticated corpora in the future. Due to time constraints, however, we could not integrate this approach into our work.

For all of our targets, we provide the compiled binary and its system dependencies in our artifacts repository, since the resulting binary is highly dependent on the host system, the used compiler, and other external factors. Providing the compiled binaries and the system dependencies makes our tests fully reproducible.

4.2.1 open62541. open62541 is an open-source implementation of the OPC Unified Architecture (OPC UA) architecture, which is standardized as IEC 62541 [18]. We select this target because it has an open-source implementation and represents a standard that has seen increasing adoption in recent years [25]. It is deployed in diverse industrial applications, particularly in critical infrastructure systems [17].

OPC UA implements a complex protocol comprising numerous service sets. The most basic services are the `SecureChannel` and `Session` services. A `SecureChannel` is required for all the other services to function, whereas a `Session` is required for all services except for a few like the `GetEndpoints` service. To establish a working session, a client typically must send the following messages in sequence: `Hello`, `OpenSecureChannel`, `CreateSession`, `ActivateSession`. There are also messages like `CloseSession` and `CloseSecureChannel` that can be used to cleanly tear down the session and connection.

The protocol's inherent state complexity, with multiple concurrent sessions, subscription management, and browse operations, results in a vast state space that traditional stateless fuzzing approaches struggle to explore effectively. This complexity makes open62541 an ideal candidate for evaluating coverage-based state learning techniques.

For our evaluation, we use two different corpora as listed in table 1. The first corpus is small and consists of only the essential messages required for managing the `SecureChannel` and `Session`, and one service message that requires an active session. We select this corpus as it facilitates easier manual analysis of the resulting state machine. The second corpus is a superset of the first corpus and includes additional messages of a few services that either only require a `SecureChannel` or a fully established `Session`.

[2]We plan to add support for additional communication stacks in the future.
[3]We are working on multithreading support in an effort to make finding concurrency bugs easier.

Table 1: The messages used in the open62541 corpus. The `large` corpus additionally includes all messages of the `small` corpus. We include short names for visibility reasons in the state graph. Messages marked with * require an active `SecureChannel`. Messages marked with † require an active `Session`.

Corpus	Message	Short name
small	Hello	Hel
	OpenSecureChannel	Open
	CloseSecureChannel*	Clo
	CreateSession*	CSess
	ActivateSession*	ASess
	CloseSession†	CloSess
	Read†	Read
large	GetEndpoints*	GetEP
	Browse†	Bro
	CreateMonitoredItem†	CMoni
	CreateSubscription†	CSub
	DeleteSubscription†	DSub
	TranslateBrowsePath†	TBP
	FindServers†	FServ
	FindServersNetwork†	FServN

For our evaluation, we use the commit with the hash cc5c7da[4]. We compiled open62541 in release mode with the following options:

- `UA_BUILD_EXAMPLES=true`
- `UA_ENABLE_DETERMINISTIC_RNG=true`
- `FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION`

In addition to these options, we disable security token checks when `UA_ENABLE_FUZZING` is set to `true`. Leaving these checks in would simply make crafting correct input messages more difficult, but it would still be possible since our emulator is fully deterministic, and the tokens use incrementing IDs.

4.2.2 Redis. With Redis [40], we added a popular single-threaded key-value database server application to our set of targets. Being single-threaded is the main constraint for a target to be executable inside our emulator. For that reason, we compiled Redis in its default configuration.

However, to emulate Redis, we had to deactivate features like background IO and tasks that are handled by creating new tasks manually in the Redis code. To ensure performance, we build Redis in release mode. We work with Redis unstable[5].

Redis provides numerous messages to store, update, or retrieve data. For our evaluation, we use two different corpora as listed in table 2. In our `small` corpus we focus on a small set of basic operations: `Get`, `Inc`, and `Set`, which work on keys in the data store. With `Inc`, a counter value is incremented or created that can be accessed by `Get`. For a real-world setting, we activated the `requirepass` option in the configuration. Invalid clients are forbidden from executing messages. For that reason, our `small` corpus also contains

[4]https://github.com/open62541/open62541/commit/cc5c7da6a11bf0fe012dbcd5af48a4f1d9fc6297
[5]https://github.com/redis/redis/commit/f6d1fd08f9b03a90c4e6a18866f8709b86bae627

Table 2: The messages used in the Redis corpus. The `large` corpus additionally includes all messages of the `small` corpus. We include short names for visibility reasons in the state graph. Messages marked with * can be executed without authentication. `AuthenticateTest` can only execute messages marked with †.

