Protocol-Aware Firmware Rehosting for Effective Fuzzing of Embedded Network Stacks

Moritz Bley
moritz.bley@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Tobias Scharnowski
tobias.scharnowski@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Simon Wörner
simon.woerner@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Moritz Schloegel
moritz.schloegel@asu.edu
Arizona State University
Tempe, AZ, USA

Thorsten Holz
thorsten.holz@mpi-sp.org
Max Planck Institute for Security and Privacy
Bochum, Germany

Abstract

One of the biggest attack surfaces of embedded systems is their network interfaces, which enable communication with other devices. Unlike their general-purpose counterparts, embedded systems are designed for specialized use cases, resulting in unique and diverse communication stacks. Unfortunately, current approaches for evaluating the security of these embedded network stacks require manual effort or access to hardware, and they generally focus only on small parts of the embedded system. A promising alternative is *firmware rehosting*, which enables fuzz testing of the entire firmware by generically emulating the physical hardware. However, existing rehosting methods often struggle to meaningfully explore network stacks due to their complex, multi-layered input formats. This limits their ability to uncover deeply nested software faults.

To address this problem, we introduce a novel method to automatically detect and handle the use of network protocols in firmware called Pemu. By automatically deducing the available network protocols, Pemu can transparently generate valid network packets that encapsulate fuzzing data, allowing the fuzzing input to flow directly into deeper layers of the firmware logic. Our approach thus enables a deeper, more targeted, and layer-by-layer analysis of firmware components that were previously difficult or impossible to test. Our evaluation demonstrates that Pemu consistently improves the code coverage of three existing rehosting tools for embedded network stacks. Furthermore, our fuzzer rediscovered several known vulnerabilities and identified five previously unknown software faults, highlighting its effectiveness in uncovering deeply nested bugs in network-exposed code.

CCS Concepts

• **Security and privacy → Systems security**; **Software and application security**; • **Computer systems organization → Embedded systems**.

Keywords

Firmware Fuzzing, Rehosting, Network Fuzzing, Software Security

1 Introduction

Embedded devices are specialized computers that tightly integrate with the hardware they control. Examples include medical devices, industrial control systems, and Internet of Things (IoT) components. These devices are either powered by lightweight operating systems (OSes) or operate without an OS (referred to as “bare metal”). Consequently, the software running on these systems, known as *firmware*, cannot rely on conventional OS abstractions but must directly manage its interactions with hardware. Furthermore, embedded devices often expose network-related applications through Embedded Network Stacks (ENS) to allow for interconnectivity with other devices. The combination of missing OS abstractions, limited computation resources, and the large attack surface has made embedded systems challenging to secure, especially via automated techniques.

Recent research has turned towards *firmware rehosting* to achieve scalable and effective testing of embedded firmware. Generally speaking, rehosting runs firmware within an emulator instead of its original, resource-constrained hardware environment [31, 48]. Using a fuzzer to mimic peripheral data allows firmware to be executed on powerful hosts without relying on the physical microcontroller [13, 19, 39, 41, 43, 49].

However, testing firmware that uses networking via rehosting-based fuzzing still faces two substantial challenges: First, while AFL-like mutations [50] are effective in testing lightly structured formats, current solutions are ill-equipped to progress past the initial layers of embedded network stacks. Consequently, for firmware that utilizes a complex ENS, current rehosting-based firmware fuzzers are stuck, as they fail to generate the highly complex structure of network packets. This forces them to re-test only the low-level logic that verifies the packet structure. In particular, they fail to test higher network layers or the application-level logic built on top of the ENS. Second, while network protocol fuzzers exist in the general-purpose domain [2, 4, 35], their applicability to embedded firmware is limited. For example, AflNet [35] relies on existing (target-specific) traffic captures for testing. While such a capture is easy to generate on a general-purpose system, this is not the case in the embedded domain. Aside from various practical challenges, the requirement of a hardware setup to bootstrap the testing process results in a tedious manual process. This effectively negates the scalability effects that rehosting provides. Similarly,

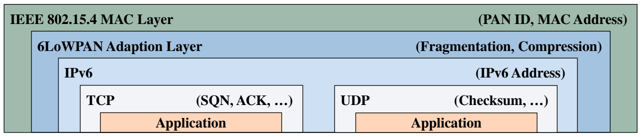


Figure 1: Example network layers involved in a firmware that exposes applications through a typical 6LoWPAN stack.

EmNetTest [2] also relies on a set of seed network packets, for which it systematically generates variants to test the known layers of an ENS. Although more broadly applicable to embedded network stacks, EmNetTest requires source code and knowledge about the target to specify network metadata. Both are rarely available in practice. Furthermore, it does not provide fuzz testing capabilities for the firmware application logic beyond the transport layer.

