# NOTHORG; A digital twin for ECU security testing

Marek Wehmer
FZI Research Center for Information Technology
Karlsruhe, Germany
wehmer@fzi.de

Jean Diestl
FZI Research Center for Information Technology
Karlsruhe, Germany
diestl@fzi.de

Dominik Waibel
FZI Research Center for Information Technology
Karlsruhe, Germany
waibel@fzi.de

Ingmar Baumgart
FZI Research Center for Information Technology
Karlsruhe, Germany
baumgart@fzi.de

## Abstract

Testing cybersecurity properties of vehicle components is an inherently difficult task, especially if testing Electronic Control Units (ECUs) in isolation and outside of their intended operation environment. Rehosting ECU firmware is a challenging task because of internal and external runtime dependencies. Virtualization of the full system often fails to reach the required operational state because of unfulfilled preconditions in the distributed system environment. We propose NOTHORG, a framework architecture for creating digital twins of ECUs that allows emulation and runtime-stubbing of the required environment to facilitate dynamic cybersecurity testing. NOTHORG supports replaying external communication via Controller Area Network (CAN) bus and Ethernet and allows the integration of physical devices to the virtual environment. We validate our concept with a reference implementation and show that typical security testing tasks can be performed efficiently by combining static analysis, image preparation, dynamic runtime hooks, environment stubbing and log consolidation to reach operation states that would otherwise be unreachable in introspectable cybersecurity testing environments.

## CCS Concepts

• **Computer systems organization → Firmware**; • **Security and privacy → Distributed systems security**; **Mobile platform security**; **Virtualization and security**; **Penetration testing**.

## Keywords

digital twin, automotive cybersecurity testing, ecu rehosting

**ACM Reference Format:**
Marek Wehmer, Jean Diestl, Dominik Waibel, and Ingmar Baumgart. 2025. NOTHORG; A digital twin for ECU security testing. In *Proceedings of the 2025 Cyber Security in CarS Workshop (CSCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3736130.3764512

## 1 Introduction

In the automotive industry, regulations such as UN ECE R155 incentivize Original Equipment Manufacturers (OEMs) and suppliers to validate and monitor the cybersecurity of their products. Validation of security properties of cyber-physical systems (CPS) is a non trivial task. Efficiently testing security properties of ECUs is a challenge for suppliers, OEMs and security researchers alike. The tightly coupled design of E/E-Architectures makes testing of individual components particularly difficult and non-portable when compared to related domains such as some cohorts of Internet-of-Things (IoT)-Devices or routers that follow a relatively homogeneous system architecture where some parts of the analyses can be effectively automated in virtual environments.

While the pursuit of automating security evaluation is notable and partly successful in other domains, automotive security assessments remain a largely manual task due to the high degree of customization, software implementation and environment integration of each individual component.

A common method for security testing is to conduct black-box penetration tests to uncover weaknesses during the runtime and operation of vehicle components. During the test, security analysts try to identify and exploit flaws in the implementation and behavior of the component, often by using existing vulnerability databases or by finding new vulnerabilities (e.g. by fuzzing available interfaces over the network). While penetration tests can be an effective measure to fix easily exploitable flaws, a more thorough evaluation of the security level is often not possible with this approach. For instance, we observe that ECUs use TLS with a symmetric pre-shared key (PSK) to protect the connection to other components inside the vehicle. While black-box approaches can see the TLS-protected channel, a more thorough security assessment benefits from answering more intricate questions:

(1) How is the PSK managed and stored inside the ECUs microprocessor and storage architecture?
(2) Are the currently unexposed communication endpoints inside the TLS channel vulnerable?

Even though the TLS-channel reduces the exposure of the inner vulnerabilities, these weaknesses might become exploitable for attackers that possess key material or are able to compromise the other endpoint of the connection. While static analysis can help in the analysis, the process is time consuming and error prone, because the runtime behavior can in many cases not be inferred. For this reason, most practical security evaluation techniques rely on inspecting the runtime behavior of the firmware. In the IoT-domain, rehosting firmware to provide a functional environment for analysis and automated test cases is a well established approach to separate the hard-to-debug and potentially inaccessible hardware from the firmware as Target of Evaluation (ToE) [1].

Rehosting firmware of embedded devices in general is challenging, often labor intensive and a young field of research in general. We observe that rehosting modern ECU firmware entails some additional challenges specific to the automotive domain, which are not considered by existing approaches, i.e.

(1) Use of in-vehicle communication busses (CAN, LIN),
(2) Multiple microcontroller targets on the same ECU,
(3) Strong interdependence between complex components in the E/E-Architecture,
(4) Hardware-backed cryptographic key distribution protocols,
(5) Use of secure elements and trusted execution environments,
(6) Coupling to vendor cloud services.

The most notable difference is that ECUs typically only work inside their strongly coupled environment consisting of other ECU and peripherals. Rehosting is therefore not sufficient to inspect runtime behavior, if other ECUs and external services are unavailable.

A fully functional virtual execution can only be achieved with simultaneous emulation of the necessary environment to handle challenges (1), (2), (3) and (6). At the same time, challenges (4) and (5) significantly complicate whole-system emulation by rehosting the firmware, because the required cryptographic material is stored in protected memory regions or dedicated Secure Elements (SEs). This has two fundamental consequences: First, the assumption that, in principle, any firmware can be rehosted perfectly, if the virtualization environment is modeled in sufficient detail does not hold when key material is not known. Second, because pre-shared cryptographic keys used to establish trusted communication channels (authentication, encryption) to on-board components, external E/E-services or vendor cloud services are not available in the virtualized firmware, hardware-in-the-loop approaches are limited and rehosting requires modification of the firmware. Some parts of the data and firmware may even be unavailable for rehosting, if they are encrypted with hardware-protected keys. This is often the case for partitions of trusted execution environment such as ARM TrustZone. [11]

For classifying security mechanisms, [11] distinguishes between three types of embedded systems based on their Operating System (OS):

**Type-I:** General purpose OS-based devices.
**Type-II:** Embedded OS-based devices.
**Type-III:** Devices without an OS-Abstraction.