Corpus	Message	Short name
small	AuthenticateDefault*	Auth
	Increment	Inc
	Get†	Get
	Quit*†	Quit
large	AuthenticateTest*†	ATest
	CreateUserTest	CTest
	DeleteUserTest	DTest
	Echo	Echo
	Reset*†	Rst
	Set	Set
	ListUsers	LUsers
	ListRules	LRules
	WhoAmI	WhoAmI

`AuthenticateDefault`. After authentication, the default user has admin rights to execute any message.

Our second corpus—`large`—which is a superset of the small corpus, covers more complex access control behavior. We thus add specific access control messages reserved for the admin to the `large` corpus: `LUsers`, `LRules`, `WhoAmI`. For differentiation, we included `CreateTest` and `DelTest` to have a user that can only execute `Get`. To close a session with the server, we add `Quit`. `Reset` undoes the current authentication.

4.3 Experiments

Our approach provides multiple adjustable parameters. L-SCALE implements the four different observation (see section 2.1) methods $\mathbf{O}_{\text{CutD}}$, $\mathbf{O}_{\text{CutS}}$, $\mathbf{O}_{\text{CloD}}$, and $\mathbf{O}_{\text{CloS}}$. It is also possible to run the learning process with or without `refinement` as described in section 3.3. Additionally, the threshold may be configured to an arbitrary positive integer.

For our evaluation we choose to run L-SCALE on both of our targets with all four observation methods and with thresholds in the interval $[0, 40]$. We also run L-SCALE both with and without `refinement` enabled. We perform our evaluation for both the `small` and `large` corpus of the targets, resulting in a total number of runs for each target of $4 \cdot 41 \cdot 2 \cdot 2 = 656$. We run as many instances of the learner in parallel as we have cores on our evaluation machine (see section 4.1), pinning each instance to one core. Since for small thresholds the resulting state space is very large, we limit the runtime to 4 hours after which the experiment is stopped.

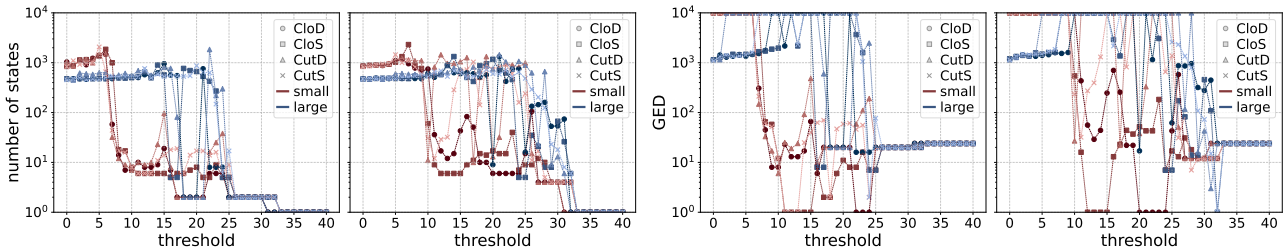
4.4 Results

In the following we present the results of our experiments per target. We plot the number of states discovered by our state learning tool across the different targets, corpus combinations, and threshold values. We plot the `small` and `large` corpus in the same graph to compare them. We also plot the Graph Edit Distance (GED) which we calculate as described by Chang et al. [9] between the expert graph and the learned graph for each target. As computing the GED is NP-hard we limited the difference between the expert and learned state graph to be 500 nodes, effectively limiting the computation time. We also use a timeout of one hour after which the closest approximation of the GED that has been found until then is used. We also present the learned graph that most closely corresponds to our expert graph for open62541 and Redis respectively.