Based on these observations, we identify two properties of an optimal rehosting solution for effective fuzzing of ENSs: First, neither a physical hardware setup nor target-specific configurations should be required, as both introduce manual effort. More specifically, this means that *inputs need to be bootstrapped without any seeds*, and any *network protocols need to be automatically inferred* from firmware behavior. Second, the desired solution should allow the fuzzer to test *not only common network layers*, but also the *application logic that is specific to each firmware.*

We need to solve different challenges to achieve such automation and testing flexibility. First, we must generate well-formed messages for embedded network protocols. Figure 1 shows a typical ENS that exposes IPv6-based communication through 6LoWPAN over IEEE 802.15.4 radio frames. The firmware expects to receive low-level radio frames, which it decompresses and re-assembles into IPv6 packets. To enable a fuzzer to test the application logic of the firmware under test, we need to wrap application-layer data into a series of low-level radio frames that successfully traverse all network layers and deliver the fuzzing payload to the application logic. This is in contrast to typical general-purpose network fuzzing scenarios, where the operating system abstracts away most network layers via network sockets.. Given that we do not have any information on the ENS of the firmware, we need a mechanism that allows us to automatically discover the tree of different network protocols via which the ENS expects to communicate. Additionally, we also need to infer the static device identifiers by which the firmware expects to be addressed, such as the PAN ID, as well as the MAC and IPv6 addresses. After deriving this information, we can focus on generating valid network packets that encapsulate fuzzing inputs. However, to fully explore the ENS, the fuzzer also needs to be able to account for the dynamic state of protocols. For example, Figure 2 displays how the DHCPv6 protocol relies on a defined sequence of messages, including DHCPv6 solicit, advertise, request, and reply packets that must occur in a specific order. Only if the rehosting environment correctly handles these exchanges can the fuzzer fully explore the protocol implementation and any application built on top of it.

To this end, we design and implement Pemu, a generic framework that extends existing rehosting platforms to handle network communications. At a high level, our approach provides a self-configuring virtual network interface that connects a network-unaware fuzzer and the firmware under test. Pemu encapsulates raw fuzzing input into valid network packets, enabling fuzzing input to reach deeper into network-related code, all the way into the application logic. To infer the network protocols the firmware under test uses, Pemu performs active probing. We apply two techniques to monitor how the firmware responds to different types of probing packets. First, in contrast to traditional rehosting systems that largely ignore firmware output, we parse the outgoing low-level frames of the firmware to detect available network protocols. Second, we evaluate the firmware coverage with different types of probing packets. This allows us to detect which types of packets trigger unique coverage in the firmware, i.e., which types of packets its ENS is sensitive to. By interleaving fuzzing and active probing based on previously detected network layers, Pemu iteratively recovers the tree of network protocols that the ENS uses. By encapsulating fuzzing inputs according to the identified protocol tree, Pemu allows the fuzzer to test the implementation of different layers of the ENS, as well as the application logic of the firmware.

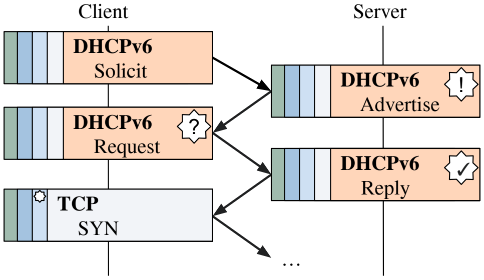


Figure 2: Exchange of context-sensitive messages involved in an IPv6 address retrieval via DHCPv6 between a client (left) and server (right).

To demonstrate the versatility and applicability of our approach, we integrate Pemu with three popular rehosting-based firmware fuzzers: Fuzzware [39], Hoedur [41], and SEmu [54]. We then evaluate Pemu's performance on a firmware sample set that covers a wide variety of network protocols. Our results show that Pemu consistently improves code coverage for the tested firmware samples. More specifically, our approach increases the average basic block coverage by 40.7% when used together with Fuzzware, by 39.2% with Hoedur, and by 8.5% with SEmu. Moreover, we evaluated Pemu against EmNetTest, the only other technique specifically targeting ENS testing. Using EmNetTest's dataset of 12 real-world vulnerabilities, we show that Pemu not only successfully rediscovered all of them, but Pemu also identified three additional bugs missed by EmNetTest. Combined with two further novel bugs—one of them in the SEmu dataset—Pemu managed to discover five new bugs in network-exposed code throughout our evaluation. We are in the process of disclosing all of our findings to the affected vendors in a coordinated way and one of them has been fixed.

Contributions. We provide the following three key contributions:

- We present a new approach for dynamically detecting the use of network protocols in embedded firmware. Based on the identified network interactions, we propose a method to encapsulate fuzzing data in valid network packets based on a *virtual network* to effectively test different layers of network interactions.

- To demonstrate the general applicability of our method across different fuzzing frameworks, we integrate our approach into three frameworks: Fuzzware, Hoedur, and SEmu.
- In a comprehensive ablation study, we show that Pemu significantly improves code coverage across all application scenarios and outperforms existing techniques on vulnerability benchmarks by finding five previously unknown bugs.