For type-II and type-III systems, rehosting for security testing has been approached with different approaches and the concept has been proven [4]. Fasano et al. distinguish previous works by four different rehosting approaches: pure emulation, hardware-in-the-loop emulation, symbolic modeling of peripherals and hybrid systems combining hardware-in-the-loop with symbolic modeling [6]. While symbolic modeling has been implemented for type-II and type-III ECUs before [4], we observe that the complexity of type-I ECU firmware is typically not compatible with the constraints of symbolic modeling and the first two are necessary, but due to the above not sufficient to be useful. To facilitate efficient dynamic security testing for type-I ECUs, we propose a reusable pluggable architecture to rehost the firmware.

In the last years, digital twins have been developed to provide a virtual representation of systems and their environment. Digital twins aim at testing and predicting behavior for dynamic systems and specifically allow testing the interactions between multiple systems and their reaction based on changes.

## 2 Related Work

Using digital twins in the context of vulnerability scanning has been explored before. Specifically, for automotive components, and ECUs, several works are known. Most of those share their focus on regulatory compliance and the testing of security requirements instead of technical description of the digital twin itself.

[10],[9] introduce methods for automated security testing based on a formal representation of the ECU and automated test case generation. Their achievements include generation of test cases, generalization of attacks to different systems under tests, edge case detection through model checking and the automation thereof. They lack a implementation and therefore tend to underestimate the complexity of single steps. Notably they assume a cyber digital twin is easily derivable from the firmware image.

[7] uses the term cyber digital twins to create automated analysis of security requirements for automotive systems. The authors seem to have implemented their method and claim to have found numerous vulnerabilities in multiple ECUs. However they have not published their implementation, which leaves many technical details, such as the generation of the digital twin, unexplained.

Focusing on the emulation of ECUs, [4] introduces and publishes, a new framework based on Ghidra's [12] SLEIGH processor specifications to emulate arbitrary firmware. They motivate their framework with automotive ECUs due to their esoteric architectures and their undocumented peripherals. Chen et al. are able to emulate multiple devices with different architectures in parallel and facilitate communication between them. By lifting the instructions to their custom intermediate language and optimizing them, they are able to achieve significantly better performance than Ghidra's built-in emulation engine. They show that their emulation framework is on par with existing emulators by comparing their performance to Unicorn [13] and showing their extensibility compared to QEMU [2] by implementing various tools including a fuzzer and a symbolic execution engine. Further they evaluated their work on multiple ECUs including bare metal and Real-Time Operating System (RTOS) based systems. Their comprehensive approach is however limited to type-II and type-III systems, which is also reflected in their simplified ECU definition that only considers CAN as external interface to other ECUs. The authors also identify peripheral emulation as a major challenge,

HALucinator [5] solves the problem of emulating hardware peripherals by replacing the vendor specific hardware abstraction layer. While they base their implementation on QEMU, they still focus on microcontrollers and real time operating systems. Similar to HALucinator, NOTHORG also uses High-Level Emulation for unsupported peripherals. Unlike HALucinator we propose to write the handler functions in javascript instead of c++ thereby decreasing human labor at the cost of performance.

FIRMADYNE [3] proposed a framework to emulate Linux based commercial of the shelf IoT devices. For this they extract the filesystem from the firmware image and load the filesystem with a custom kernel into QEMU. Using the init mechanics of the firmware userspace result in a similar environment to the original device. They address common peripherals by stubbing them in their custom kernel, they try to detect network interfaces and succeed in most cases. FirmAE [8] builds upon this and improves the network detection. Both papers evaluate their work on a large number of firmware images and show effectiveness. FirmAE explicitly states that the most common reason why FIRMADYNE fails is incorrect configuration. Notably, both works have limited support for peripherals.

## 2.1 Contribution

We identify typical challenges for ECU whole-system emulation and propose our technical framework architecture for rehosting firmware and emulating peripherals to create digital twins of cybersecurity evaluation targets. As described in Section 1, some peripherals are necessary to achieve the desired runtime behavior of the digital twin, but are unavailable during the virtualized execution. NOTHORG handles these challenges by using runtime hooks to place stubs at carefully chosen locations in the firmware, which allows the firmware to run and at the same time control the input to external interfaces. We present our framework architecture and reference implementation in Section 4 and demonstrate the feasibility with three case studies in Section 5, showing that core functionality of a production ECU can be established in our digital twin even for complex communication scenarios, e.g. to evaluate security properties of the key material used for the communication protocols.

## 3 Challenges

Cybersecurity analysis is the task of finding vulnerabilities, design weaknesses and unintended hazardous behavior of the system under adversarial conditions, usually as a consequence of external interactions. Assessing the security level of a component requires testing interfaces of components and their runtime behavior to identify unwanted behavior and provoke breaking security requirements. Some parts of this analysis can be done at design time or with static analysis (e.g. by checking for known weaknesses in identified software libraries), but even in these cases, the exploitability can only be tested dynamically, i.e. when the system is operating in the intended runtime environment. Modern ECUs are composed of multiple components and interfaces, with different controllers to handle a set of responsibilities. Figure 1 shows the reference architecture of ECU targets.