4.4.1 open62541. In fig. 2a, we can see the number of states plotted against the threshold values for our `small` and `large` corpora and different bucketing strategies. In general, increasing the threshold leads to a decrease in the number of states since more observations are being merged. Thresholds past 33 cause all experiments to collapse into a single state. There is a sharp drop in the number of states at a threshold of around 7 for the `small` and around 22 for the `large` corpus. If we enable refinement, this drop is shifted slightly to the right since the random exploration strategy manages to find more states. This is also why equal parameters yield strictly more states with refinement enabled. Since refinement is inherently probabilistic and the exploration path depends on the order of insertion due to the bucketing, increasing the threshold does not necessarily decrease the number of states. Using the `large` corpus for small thresholds causes the fuzzer to run into a timeout due to the larger number of messages to explore. This prohibits L-SCALE from finding all possible states which is the reason for the `large` corpus having fewer states than the `small` corpus, which is expected, since L-SCALE has to perform more work for each individual state it discovers due to the increased number of messages to test.

Figure 2b depicts the GED of each learned graph to the expert graph for each threshold value for our `small` and `large` corpora. For both corpora, increasing the threshold beyond 25 without refinement, and 35 with refinement, results in the GED converging to the same value. This is because beyond this threshold the infered graph consists of only a single state. For the `large` corpus, different strategies yield edit distances exceeding 1000 below a threshold of 20, either due to a genuinely large ged or because the script terminated the computation early. With refinement enabled, this pattern and the convergence for both corpora shifts further to the right, as the random exploration strategy remains effective in discovering new states despite higher thresholds. For the `small` corpus, thresholds below 7 result in very large edit distances. The low threshold causes fine-grained differences in coverage measurements to be treated as distinct observations. We can observe similar behavior for the other corpus, and for the runs with refinement enabled. In general, the minimum GED is reached at localized minima (e.g. 22 to 24 for small and $\mathbf{O}_{\text{CloD}}$). We can see that the minimum GED is highly dependent on the threshold, the bucketing method as well as the coverage measurement method.

We also manually analyzed a few of the state machines L-SCALE produced for open62541. One such state machine is depicted in fig. 3. It is the result of L-SCALE running with a threshold of 24 using the $\mathbf{O}_{\text{CloD}}$ mode without refinement on the `small` corpus. We can see that L-SCALE correctly inferred the connection establishing



(a) Learned states without (left) and with refinement (right)

(b) GED without (left) and with refinement (right)

Figure 2: Total states achieved by applying L-SCALE to open62541 on a set of different threshold values (range 0 to 40) with all observation measurements. The y-axis is scaled logarithmically. The red plots correspond to the four different observation techniques applied to the `small` corpus. The blue plots correspond to the four different observation techniques applied to the `large` corpus. We add 1 to the GED such that it can be easily displayed in a logarithmic scale.

procedure: Any message sent in Node 0, Node 1, or Node 2 will simply result in the connection being closed which corresponds to a transition to Node 0. The self loops containing `TConn` are a result of our approach not allowing more than one TCP connection, and treating a new connection request as a no-op. Node 3 represents the state where a `SecureChannel` has been successfully established. Sending `Hel` or `Opn` in an established `SecureChannel` results in the connection being closed, which was correctly inferred as well. Performing anything other than `CSess` in order to create a `Session` in Node 3 does not change the state and simply returns an error to the user via the connection, leaving the state untouched. Node 4 corresponds to the intermediate state of having created a session, but not yet activated it. Here, our approach differs from our expert graph: We model our state machine with `Read` staying in Node 4, whereas the inferred state machine transitions to Node 3. This