To foster research on this topic, we publish the source code, our sample data set, and other research artifacts at https://github.com/MPI-SysSec/pemu.

2 Background

We first provide a brief overview of firmware and the specific challenges involved in analyzing the security of their network stacks.

2.1 Firmware in Embedded Systems

Embedded systems tightly integrate hardware and software to perform specific tasks within devices ranging from IoT gadgets to industrial systems [18, 34]. Unlike general-purpose computers, they typically contain a microcontroller unit (MCU). This self-contained system typically includes a processor, memory, and various on-chip peripherals such as Serial Peripheral Interface (SPI), Universal Asynchronous Receiver-Transmitter (UART), or Direct Memory Access (DMA). These peripherals allow connectivity to off-chip components like Ethernet controllers, Bluetooth modules, wireless modems, or LED screens. The wide range of customized processors and peripherals, both on-chip and off-chip, adds to the complexity and diversity of these systems.

Firmware, the software controlling an embedded system, is usually built on either a minimal embedded operating system (EOS) or directly on the hardware (referred to as “bare metal”), resulting in fewer abstraction layers than in general-purpose software. Furthermore, embedded applications are strongly integrated with either the EOS or the system’s underlying hardware. Firmware components, such as network stacks and libraries, are specifically tailored to the hardware and closely integrated, which makes standard security analyses challenging and calls for customized techniques.

2.2 Linux-based Fuzzing

Fuzzing is a well-established security testing technique that generates semi-random inputs to identify unexpected or crash-inducing behavior in a system [14, 27]. However, fuzzing embedded systems on real hardware is often impractical due to poor scalability and limited observability [31]. For Linux-based, type 1 [31] devices like routers or IP cameras, previous research has focused on fuzzing user-space applications. This is either done by emulating the target application in QEMU user-mode [45] or on top of a customized kernel in QEMU system-mode [7, 21]. In both cases, the primary focus of these approaches is setting up the target’s Linux-related environment, such as the file system or network interfaces. To this end, they use well-known kernel interfaces, like system calls, to collect data such as required files and IP addresses.

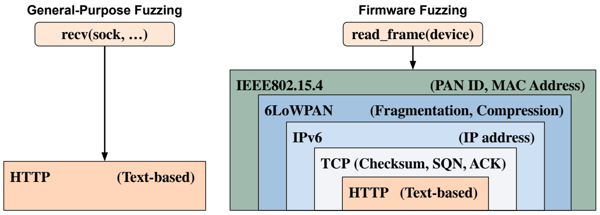


Figure 3: Comparison of the input formats required for testing an HTTP server implementation between general-purpose fuzzing and firmware fuzzing: given the lack of abstractions provided by an OS, a firmware fuzzer needs to deal with all individual network layers.

2.3 Rehosting-based Fuzzing

In contrast, applications running on tightly integrated type 2 and type 3 EOSs [31] cannot be easily emulated by these Linux-based approaches. This is because these EOSs do not provide the same high-level abstractions that Linux provides to mask the underlying hardware complexity. Instead, *firmware rehosting* addresses the challenge of embedded fuzzing by emulating bare-metal or EOS-based firmware on commodity hardware, enabling the use of dynamic analysis techniques that are otherwise unfeasible on embedded devices. By decoupling the firmware from specific hardware dependencies, rehosting provides a more flexible and scalable platform for security analysis. Recent approaches have further reduced the need for exact peripheral emulation by using generic abstraction models [10], automatic peripheral interaction modeling [13, 39], and customized input formats [9, 41]. These techniques support effective fuzzing by providing raw input to the firmware through modeled peripherals.

2.4 Fuzzing Embedded Network Stacks

While rehosting-based fuzzing enables broad firmware testing, it still falls short when dealing with more complex components, especially network stacks and the applications running on top of them. Network stacks consist of multiple interdependent layers, and data is passed to higher layers using complex formats and specific sequences. In general-purpose systems, user-space fuzzing of network services benefits from OS-level abstractions, e.g., a fuzzer can inject test cases directly into a program via system calls like `recv`, without needing to construct the underlying network protocol layers. In contrast, embedded firmware fuzzing lacks such abstractions. As illustrated in Figure 3, this lack of abstraction makes fuzzing far more complex. Instead of directly testing the application layer, the fuzzer must provide input that satisfies *all* underlying network layers (i.e., from IEEE 802.15.4 to 6LoWPAN and IPv6 up to TCP and the actual HTTP payload). This means the fuzzer must generate inputs that satisfy the expectations of every layer (from the data link layer up to the application) without any OS assistance. In contrast, Linux-based fuzzing uses readily available network interfaces provided by the Linux kernel to send application data directly to the target process [7, 17, 21].