Creating a synthetic runtime environment for ECUs is a labor intensive manual tasks requiring deep knowledge of the target under test. MetaEmu demonstrated that the necessary manual effort for type-II and type-III ECU rehosting can be reduced with appropriate emulation tooling to less than one day per ECU [4]. For type-III ECUs, the software complexity tends to be higher, but the abstraction provided by the OS reduces the coupling to the system architectures implementation details. We therefore aim to reduce the effort needed to re-implement common requirements when testing different ECUs over time. Special consideration is necessary for hardware-protected key material that is not part of the firmware. This is the case for Hardware Trust Anchors (HTAs) that are commonly implemented in modern vehicles as Hardware Security Modules (HSMs) and Trusted Execution Environments (TEEs) [14].

We consider the preparatory step of obtaining a suitable firmware image out of scope and assume that the tester has been provided with at least an exact copy of the firmware stored on the target of evaluation.

Setting up a digital twin for cybersecurity testing requires the analyst to provide (1) an emulated virtual execution environment for the firmware, (2) an implementation of required peripheral interfaces and (3) a setup of virtual communication infrastructure.

## 3.1 Peripherals

Peripherals pose the most significant challenge when virtualizing ECUs, as they are often coupled using undocumented or proprietary interfaces or unavailable kernel modules. At the same time, custom peripherals are abundantly used in the automotive industry and the main reason why general IoT-approaches to rehosting and digital twins fall short. In the context of ECUs, we distinguish between four types of peripherals that require different handling. Figure 2 illustrates the classification of common peripherals into the four groups described below. Note that some peripherals (e.g. the CAN bus interface) appear more than once. This is because some SoCs offer integrated support while others rely on external (On-board-)controllers to achieve the same function-wise. On some ECUs, both variants are even used simultaneously. This distinction illustrates that the firmware view is often ignorant of the actual implementation: in type-I systems, the kernel abstraction hides these implementation details and provides a uniform interface for the software. For rehosting purposes, differentiating the different peripheral types is helpful, because it allows to emulate the peripheral on different abstraction layers.

*3.1.1 System-on-Chip-peripherals.* System-on-Chip-peripherals are implemented in hardware and placed on the same package as the microcontroller. These typically include general interfaces such as controllers for Serial Peripheral Interface (SPI), Inter-Integrated Circuit ($I^2C$), CAN and Ethernet or timers, which are often supported by the Linux kernel and emulators. In this case, the emulator and kernel can be configured accordingly. Due to the long life cycles and limited product range in the automotive industry, these peripherals tend to be well supported by the Linux code base.

The most challenging peripherals in this group are trusted execution environments, e.g. based on ARM TrustZone, and integrated hardware security modules. Both are not well supported by emulators (with the exception of Open Portable Trusted Execution Environment (OP-TEE) [16] as an example implementation for the former) and usually require hardware-protected keys to fulfill their purpose, which are not easily extracted from the target.

*3.1.2 On-board peripherals.* On-board peripherals are part of the same ECU and usually controllers on the same PCB or sensors such

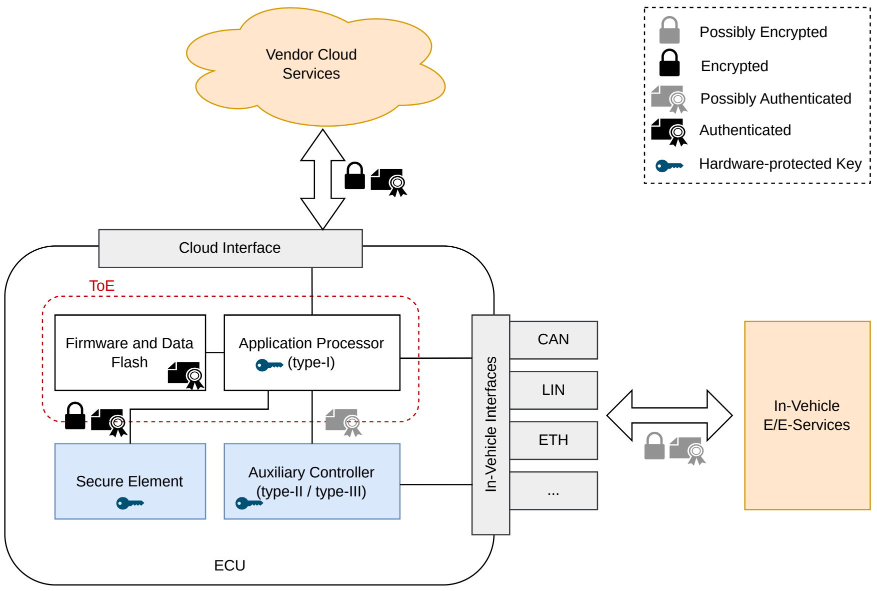


**Figure 1: Reference architecture overview showing the components of the ECU. The ToE is composed of the application processor and the Linux-based firmware. On-board dependencies (marked in blue) and external dependencies (marked in orange) need to be fulfilled and control the operation of the ToE.**

as antennas or temperature probes that are directly connected to one of the internal busses. While many System-on-Chip-peripherals are well handled by the kernel, on-board peripherals are a very heterogeneous group where the heavy customization of automotive vendors becomes apparent. Some on-board controllers such as CAN or Ethernet-transceivers are properly handled by kernel-drivers and can therefore be substituted in the virtualization. Other parts, such as (automotive-specific) HSMs or modems, are not always abstracted by the kernel. Communication for these peripherals is handled by the firmware via direct communication over the respective busses, e.g. the firmware skips the kernel device abstractions and communicates with the HSM by implementing its I²C-protocol directly.