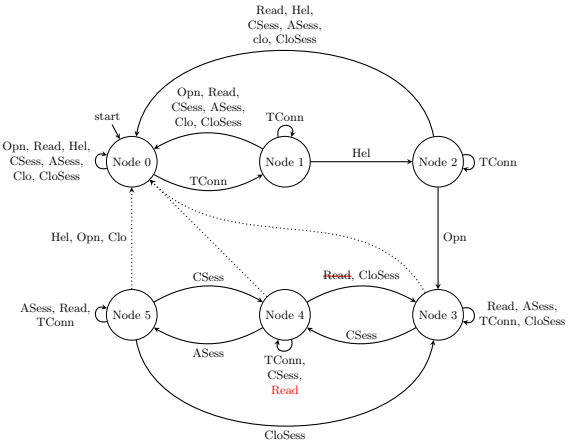


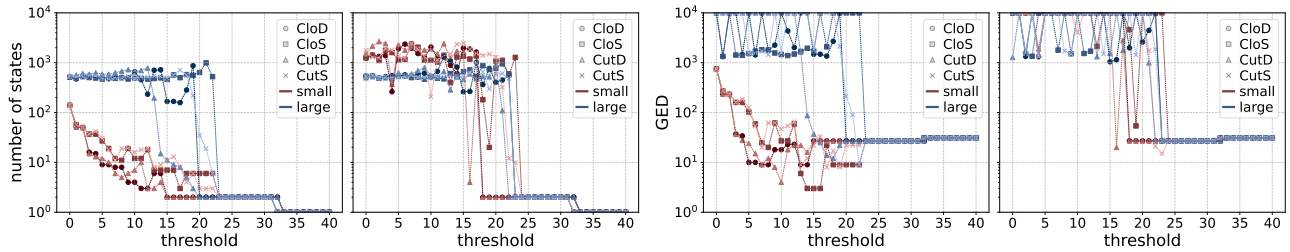
Figure 3: State graph generated by applying L-SCALE to open62541 with a threshold of 24 using O_{CloD} without refinement. The graph shows the learned states, connected by labeled edges representing different operations from the `small` corpus. For simplicity, the dotted arrows' labels are omitted and only written on the leftmost arrow. The red corrections are required to achieve the expert graph.

discrepancy is most likely the result of the behavior of `Read` in Node 4 being slightly different due to an internal state change and thus being categorized into a different bucket than the one in Node 3. Finally, Node 5 corresponds to an active session, which is correctly identified by L-SCALE. In this state, activate session simply returns an error and read can successfully perform the operation. L-SCALE also correctly identifies that `CloSess` closes an active session. We also find that the graph for the `large` corpus with a threshold of 32 using the O_{CutS} or O_{CloS} modes with refinement yields an equivalent graph, simply with more messages on the transitions.

Overall, this shows that L-SCALE is able to identify a very reasonable state machine given a fitting threshold and observation method. Even at low thresholds, the resulting state machine still shows the connection-establishing states. However, once a session is activated, the state space becomes very large, as L-SCALE detects even very small internal state changes. Using other thresholds reveals more fine-grained states. Since the state machines are very large, we cannot include them in our paper, but all learned state machines are available in our artifacts and can be reproduced.

4.4.2 Redis. Figure 4a shows the number of states learned for specific threshold values with and without refinement. Similar to the states of open62541 depicted in fig. 2a, we can see a cutoff for the `large` corpus due to the larger number of paths to explore. In contrast to open62541, the `small` corpus does not exhibit a clear cutoff but rather a gradual reduction of the number of states. Similar to open62541, enabling refinement, shifts the dropoff to higher thresholds and makes it more pronounced. This is partly because more runs encounter a timeout, which flattens out the curve on the top. We would expect to see an increase in states with lower thresholds, given more execution time.

Figure 4b depicts the GED of each learned graph to the expert graph for each threshold value for our `small` and `large` corpora. For the same reason as for open62541, increasing the threshold beyond 22 without refinement, and 24 with refinement, results in the GED converging to the same value. As for open62541, small thresholds result in a larger GEDs, since the learned graph becomes very detailed. However, in contrast to open62541 no learned graph reaches a GED of zero, and open62541 has state graphs with distance 0 even with refinement enabled. Furthermore, no configuration



(a) Learned states without (left) and with refinement (right)

(b) GED without (left) and with refinement (right)

Figure 4: Total states achieved by applying L-SCALE to Redis on a set of different threshold values (range 0 to 40) with all observation measurements. The y-axis is scaled logarithmically. The red plots correspond to the four different observation techniques applied to the `small` corpus. The blue plots correspond to the four different observation techniques applied to the `large` corpus. We add 1 to the GED such that it can be easily displayed in a logarithmic scale.