As current state-of-the-art fuzzers and rehosting frameworks offer no notion of message encapsulation or internal state, they are ill-equipped to test such complex components. A fuzzer with a

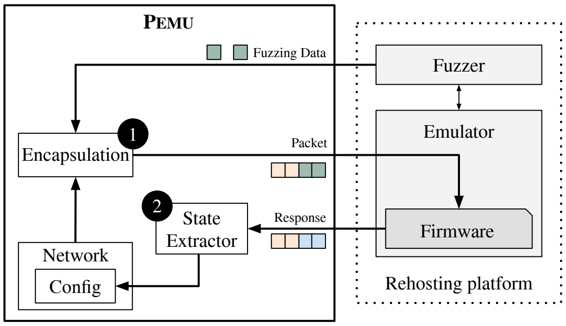


Figure 4: High-Level Overview of Pemu's architecture

focus on bitflips is unlikely to craft input that resembles a nested packet valid on different layers, each with its unique constraints. At the same time, existing fuzzers have no notion of statefulness between messages, nor do they explicitly account for values that must be maintained across individual messages.

3 Design

While fuzzing network applications is well studied for general-purpose systems, embedded systems pose unique challenges that remain largely unaddressed. State-of-the-art firmware fuzzers typically provide only raw data to the network stack, which fails to match the complex syntax and semantics of network protocols. As a result, the network stack often discards packets generated during fuzzing before they can reach deeper code sections, including custom application logic. This limits the fuzzer's effectiveness and leads to repetitive, ineffective attempts to bypass lower-layer constraints such as checksums, static addresses, or incrementing values. Existing firmware fuzzers do not effectively address this limitation. Using captured network traffic as seeds, as AflNet does, has several drawbacks. Rehosting is supposed to work without physical hardware, so relying on real traffic undermines this benefit. Additionally, mutating such seeds regularly breaks checksums or protocol structures. Fixing this either requires source code access (which rehosting avoids) or checksum recalculation after every mutation, making it rather expensive.

To overcome these limitations, we present the design of Pemu, a virtual network that delivers realistic, syntactically valid packets to embedded firmware under test. Through a transparent and self-configuring encapsulation mechanism, Pemu allows the rehosting platform to request and deliver well-formed network packets just in time. Pemu also analyzes firmware-transmitted packets, from which it can extract important network values, such as addresses and sequence numbers. By enforcing both syntactic and semantic correctness at each network layer, Pemu enables fuzzing at the application layer, bypassing lower-layer constraints like checksum and header validation. Based on this foundation, we introduce two automated analysis methods that identify protocols and protocol-specific values in the firmware *without* requiring prior knowledge or manual intervention.

3.1 High-Level Overview

From the firmware's perspective, Pemu dynamically creates a *virtual network* that facilitates valid network interactions between the firmware and a fuzzing-based rehosting platform. To achieve this, Pemu consists of multiple components (see Figure 4) that interact with each other. When the rehosting platform detects that the firmware is in a state where it can receive a packet, the encapsulation module generates such a packet by using raw fuzzing input from the rehosting platform's fuzzing module. It then encapsulates this input into a valid network packet based on the current (initially empty) network configuration (❶). Guided by the fuzzer, the encapsulation module adjusts how deeply it encapsulates the input, allowing targeted testing of different layers in the network stack. Once the firmware transmits low-level frames, Pemu parses these frames to extract information about the current protocol state, e.g., source and destination addresses (❷). This state is later used by the encapsulation module to encapsulate network packets correctly. The network is initialized with an empty configuration to facilitate automated end-to-end testing. Through iterative analysis steps, Pemu dynamically derives the protocol stack used by the firmware by using protocol-specific probes and code coverage as feedback. This probing process is repeated throughout the fuzzing campaign to continuously refine the configuration as new behavior of the firmware's network stack is discovered (see Section 3.4.1).

3.2 Protocol-Aware Firmware Rehosting

We now discuss the main components of Pemu in detail and explain how the encapsulation and extraction steps work.

3.2.1 Packet Encapsulation. Given the current network configuration, the fuzzer can utilize the encapsulation module to encapsulate raw input into realistic network packets. Varying the depth of the encapsulation in a fuzzer-driven manner allows Pemu to equally test every layer of the network stack, unlike most existing general-purpose network application fuzzers. By generating semantically correct packets, the encapsulation module acts as a funnel that enables the fuzzer to target specific firmware components within the network stack dynamically.

The underlying insight is that all network packets follow a layered architecture: each protocol within a packet consists of a header with metadata and a body. Only the body of the highest layer contains the actual application data, while the body of each lower layer encapsulates the layer immediately above it. Header fields may contain static values (e.g., `version = 4` in IPv4), references to another layer (e.g., `EtherType = 8` in Ethernet when the next layer is IPv4), or dynamically computed fields such as checksums. We supply the encapsulation module with a set of protocol grammars to translate these constraints into raw network packets. We use the official specifications to manually derive these grammars (which is a one-time effort). Our grammar can depict complex constraints like fragmentation, compression, checksums, or static values. These protocol grammars allow the encapsulation module to create valid packets that contain fuzzing input as the body of a given encapsulating protocol. To enable broad testing of each protocol layer, we designed the grammar to use fuzzing input for every header field, which is not bound by strict semantic requirements (e.g., addresses and checksums).