A feature-complete emulation of these peripherals on the bus layer is complex, because it requires testers to re-implement the communication protocol in addition to the (often non-trivial) component behavior. When the communication is cryptographically authenticated with hardware-protected keys, emulation requires even more manual steps for patching out the authentication steps.

*3.1.3 E/E-Services.* E/E-Services are strongly coupled runtime dependencies of the ECU that are required to fulfill it's tasks. While they are not peripherals in the classical sense, they are part of the distributed system that forms the vehicle and are of the same importance as on-board-peripherals. They are normally connected to the ECU via in-vehicle busses such as CAN, Ethernet, LIN or FlexRay. For CAN, these services are specified by the CAN-Matrix where some frames deliver important information about the vehicle state, configuration or user interaction that are required by the target. A typical example for this group would be a tachometer that receives the vehicle state information by the odometry sensor via CAN bus. Modern vehicles also use web technologies in the car to implement communication between ECUs over Ethernet [1].

*3.1.4 Cloud-Services.* Modern vehicles depend on cloud services for integral parts of their functionality. Use-Cases such as fleet monitoring, cryptographic key provisioning and firmware over-the-air-updates are implemented over proprietary cloud interfaces. While many ECUs can still function without cloud interactions, some do not and the testing scope is notably limited when these interfaces are not considered. The interaction is usually implemented over the internet with protocols such as HTTP(S) or MQTT.

## 3.2 Cryptographic Keys

Section 1 discusses the problem of missing knowledge of required cryptographic keys. Figure 2 shows that most controllers provide cryptographic protection mechanisms and even on-board communication may be protected by cryptographic primitives to assure confidentiality and authenticity. To achieve correct behavior in the digital twin, both sides need to be emulated and provisioned with identical keys (in case symmetric cryptography is used).

## 4 Design

We designed an extendable peripheral-centric platform architecture to address the identified challenges. The design is based on the central subsystems *firmware rehosting*, *communication infrastructure* and *peripheral environment*.

[1]One example is the Volkswagen Infotainment Web Interface, https://www.w3.org/submissions/viwi-protocol/

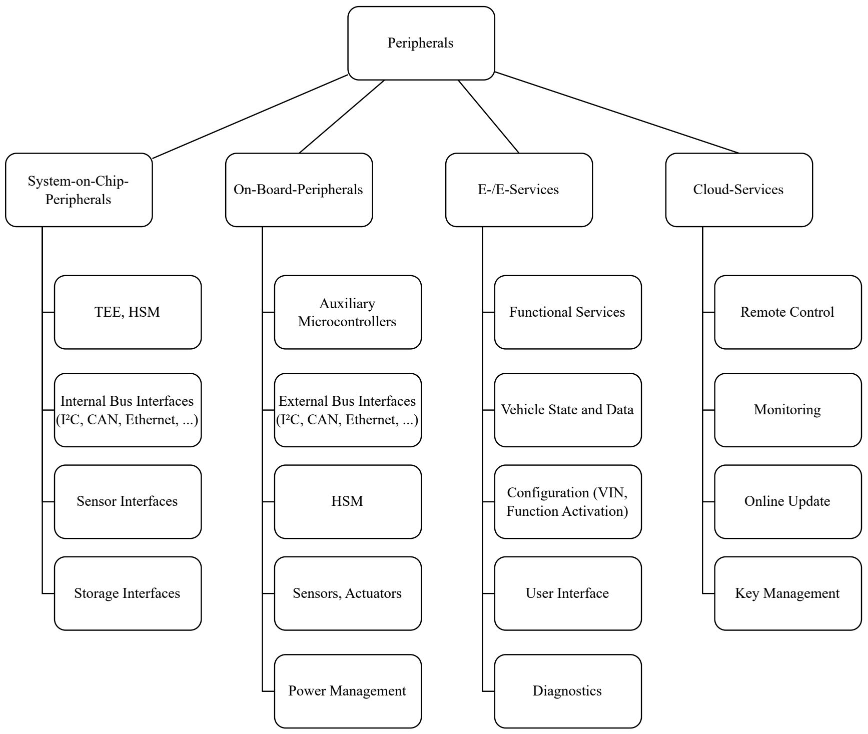


**Figure 2: Classification of peripherals into different groups.**

## 4.1 Firmware Rehosting

NOTHORG focuses on emulating automotive ECU firmware that is based on Linux. NOTHORG expects to have a functioning kernel and root filesystem. In practice, we propose to discard the original kernel and build a new kernel. This allows for more flexibility during emulation (e.g. by using a different target board for the emulation) and analysis (e.g. allowing for Extended Berkeley Packet Filter (eBPF) instrumentation). However, doing this means giving up on existing kernel modules that were used in the original firmware. We argue that this is a reasonable trade off, since custom kernel modules seldom provide features apart from supporting custom peripherals, which are not supported by the emulator anyway. NOTHORG works with arbitrary emulators, but is built around the feature-set of QEMU.

## 4.2 Peripheral Environment

In this section, we focus on the *peripheral environment*, which is responsible for providing the emulation with access to the peripherals it depends on. After identification of the required peripheral interfaces, our architecture allows the use of different providers to integrate the peripheral interface. Table 1 shows which mechanisms are suitable for the different peripheral classes.