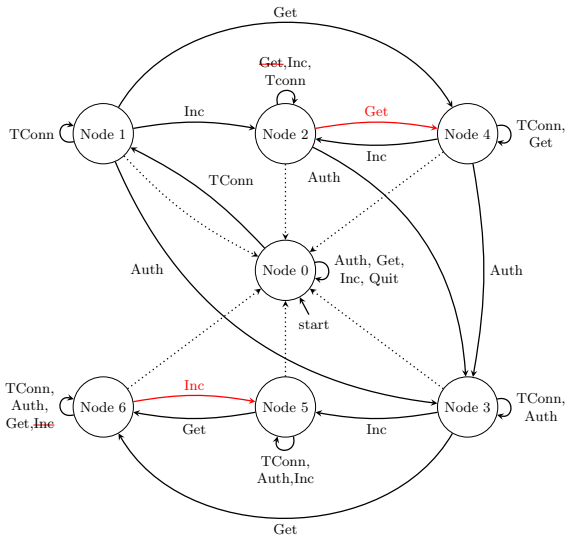


Figure 5: State graph generated by applying L-SCALE to Redis with a threshold of 16 using O_{ClOS} without refinement. The graph shows the learned states, connected by labeled edges representing different operations from the `small` corpus. The dotted arrows correspond to the `quit` command. The red corrections are required to achieve the expert graph.

reaches a GED below 10, as the learned state machines are rather large and fine-grained. Manual analysis however showed that these state-machines are still sensible, albeit very large due to many internal states being learned. Overall our observations are very similar to those for open62541, with the differences arising from the different code base.

As for open62541, we also manually analyzed a few of the state machines L-SCALE produced for Redis. In fig. 5 the result of L-SCALE running with a threshold of 16 using the O_{ClOS} mode without refinement is depicted.

After establishing a connection, Redis is in an unauthenticated state (Node 1). This is evident in Node 3, where all incoming edges refer to the `Auth` request. Before authentication, any message except `Auth` logically should not change state. However, Redis checks permissions only after parsing and potentially caching the entire request. This caching behavior leads to states in Nodes 2 and 4. For example, starting with `Inc` in Node 1 transitions to Node 2. Another `Inc` maintains the state as Redis caches the message. Executing a different message switches to the cached state of that message. Although L-SCALE correctly infers most of the states in the expert graph as seen in fig. 5, there remain some discrepancies: The red edges in the graph show how to transform the learned state machine into our expert graph. At first, we learned the path from Node 1 to Node 3 via Node 2. Due to the threshold of 16, we can not observe differences in executing `Get` and `Inc` in Node 2. Consequently, the transition with `Get` to Node 4 is not recognized. However, with decreasing threshold our statelearner is capable of recognizing the difference. Configuration O_{ClOS} with threshold 6 includes exactly the toggling between the two states.

In the authenticated state (Node 3), the caching behaviors of `Get` and `Inc` are similar. Since we are already in an authenticated state, `Auth` does not change the state. Due to the large code base of Redis the impact of the actual behavior of `Get` and `Inc` is not very noticeable. Lowering the threshold reveals more detailed and accurate representation of these transitions, but also introduces more noise in the form of overly detailed transitions. Other state representations L-SCALE infers manage to capture more of the internal state space at the cost of a more complex state machine. For example, the graph learned with threshold 8 and O_{ClOS}—which we could not include due to its size—reflects more of the internal behavior: After `Inc` has been executed for the first time, terminating the connection leads to a new subgraph that reflects the behavior after the counter variable has been created.

4.5 Overall Insights

Our evaluation reveals several key insights about the behavior of L-SCALE. The number of states tends to decrease with higher threshold values (see figs. 2a and 4a), as expected from the LSH-based merging mechanism, thus answering RQ1. The learned state machines appear reasonable and accurately reflect the target behavior as far as we can determine, answering RQ2. Refinement operations consistently increase the number of discovered states by revealing previously hidden state distinctions. For Redis we find 71

states more on average, and for open62541 103. Regarding RQ3 we observe that the choice of parameters is highly target-dependent, requiring careful tuning for each SUT and the intended use-case. Additionally, larger corpora require more exploration time per state but yield more comprehensive state representations. The results further demonstrate that lower threshold values preserve more fine-grained state distinctions, resulting in higher state counts, while higher threshold values aggressively merge similar states, leading to more compact state machines. The choice of threshold therefore represents a trade-off between state machine detail and complexity reduction.