The encapsulation module often has to send multiple low-level frames, for instance, due to fragmentation or to perform a handshake before any data reaches the firmware's application layer. Hence, in addition to correctly assembling individual packets, the encapsulation module must maintain state information across a sequence of packets, such as sequence numbers, TCP SYN/SYN-ACK pairs, or fragmentation metadata. This highlights one of the core insights of Pemu: While the firmware might successfully parse a single network packet consisting of only random data by chance, the likelihood of two subsequent randomly generated packets being accepted as a valid application-level input is very low. As a result, current firmware fuzzers are often unable to progress up to the application layer of firmware, as they lack an understanding of the underlying protocol semantics. Furthermore, based on fuzzing input, the encapsulation module can mutate individual header fields. Introducing mutations into the packet allows us to test the boundaries of the network stack. This approach is motivated by the findings of Amusuo et al., who found that 95% of vulnerabilities in embedded network stacks depend on only one or two incorrect header fields [2].

3.2.2 State Extraction. To improve the encapsulation process, we analyze *packets that are sent by* the firmware to derive values related to the network state. While existing rehosting platforms largely ignore the firmware output, Pemu actively uses these output packets to inform the input encapsulation performed by the encapsulation module. There are two reasons for firmware network interactions that we aim to capture protocol state from, *unsolicited* and *solicited* packets. *Unsolicited packets* are packets the firmware network stack sends without any external trigger. The sending occurs either during the firmware's initial setup process or periodically. In the case of a TCP/IP stack, examples are ARP packets to identify the gateway's MAC address, DHCP packets to request an IP address, and ICMP pings to verify connectivity. The firmware includes header metadata of all involved protocols in all cases, e.g., an ARP packet contains the MAC address as well as the sender's IP address. In contrast, *solicited packets* are prompted in response to a packet the firmware received. Examples include the firmware sending a TCP SYN-ACK segment with a sequence number in response to a TCP SYN segment or requesting an IP address in response to a DHCP offer packet.

Whenever the firmware sends a packet, the state extractor module extracts the included protocol state values so they can be added to the configuration of the *virtual network* accordingly. These values are then used when encapsulation fuzzing input, as described in the previous section. This approach eliminates the guesswork that a fuzzer would otherwise face when constructing sequences of packets with valid metadata across multiple protocol layers.

3.3 Platform Independency

Pemu is designed to be independent of the underlying rehosting platform, requiring only a minimal adapter for integration. This adapter enables communication between the platform and Pemu, while the platform itself remains responsible for deciding when to request packets and where to deliver them. This intentional choice allows Pemu to remain agnostic to the specific emulation context and the methods used for input handling. Fundamentally, the platform must provide Pemu only access to fuzzer-generated input (for packet encapsulation). It can then leverage Pemu's straightforward API of two commands: `get_packet` and `send_packet`. To showcase our design's generic applicability, we implement it on top of three different rehosting platforms (see Section 4).

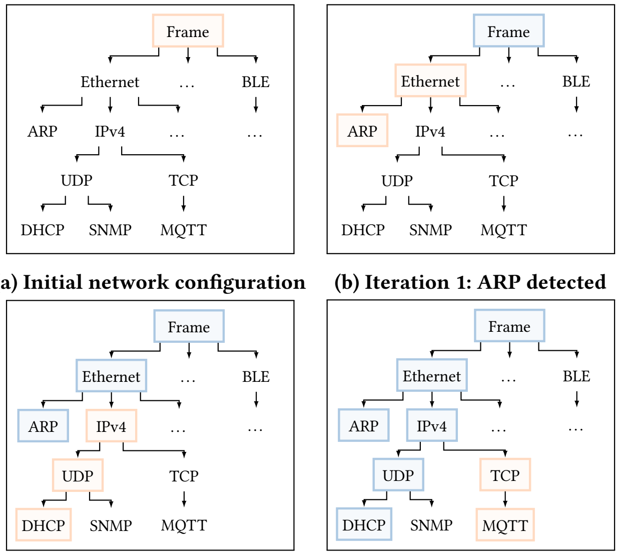
3.4 Pemu-based Network Traffic Analysis

Based on Pemu's encapsulation and state extraction, we design two analysis methods that enable the underlying rehosting platform to perform an end-to-end analysis of a target *without* requiring any manual effort or previous knowledge to extract a network configuration. Furthermore, the gained information can be used to broaden Pemu's knowledge of the firmware's network stack. These passes are designed to be run while the fuzzer is paused to ensure a coherent configuration.