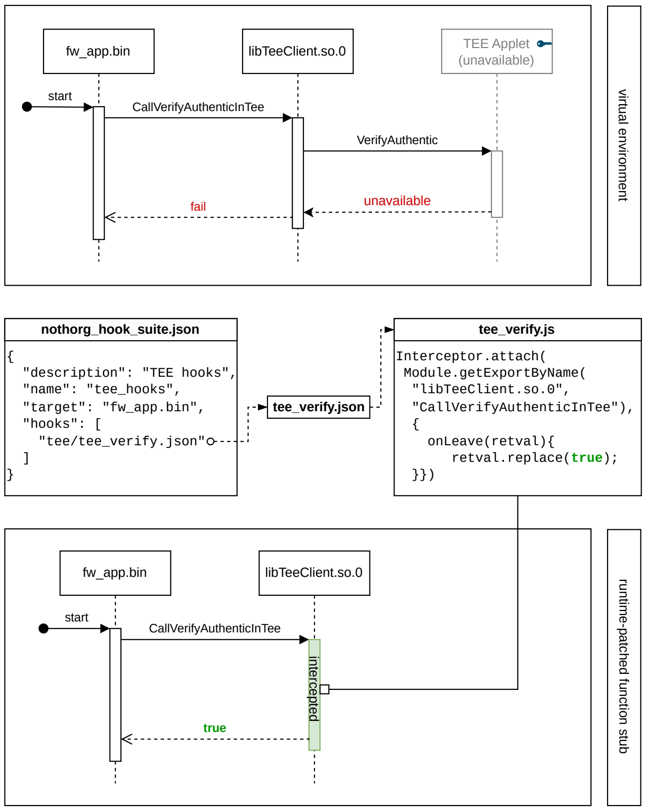
*4.2.1 Emulation.* Emulation can be used to provide a (basic) implementation of a required peripheral without providing the peripheral or a rehosted firmware of the peripheral. Emulation is the preferred method for analyses, because it allows reproduction of complex interactions, but also allows to conduct "what-if" tests for evaluating the behavior in uncommon cases. Our analysis in Section 3 shows that the integration interface of peripherals is highly target dependent, the best abstraction for emulation should be a choice of the analyst. NOTHORG therefore provides emulation capabilities not only for kernel devices, but also for arbitrary function calls inside running processes. This versatility makes emulation of virtually all peripherals possible and reduces the effort for reimplementing kernel drivers for custom components. We consider the efficiency of the emulation implementation as one of the most important achievements of NOTHORG: Instead of reverse engineering drivers on the kernel level, implementing the emulator on the userspace library wrapping the kernel interface suffices. While this requires static analysis to find the right place, and make sure that the stub behaves somewhat like the original peripheral, it usually

**Table 1: NOTHORG providers for handling different peripheral classes**

| Peripheral Class | NOTHORG providers |
|---:|:---:|
| System-on-Chip | Emulation, Stubbing |
| On-Board | Emulation, Stubbing, Passthrough |
| E/E-Services | Emulation, Stubbing, Injection, Passthrough |
| Cloud-Services | Emulation, Stubbing |

is much simpler than implementing the emulator at a lower level. This generalizes well to other troublesome components like Trusted Platform Modules (TPMs), HSMs or TEEs. Since, in the automotive setting, software components from multiple vendors are often integrated and thus encapsulated well, we find that this approach works exceptionally well.

*4.2.2 Stubbing.* In some cases, the presence of a peripheral is required, but incorrect data received from peripherals does not impact the analysis. In this case, stubbing can be used to return satisfactory values to the firmware to continue operation. When emulation of the peripheral is infeasible or too complex, stubbing is often sufficient to achieve a full system initialization. Figure 3 shows how runtime hooks are used to create stubs for peripherals in the virtual environment.



```
{
  "description": "TEE hooks",
  "name": "tee_hooks",
  "target": "fw_app.bin",
  "hooks": [
    "tee/tee_verify.json"
  ]
}
```

```
Interceptor.attach(
 Module.getExportByName(
  "libTeeClient.so.0",
  "CallVerifyAuthenticInTee"),
  {
   onLeave(retval){
      retval.replace(true);
  }})
```

**Figure 3: Example of stubbing a critical TEE-call that would otherwise fail in the virtual environment.**

*4.2.3 Injection.* Injection allows to replay previous captures of communication busses. E.g. a CAN or Ethernet capture can be replayed during runtime. If the captures are crafted carefully, injection is the simplest method to put the ToE in an initialized and running state, but is only suited for non-interactive protocols. The potential of manipulating input is low.

*4.2.4 Passthrough.* Passthrough is an optional, but powerful option to connect the digital twin with a physical environment. Passthrough is limited to devices supported by the emulator via the kernel interface, but allows to observe the interaction between ToE internals and another ECU or even the complete vehicle network. In combination with the capture module, this mode can be used as an effective starting point for reproducing the system state with emulator implementations.

## 4.3 Instrumentation Mechanisms

Interfacing the ECU runtime to achieve the functionality described above and provide enough flexibility to fit different ToEs from multiple vendors with an efficient workflow excludes several promising approaches that are used elsewhere. The `LD_PRELOAD`-mechanism used by FirmAE to intercept a commonly used NVRAM-library only works for shared library symbols.

*4.3.1 Runtime-Hooking.* The most fundamental concept used by NOTHORG to implement emulation, stubbing and logging is dynamic instrumentation, to which we refer to as runtime hooking to be concise. Runtime hooking binaries in the firmware allows introspection, centralized logging and tracing and executing code on arbitrarily chosen function calls. NOTHORG provides the ability to modify, upload and inject hooks during runtime and thus facilitates a short feedback loop without restarting the whole system emulation. During a typical component assessment process, it is expected that the tester needs to write numerous hooks to achieve the desired operation sequence in the ToE, possibly in multiple different variants. For this reason, hook management is a core feature in NOTHORG that allows the efficient grouping and organization of reusable hooks.