5 Limitations and Future Work

Due to the nature of stateful SUTs, it is impossible to model them accurately with a DFA. With L-SCALE we have demonstrated that coverage information can be used as observations in active automata learning. However, there exist more suitable models like the Visibly Pushdown Automaton (VPA) [1] which can be learned by the TTT algorithm [24]. For our work we opted for the simpler L^*-Algorithm to establish a proof of concept. In future work, we plan to integrate TTT into L-SCALE to learn a more appropriate state model and also improve the learner's performance.

Currently, the observations are not distributed equidistantly due to the way our bucketing methods work. This also causes the exploration path to be dependent on the order of how we insert observations which in principle is undesirable. It may be beneficial to pre-allocate the buckets in an equidistant manner and this should be evaluated. Furthermore, we propose adjusting the threshold dynamically. This could take into account the number of states learned, adjusting the threshold to a larger value if there are too many states, and to a smaller value if there are too few states. Doing so would require tracking more information and rebuilding the observation table at runtime.

Liu et al. [29] identify slow execution speed (20 exec/s) as one factor why the state selection algorithms do not differ much. Although still slow, L-SCALE manages to execute more quickly and in a fully deterministic manner (between 50-100 exec/s). They also identify the complexity of the structured inputs to be problematic for state discovery. This could be alleviated by combining L-SCALE with tools like fandango [42] and other input-to-state mapping approaches [4, 27].

One of the big limitations on throughput for our approach is the speed of the emulation. While approaches like TTT or other automata learning methods require less queries, there is still potential to improve the speed of our emulator, especially the performance of restoring memory snapshots, and we have already begun work on improvements. Further performance improvements could also be gained by making L-SCALE more distributed, having multiple emulator instances, and delegating the work of gathering observations to these instances.

In this work, we mainly focused our efforts on demonstrating that coverage combined with LSH can be used as a successful observation metric for active automata learning. In future work, we might explore different sequence mutation strategies to further enhance the quality of the learned state machine, while at the same time keeping the number of less interesting states small.

Also, our approach should be integrated with existing techniques for fuzzing to build a more efficient fuzzer for stateful SUT. This includes using state-specific seed weights, and input-to-state correspondence. It is also possible to integrate the state-discovery alongside the fuzzing queries, i.e., only collecting observations for the inputs applied during fuzzing, and enhancing the state machine during the fuzzing process, instead of performing a separate state-learning phase. This integrated approach should then further be compared with existing fuzzers like AFLNet[36].

6 Conclusion

In this work we propose L-SCALE, a novel approach for inferring state machines of stateful SUT that utilizes the coverage information of the SUT as observations. This contrasts with previous approaches that only use the responses of the SUT as observations.

We use LSH with an adjustable threshold to merge similar observations, thereby reducing the noise introduced by our detailed coverage feedback. We use two different methods for merging similar observations. One method picks the closest existing observation that lies within the provided threshold, whereas the other method picks the first observation that lies within the provided threshold. We combine both of these bucketing methods with two different coverage measurement techniques, one of which measures each executed address in a block, whereas the other only measures the first address executed in a block.

We further contribute `mogi`, an extensible emulator framework that enables deterministic execution and efficient snapshotting. Preliminary evaluation shows our framework to be faster than ICICLE[10]. We use `mogi` as runtime environment for our state-learning approach L-SCALE.

We evaluate L-SCALE on two real world targets representing the IT and OT domain: Redis and open62541. We find that the choice of the bucketing method depends on the SUT and the used corpus. In general though, the number of learned states L-SCALE manages to infer becomes less with increasing threshold regardless of the used bucketing method. We also find the choice of observation method and threshold to be highly dependent on the SUT. By comparing the state machines inferred by L-SCALE with carefully constructed expert graphs, we show that the inferred state machines are sensible and represent an accurate model of the SUT.

Overall, L-SCALE provides a means to effectively learn state machines that represent the internal state of SUT, while at the same time providing tunable hyper-parameters in the form of four observation methods and a threshold to control the granularity of the inferred state model. With L-SCALE, we provide a promising tool that can serve as a foundation for future stateful fuzzing research.