3.4.1 Packet Sequence Extraction. For large-scale analyses, avoiding manually reverse-engineering individual samples to extract information on the type of network packets the firmware accepts is desirable. Manual analysis is not only time-consuming and labor-intensive but also error-prone and difficult to scale across large and diverse firmware samples. Hence, we propose an end-to-end approach that can extract this information without any manual involvement. We introduce a technique called *coverage-based probing* to achieve this goal. The underlying idea is the following: We expect the firmware implementation of a given network layer to react distinctly to well-formed input, i.e., a well-formed input triggers unique coverage when compared to a malformed input. By generating a set of well-formed inputs for each network layer candidate, we can observe which packets elicit a distinct reaction from the firmware. The packets that trigger such distinct reactions likely represent the network protocols used by the firmware under test.

There are several ways to measure these reactions based on firmware coverage. The naive approach of only selecting the packet types that cover the highest number of basic blocks has a significant drawback: The error-handling routine, which is likely triggered by a malformed input, produces coverage. We even found that the number of basic blocks that are part of error handling can be higher than the amount of "regular" coverage induced by a correct packet. To address this issue, we use the metric of *uniquely covered basic blocks* instead of the total number of basic blocks covered. This approach is based on the insight that well-formed packets of the expected type will progress further into the network stack, while unexpected packets will eventually trigger some common error-handling routine. Hence, valid packet types will likely cover the highest number of unique basic blocks.

A challenging aspect of automatically identifying the network configuration is that protocols in the network stack rely on other protocols, even protocols on the same topological layer. More specifically, often, before the firmware can receive a packet containing a given protocol, separate packets of an entirely different protocol may have to be exchanged first. An example of this is ARP and IPv4, where ARP is used to identify the MAC address of a gateway so that the firmware can further communicate with it via IPv4. Similarly, a firmware that implements a web server via TCP may need to



(a) Initial network configuration **(b) Iteration 1: ARP detected**

(c) Iteration 2: DHCP detected **(d) Iteration 3: MQTT detected**

Figure 5: Exemplary space of available network protocols and their nesting, from the point of view of PEMU (🟧 newly detected, 🟦 previously detected). PEMU successively detects more and more protocol layers and adds them to the detected network configuration at runtime.

dynamically configure its IPv4 address via DHCP, which relies on UDP, before it can accept TCP segments.

Building on these insights, we devise an iterative approach for detecting the entire stack of network protocols used by the firmware. For a given state of fuzzing progress, PEMU detects the next network layer currently reachable. This newly identified layer is then added to the existing network configuration, i.e., the set of protocol layers currently assumed to be used by the firmware. This configuration is usually empty at the beginning of the fuzzing campaign. By incrementally updating the network configuration, our approach allows the fuzzing to progress deeper into the network stack.

Figure 5 shows how the configuration is updated over three iterations for an example firmware that exposes an MQTT server. Initially, no protocol layers have been identified (Figure 5a). In this initial configuration, fuzzing input is passed into raw frames. This state is identical to how existing rehosting platforms pass fuzzing input without PEMU's *virtual network*. The example firmware starts resolving the MAC address of its gateway. To this end, the ENS implementation sends ARP requests, expecting an ARP response. In this state, a probe packet containing a valid ARP response triggers the network stack to store the MAC address of its gateway successfully. This event leads to unique code coverage compared to other invalid probe requests (which are discarded by the corresponding layers). Consequently, PEMU adds the Ethernet and ARP protocols to its network configuration, resulting in Iteration 1 (see Figure 5b). Similarly, once the example firmware requests an IPv4 address via DHCP, a probe request that contains a valid DHCP packet will generate unique coverage further up the firmware's network stack. This progression leads to Iteration 2 (Figure 5c). Finally, after the IPv4 address is assigned, the firmware will start listening for incoming TCP connections to serve MQTT requests. This behavior can again be detected by probe requests, leading to Iteration 3 of the configuration (Figure 5d).

An appealing property of this iterative, incremental approach is its *robustness*: Even in the unlikely case that a probe request mistakenly adds an unused protocol layer into the detected network configuration, this protocol is given to the fuzzer as an *option* for encapsulation. As the fuzzer decides up to which layer to encapsulate, it will not be forced to send a wrong type of network packet continuously. In fact, coverage feedback will lead the fuzzer to deprioritize wrong choices.

3.4.2 Parsing-Based Analysis. The concept of parsing-based analysis refers to the comprehensive analysis of the results of the state extraction provided by PEMU. First, all extracted values are collected and analyzed. When multiple options exist for a header field, the most likely one is chosen and integrated into PEMU's configuration for the next fuzzing run. This iterative and self-correcting approach significantly improves the quality of generated network packets, while reducing the required fuzzing input.

4 Implementation

To evaluate our approach, we implemented a prototype of our design. Additionally, we integrated our approach with three state-of-the-art rehosting platforms: *FUZZWARE* [39], *HOEDUR* [41], and *SEMU* [54]. We release PEMU as well as our patches to these platforms at https://github.com/MPI-SysSec/pemu.