*4.3.2 Filesystem patching.* Filesystem patching is done before the ToE is executed by the emulator. It is a vital step to enable runtime hooking, which is controlled by a process in the target system, but also serves to change the ToE configuration or startup flow to simplify testing (e.g. the firewall might be reconfigured). Sometimes, adaptions of the ToE file systems are necessary to ensure the compatibility with the emulator. In NOTHORG filesystem modifications can be provided as an overlay folder that is copied over the ToE filesystem before each run. This allows the changes to be temporary and original files to be preserved.

*4.3.3 Kernel Device Substitution.* Peripherals that are interfaced through the OS abstraction (i.e. the kernel device provided by a kernel driver) can in some cases be substituted by another device that is controlled by the analyst or passed through by the host system. This mechanism is based on the emulator and is primarily used to provide the *passthrough* functionality.

## 4.4 Configuration

The setup process can be labor intensive for complex ECUs and always requires manual work. To provide flexibility, NOTHORG is designed to be modular and extensible. The central configuration of NOTHORG is specified in a single YAML file. Users can reuse large parts of configuration for different ECUs and firmware images. Besides similar configurations, this also enables the user to share hooks and extensions between multiple projects, thereby reducing the amount of manual work per ECU. A secondary benefit of this is that the configurations can be given to other users who might not have the same knowledge about rehosting and emulation.

## 4.5 Logging

Logging is helpful to understand how components work and why they fail. Even production code often provides logging facilities to help reproduce failures in production. Using runtime hooking, NOTHORG allows redirecting log streams to the logging subsystem, where it can be processed and filtered by the user. Hooking common logging libraries captures nearly all logging output of the firmware with few portable hooks. We show a practical example in Section 4.7.

This approach also allows to capture debugging log statements that remain in the code but are not normally displayed due to the logging configuration. Combined with additional logging information generated by the hooks themselves, a fine-grained feedback about the internal state can be achieved.

## 4.6 Basic Workflow

As motivated in Section 1, NOTHORG aims at providing an environment for effective security analyses.

We assume that the evaluation goals are defined beforehand and guide the decisions made in the following setup. Figure 4 shows the typical workflow when using NOTHORG for cybersecurity testing, which consists of three phases.

*4.6.1 Rehosting Preparation.* Before configuring NOTHORG, the rehosting needs to be configured for the target platform. Depending on the target, this includes building a kernel and choosing a suitable emulation configuration. While in theory the kernel of the target can be used to obtain more reproducible results, the use of a custom built kernel has several advantages and simplifies the emulation setup [6]. An important part of the preparation is the analysis of hardware interfaces and networks of the target, as they are used in the next phase to define a minimal environment.

*4.6.2 Environment Definition.* The user starts by providing an initial configuration `nothorg.yaml`, which includes the kernel image, the root filesystem and the network configuration and selects or writes initial hooks to collect logs and provide a minimal set of peripheral emulation definitions or stubs. For this step, predefined hook-suites for commonly used logging frameworks help to reduce the time to find the correct hooks.

*4.6.3 Cybersecurity Evaluation.* Once the environment definition leads to an operational virtual execution setup, the instrumentation can be iteratively refined with information gained from logging and debugging. In case the logs and operation states indicate missing peripheral dependencies, these can usually be emulated or stubbed quickly with additional hooks.

To not lose information, NOTHORG creates a copy of the filesystem and patches user changes before each execution. After starting the emulation, NOTHORG attaches runtime hooks and instrumentation to the emulation, and presents the user with useful output like logs or network captures. The user can refine what is logged and captured to focus on specific parts of the firmware. The user can add new hooks and instrumentation to overcome problems arising in the emulation.

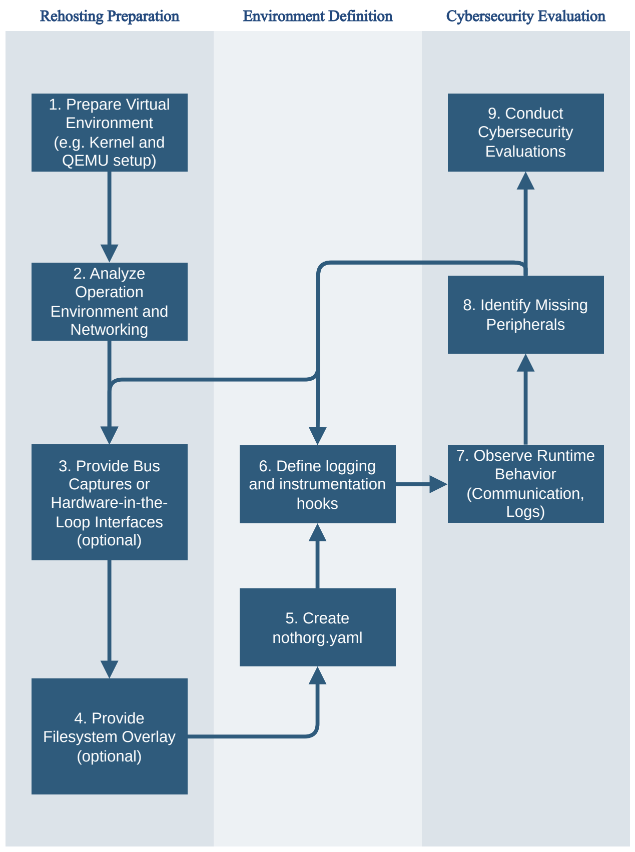

Rehosting Preparation
Environment Definition
Cybersecurity Evaluation
1. Prepare Virtual Environment (e.g. Kernel and QEMU setup)
2. Analyze Operation Environment and Networking
3. Provide Bus Captures or Hardware-in-the-Loop Interfaces (optional)
4. Provide Filesystem Overlay (optional)
5. Create nothorg.yaml
6. Define logging and instrumentation hooks
7. Observe Runtime Behavior (Communication, Logs)
8. Identify Missing Peripherals
9. Conduct Cybersecurity Evaluations