Acknowledgments

This work was partly funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs, Karlsruhe. We would also like to thank Anne Borcherding for her helpful insights and feedback.

References

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, Chicago IL USA, 202–211. doi:10.1145/1007352.1007390

[2] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

[6] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.

[7] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

[8] Anne Borcherding, Mark Giraud, Ian Fitzgerald, and Jürgen Beyerer. 2023. The Bandit's States: Modeling State Selection for Stateful Network Fuzzing as Multi-armed Bandit Problem. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, Delft, Netherlands, 345–350. doi:10.1109/eurospw59978.2023.00043

[9] Lijun Chang, Xing Feng, Xuemin Lin, Lu Qin, Wenjie Zhang, and Dian Ouyang. 2020. Speeding up GED verification for graph similarity search. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 793–804.

[10] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. 2023. Icicle: A redesigned emulator for grey-box firmware fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 76–88.

[11] Lesly-Ann Daniel, Erik Poll, and Joeri De Ruiter. 2018. Inferring OpenVPN state machines using protocol state fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 11–19.

[12] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. 193–206.

[13] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.

[15] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065.

[16] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2523–2540.

[17] OPC Foundation. 2025. Aker BP Chooses SCADA System Powered by OPC UA. https://opcconnect.opcfoundation.org/2018/10/aker-bp-chooses-scada-system-powered-by-opc-ua/. Accessed: 2025-07-06.

[18] OPC Foundation. 2025. OPC UA Reference. https://reference.opcfoundation.org/. Accessed: 2025-07-07.

[19] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. 2020. Grey-Box Learning of Register Automata. arXiv:2009.09975 [cs.FL] https://arxiv.org/abs/2009.09975

[20] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[21] Ghidra. 2025. Ghidra: SLEIGH specification. https://github.com/NationalSecurityAgency/ghidra/tree/master/GhidraDocs/languages/html. Accessed: 2025-07-16.

[22] Mark Giraud. 2020. *Design and evaluation of methods for efficient fuzzing of stateful software*. Master's thesis. Karlsruhe Institute of Technology (KIT).

[23] Falk Howar and Bernhard Steffen. 2018. Active automata learning in practice: an annotated bibliography of the years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*. Springer, 123–148.

[24] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer, 307–322.

[25] Marc Ladegourdie and Jonathan Kua. 2022. Performance analysis of opc ua for industrial interoperability towards industry 4.0. *IoT* 3, 4 (2022), 507–525.

[26] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2673–2687.

[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 627–637.

[28] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[29] Dongge Liu, Van-Thuan Pham, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. 2022. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 720–730.

[30] Stefan Marksteiner, David Schögler, Marjan Sirjani, and Mikael Sjödin. 2025. Learning single and compound-protocol automata and checking behavioral equivalences. *International Journal on Software Tools for Technology Transfer* (2025), 1–18.

[31] Ruijie Meng, Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2024. AFLNet Five Years Later: On Coverage-Guided Protocol Fuzzing. doi:10.48550/arXiv.2412.20324

[32] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. doi:10.14722/ndss.2018.23166

[33] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.

[34] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In *2013 fourth cybercrime and trustworthy computing workshop*. IEEE, 7–13.

[35] open62541. 2025. Open62541: Open Source OPC UA licensed under the MPL v2.0. https://www.open62541.org/. Accessed: 2025-07-01.

[36] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[37] Peng Qian, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He. 2024. Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1972–1985.

[38] NGUYEN Anh Quynh and DANG Hoang Vu. 2015. Unicorn: Next generation cpu emulator framework. *BlackHat USA* 476 (2015).

[39] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*. 62–71.

[40] Inc. Redis. 2025. Redis: The Real-time Data Platform. https://redis.io/. Accessed: 2025-07-01.

[41] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. 2015. Security and privacy challenges in industrial internet of things. In *Proceedings of the 52nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. Association for Computing Machinery, New York, NY, USA, Article 54, 6 pages. doi:10.1145/2744769.2747942

[42] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 894–916.

[43] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. 2014. IoT security: ongoing challenges and research opportunities. In *2014 IEEE 7th international conference on service-oriented computing and applications*. IEEE, 230–234.