4.1 Implementation Aspects of PEMU

PEMU's *virtual network* exposes two core interfaces: one for receiving packets from the network and one for sending packets to the network. We implemented PEMU in Python, which is supported by most rehosting platforms. The implementation consists of about 3,600 lines of Python code.

Encapsulation Module. The *virtual network* provides a dedicated module, the encapsulation module, which monitors the current network state and assembles packets as needed. The encapsulation module is initialized with a predefined list of network packets (i.e., the network configuration). Upon each request from the rehosting platform, it processes the list and assembles the packet at the current index before advancing to the next. Protocol semantics and syntax are defined in YAML configuration files and are derived from official specifications. Header fields are classified into the following categories:

- Static values that are specified directly in the configuration.
- Length values and the scope they refer to.
- Fields that reference higher layers
- Fields that link handler functions for complex calculations, like checksums.
- State-related values that require updates, such as flags and sequence numbers.
- Remaining fields that can be populated with fuzzing data.

Like an actual network stack, the encapsulation module assembles packets layer by layer, starting from the highest to the lowest protocol layer. Additionally, the *virtual network* supports fuzzer-guided fault injection, allowing it to introduce mutations into packets to test edge cases and error-handling logic. These malformed packets can help uncover vulnerabilities in lower-layer parsers that might be missed with well-formed inputs alone. Alongside our tool, we publish YAML configuration files for 38 protocols.

State Extraction. The State Extraction module parses outgoing packets according to their protocol specifications. The module identifies and stores unknown protocol values (e.g., IP addresses, IDs, and nonces) for later analysis. Complex header fields, such as the Frame Control Field (FCF) in IEEE 802.15.4 or cases involving fragmentation and compression, can be managed with custom handler functions provided by the user. Adding handler functions is a one-time effort when adding a new protocol.

4.2 Pemu-based Analysis

We also implemented two complementary analysis techniques that enable end-to-end automation without manual intervention. These analyses are performed after fuzzing is paused, and their results are then integrated into the network configuration upon continuation. This is done to prevent inconsistent configurations.

Analyzing Extracted State. During fuzzing, the *virtual network*'s state extractor module inspects and extracts relevant information from packets transmitted by the firmware. This data is accumulated and analyzed to refine and expand the network configuration with new information on connection-specific values and new network packet types, enhancing future fuzzing effectiveness.

Probing. Relying solely on state extraction to explore the network state has limitations, particularly when there is no prior knowledge of the protocol suite or when the firmware, based on its role (e.g., client or server), remains idle without an external prompt. We use an active probing approach to address this issue, as detailed in Section 3.4.1. This method leverages heuristics to select probes, or packet types, that are most likely to advance exploration of the firmware's network stack. To this end, the encapsulation module supports a probe mode, where it generates packets with randomized yet syntactically valid configurations based on the current firmware state. All fuzzing bytes in the probe packets are set to zero to ensure consistency across probing runs and avoid non-deterministic behavior from random inputs. Only non-fuzzed fields—such as those derived from static values, handlers, checksums, lengths, and pointers—remain unaltered, reducing the risk of random bytes misleadingly mimicking unintended protocol fields. This ensures, for example, that a randomized Ethernet MAC address is not erroneously interpreted as an IEEE 802.15.4 frame.

Subsequently, the probing component places the encapsulation module in the dedicated probe mode, extracts the gathered coverage, and applies the heuristics to the collected basic block sets. After running the probes, coverage data is collected, and heuristics are applied to assess the firmware's basic block responses. The probing module includes a communication interface adaptable to different rehosting platforms, ensuring precise extraction and interpretation of coverage data.

Table 1: Categories of binary-only rehosting approaches

Category	Platforms
Single-Stream	**Fuzzware** [39], AIM [12], P2IM [13], DICE [28]
Multi-Stream	**Hoedur** [41], Multifuzz [9]
Spec-based	**SEmu** [54]

4.3 Integration with Existing Fuzzing Platforms

Pemu is designed to be platform-agnostic, leaving its integration, configuration, and use to the fuzzing platform. First, we outline two general methods for delivering network packets to network buffers within rehosting platforms. Afterward, we describe our implementation of Pemu with three representative platforms—Fuzzware, Hoedur, and SEmu—each illustrating a distinct approach to binary-only rehosting and firmware fuzzing, as categorized in Table 1.