**Figure 4: Typical workflow to setup a digital twin for cybersecurity testing with NOTHORG. After analyzing the environment and peripherals (steps 1-3), the configuration and target modifications are prepared (steps 4-6) before the execution can be observed and used for runtime evaluations (steps 7-9).**

## 4.7 Hook Management

As described in Section 4.3.1, NOTHORG relies heavily on runtime hooks provided by the user. Since those hooks are written by the user and can potentially be reused for other firmware images, NOTHORG groups single hooks into *hook suites*, which combine hooks for one library or component. Our implementation uses the *Frida* framework [15] to write and inject hooks into the target runtime. Frida hooks depend on symbols instead of offsets, they can be reused whenever the library is reencountered in a different firmware image. Hooks should be changeable to allow for easy debugging and hook development. Further NOTHORG provides infrastructure to load and reload hooks and to manage their output, differentiating between logging and analysis output and between hook suites.

To show the flexibility of runtime hooks, we instrument the firmware's logging facilities provided by Android's liblog. Instead of hooking the entry point *android_log_print*, we target the internal function *android_log_write*, which is called after the varargs are resolved and can be handled with a simpler hook:

```
Interceptor.attach(
 Module.getExportByName("liblog.so.0",
  "__android_log_write"),
 {
  onEnter(args){
   const prio = args[0].toInt32();
   const  tag = args[1].readCString();
   const  msg = args[2].readCString();
   console.log([prio,tag,msg]);
  }
 })
```

The hook is accompanied by a json file that describes the hook,

```
{
 "name": "android_log_write_hook",
 "description": "hook for android_log_write",
 "path": "liblog/android_log_write_hook.js"
}
```

Now this hook can be added to a hook suite. Hook suites are collections of hooks that should be injected in a common target process and share a similar purpose. In our experience, this often results in one hook suite per process. Hook suites are again defined by a json file. To continue our example, we would like to analyze a process called *target_proc*. For that test, our hook suite could look like this:

```
{
    "description": "hook suite for target proc",
    "name": "target_hooks",
    "target": "target_proc",
    "hooks": [
        "liblog/android_log_write_hook.json",
    ]
}
```

At startup, NOTHORG loads all hooks and hook suites and attaches the hooks to the processes corresponding to the target-definition. NOTHORG allows for hook suites to be reloaded at runtime which allows for iterative debugging and development of hooks.

## 5 Case Study

This section presents a case study on the analysis of ECU firmware. We pseudonymized port numbers, names and addresses in the remainder as to not reveal the identity of the manufacturer. We replaced the kernel of the firmware with a self-built version and were able to boot the ECU in QEMU. After stubbing essential peripherals, like the TPM, most applications reached a running state. Using the hook from Section 4.7, logs from the relevant processes could be captured in NOTHORG.

### 5.1 Case Study 0: logging and peripherals

Upon starting the ECU with NOTHORG and the logging hook, we find that the ECU tries to initialize communication via SPI and waits for a response. To continue our analysis, we skip the initialization of the connection by placing a hook in the current function and modifying the wait-loop condition variable to break early. This allows the rest of the setup to continue.

### 5.2 Case Study 1: A https service

By analyzing the logs, we find that the ECU tries to connect to an IP *192.168.0.5* on port *1234*. We provide this service by specifying a python server with the corresponding address and port in the NOTHORG configuration file and restart the emulation. Looking at the packets we see that a TLS handshake is tried, but not completed. Refining our stub to accept TLS connections we were not able to perform the handshake. After identifying the set of used ciphers by placing additional tracing hooks, we further find that the connection is secured with a pre-shared key that is stored in an encrypted TEE partition and therefore unavailable in the digital twin. We then added a hook to initialize the connection as plaintext http instead of https, which facilitated the communication with our peripheral emulation and enabled us to understand the protocol and test the implementation's resilience against maliciously crafted responses. Further, NOTHORG's capture and replay functionality allows us to capture the communication and replay it later. This shows that, using runtime hooking, security testing of the application layer is possible even without knowledge of the pre-shared keys.

### 5.3 Case Study 2: SPI protocol

Going back to the SPI communication, we previously stubbed away, we are now interested in what is communicated over SPI. Since SPI is quite difficult to emulate in QEMU and our board does not support SPI, we fall back to hooking the *ioctl* syscall used by the firmware to communicate over SPI. Our first step is to replace the return value of the failed *ioctl* calls (as no SPI device is available in the emulation) to return a non-failure value. By reading the log output, we can observe that the firmware tries to complete a challenge-response-protocol over SPI with apparently random generated challenges. When emulating the SPI endpoint, by sending random SPI messages and observing the log output we quickly find several interesting data flow paths. Looking at the call sites of the log calls we find a method that is responsible for parsing messages in their respective message queues. Further analysis shows that we can craft a message to be sent to a specific message queue intended for messages from the CAN bus. Thus we assume that the ECU does not access a CAN bus directly, but instead CAN messages are relayed from the auxiliary controller connected to the SPI bus. This is a major finding that would not have been possible or at least significantly harder with static analysis alone. Stepping back to the challenge response protocol we discover further that some message queues expect encrypted messages that are decrypted by invoking the TEE before being passed to the message queues.

### 5.4 Case Study 3: VIN setup

Now being able to send CAN messages to the ECU, we are able to check which CAN ID's are used on the bus. We recognize that at

CAN ID 0x131, the ECU logging shows an error because the Vehicle Identification Number (VIN) ID is expected to have 17 characters. Sending the 17 bytes 'AAAAAAAAAAAAAAAAA' the ECU seems to accept the new VIN. We check this by using the Unified Diagnostic Services (UDS) implementation that seems to be running on the ECU. For this we connect to the default Diagnostics over IP (DoIP) port 13400 and request the VIN. The response VIN received is the same we sent earlier, thus we assume that the ECU is now using this VIN. This can be confirmed with a logging hook. We can conclude that the protocol messages for setting the VIN are not authenticated.

### 5.5 Case Study 4: Fuzzing

Building on Case Study 2, we implemented a black-box fuzzing experiment for the SPI protocol. A benefit of this is that it furthers our understanding of the protocol and ECU behavior while testing the message parser implementation for vulnerabilities. Even though the reference implementation based on a cross-platform QEMU emulation and javascript Frida hooks limits the throughput that is available for input fuzzing, a simple python implementation for generating random payloads was sufficient to find inputs that produce a crash on the ToE.

Since we control the userspace of the ECU, we were able to attach a debugger and analyze the crash in real time. This setup allows for simpler debugging and proof of concept development that can be used to communicate and fix identified flaws. In this case, the crash was caused by a unguarded `memcpy`-call that copies an unchecked amount of data into a fixed size stack buffer. Although this vulnerability class is trivial, our analysis shows that it still exists in modern ECUs and the potential of NOTHORG to be used for security testing components and protocols that are untestable with classic techniques. We expect that by using a modern fuzzer with sophisticated input optimization such as fandango [17], fuzzing interfaces with NOTHORG is a practical approach even for more complex code paths.

## 6 Conclusion

Digital twins for ECU cybersecurity testing based on rehosting and peripheral emulation are a powerful tool for security testing. Our proposed framework demonstrates the feasibility of various dynamic analyses of large and complex systems. Further, they also provide a platform to test attacks and exploits in a controlled and transparent environment. Finally this introspection allows for a multitude of specialized applications such as fuzzing, vulnerability enumeration and flexible testing strategies for assessing internal confidentiality guarantees for key material.

Our digital twin framework architecture NOTHORG allows rehosting and virtualizing full ECU runtime environments by using system emulators with a preprocessed firmware image and achieve arbitrary emulation states with runtime hooks, replay capabilities from bus captures and configurable logging. The fast and incremental hook management makes it a valuable tool for deeper practical analysis of security properties that would be infeasible with state-of-the art black-box penetration testing. Our evaluation using a reference implementation based on QEMU and Frida shows that real-world black box tests can be realized effectively and allow emulation of non-available hardware functions. We use our framework to setup a digital twin of a modern ECU for cybersecurity testing and thus show the feasibility of our approach.

## Acknowledgments

This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs, Karlsruhe.

## References

[1] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. 2024. Pandawan: quantifying progress in linux-based firmware rehosting. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5859–5876.

[2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA.

[3] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. doi:10.14722/ndss.2016.23415

[4] Zitai Chen, Sam L. Thomas, and Flavio D. Garcia. 2022. MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 515–529. doi:10.1145/3548606.3559338

[5] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. 1201–1218.

[6] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. 2021. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ACM, Virtual Event Hong Kong, 687–701. doi:10.1145/3433210.3453093

[7] Ana Cristina Franco da Silva, Stefan Wagner, Eddie Lazebnik, and Eyal Traitel. 2023. Using a Cyber Digital Twin for Continuous Automotive Security Requirements Verification. *IEEE Software* 40, 1 (Jan. 2023), 69–76. doi:10.1109/MS.2022.3171305

[8] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 733–745. doi:10.1145/3427228.3427294

[9] Stefan Marksteiner, Slava Bronfman, Markus Wolf, and Eddie Lazebnik. 2021. Using Cyber Digital Twins for Automated Automotive Cybersecurity Testing. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 123–128. doi:10.1109/EuroSPW54576.2021.00020 ISSN: 2768-0657.

[10] Stefan Marksteiner and Zhendong Ma. 2019. Approaching the Automation of Cyber Security Testing of Connected Vehicles. In *Proceedings of the Third Central European Cybersecurity Conference (CECC 2019)*. Association for Computing Machinery, New York, NY, USA, 1–3. doi:10.1145/3360664.3360729

[11] 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 *NDSS*. doi:10.14722/ndss.2018.23166

[12] National Security Agency. 2019. *Ghidra*. Retrieved Sep 2, 2025 from https://github.com/NationalSecurityAgency/ghidra

[13] Anh Quynh Nguyen and Hoang Vu Dang. 2015. Unicorn: Next Generation CPU Emulator Framework. Retrieved Sep 2, 2025 from https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[14] Christian Plappert, Dominik Lorych, Michael Eckel, Lukas Jäger, Andreas Fuchs, and Ronald Heddergott. 2023. Evaluating the applicability of hardware trust anchors for automotive applications. *Computers & Security* 135 (2023), 103514. doi:10.1016/j.cose.2023.103514

[15] Ole André V. Ravnås. 2013. *Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* Retrieved Sep 2, 2025 from https://frida.re/

[16] Trusted Firmware Project. 2014. *Open Portable Trusted Execution Environment.* Retrieved Sep 2, 2025 from https://www.op-tee.org

[17] 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.