Integration Concepts. Rehosting platforms must support network packet handling to enable Pemu to deliver packets to the firmware. While some firmware applications receive packets via serial interfaces like UART [37], most rely on DMA-enabled peripherals to transfer packets directly into a RAM buffer. Although recent techniques can model simple DMA transfers [28], many network peripherals use complex DMA mechanisms that current methods do not support. Designing a generally applicable solution to this problem is inherently orthogonal to our approach; therefore, to handle it, we propose two integration approaches:

1. *Hook-Based Integration*: Many platforms allow debugging hooks at specific basic blocks. The emulator can inject packets into the correct buffer by placing a hook at the block where packet data is first accessed. During the execution of each hook, DMA control structures are checked to determine packet insertion points. These hooks are implemented once per MCU family. Identifying the hook placement is an effort that needs to be taken once per HAL. For example, every STM32 MCU built with STM32's HAL accesses the received data within the same function: `HAL_ETH_ReadData`.
2. *Peripheral Emulation*: A more robust approach is the manual implementation of network peripherals based on MCU specifications, where the peripheral handles all MMIO interactions. This enables high-fidelity emulation and removes the need for per-sample analysis, as the emulated peripheral autonomously identifies when to insert new packets. This approach's feasibility depends on the platform's architecture.

In the following, we describe how we integrated our approach with three different fuzzing frameworks.

Fuzzware. Fuzzware [39] uses symbolic execution to model MMIO accesses, optimizing fuzzing input usage. Based on Unicorn [47], Fuzzware supports both integration methods. We manually implemented network peripherals for four MCUs, with each implementation averaging 435 lines of Python code.

Hoedur. Hoedur [41] is a multi-stream fuzzer that improves the MMIO modeling introduced by Fuzzware by introducing distinct input streams per MMIO channel for targeted mutation. Built with Rust on QEMU [5], integration with Pemu required cross-language support. We implemented hooks for packet reception and transmission for four MCUs, which detect network buffer addresses and inject packets as needed while HOEDUR manages other MMIO functions. On average, each MCU implementation requires 184 lines of Rust code. In total, the integration required 823 lines of code.

SEmu. SEMU [54] diverges from the other two platforms by using natural language processing to extract condition-action rules from MCU specifications, guiding MMIO emulation with high fidelity. Although primarily MMIO-focused, SEMU includes an Ethernet peripheral for one supported MCU. We integrated PEMU using this peripheral, requiring only 34 lines of Python code.

5 Evaluation

To evaluate the effectiveness of PEMU, we conducted several experiments to determine whether and to what extent *virtual network*s can improve embedded firmware fuzzing. To this end, we implemented and tested PEMU across the three different rehosting platforms. Furthermore, we evaluate our approach against EMNETTEST [2], a testing tool for embedded network stacks. We address the following research questions in our evaluation:

- **RQ1**. Is PEMU applicable to multiple fuzzing platforms?
- **RQ2**. What are the benefits of network protocol-aware fuzzing, and how can PEMU improve previous methods?
- **RQ3** How does PEMU compare to other ENS testing approaches?
- **RQ4**. Can PEMU improve the detection of bugs in embedded network stacks?

5.1 Setup

To account for the inherent randomness of the fuzzing process, we run each fuzzing campaign five times and follow the best practices for evaluating fuzzers by Klees et al. [23] and Schloegel et al. [42]. We plotted the median and the 95% confidence interval for the results to account for any uncertainty.

Hardware Configuration. All our experiments use the same hardware configuration: two AMD EPYC 9654 CPUs running at 3.707 Ghz (192 physical cores in total), 768 GB of RAM, and SSD memory for storage.

Platforms and Fuzzers. We performed experiments across the three different rehosting platforms FUZZWARE, HOEDUR, and SEMU. For each platform, we designed an ablation study, consisting of two to three configurations, to measure the impact of PEMU as precisely as possible:

- Baseline (_BASE): Each platform's unmodified version serves as a baseline to analyze how much coverage can be attributed to the network stack
- Random packet (_RAND): Baseline extended by the ability to send network packets consisting only of random data
- PEMU (_PEMU): Baseline with the ability to send network packets emitted by PEMU.

As SEMU has the capability to send random network packets by default, we use the two configurations SEMU_BASE and SEMU_PEMU.

In addition to evaluating how PEMU enhances existing rehosting platforms, we assess its effectiveness against other network stack fuzzing tools, namely EMNETTEST. While AFLNET may appear to be a relevant baseline, fundamental differences limit its applicability in the rehosting context. Applying AFLNET to firmware would require manual source modifications (e.g., to remove checksums) and a hardware-in-the-loop setup for trace collection. These requirements conflict with the fundamental principles of rehosting. Moreover, AFLNET focuses solely on the application layer and uses response codes as feedback, offering only limited insights into other protocol-layer interactions. As a result, AFLNET is not directly comparable to PEMU's broader, ENS-agnostic fuzzing capabilities.

5.2 Sample Set

In firmware fuzzing research, standard sample sets from prior studies [10, 13, 52] are used to enable a fair and consistent comparison across different works. However, existing firmware sample sets rarely include network applications, as network stacks have generally been outside the scope of past research. To address this shortcoming, we assembled a sample set of nine applications, listed in Table 2. This set includes six novel applications along with two samples from SEMU [54] and one sample from FUZZWARE [39], featuring diverse firmware across four different MCUs from three vendors, spanning six OSes and five network stacks. These network stacks cover three of the most widely used protocol suites:

(1) Ethernet and TCP/IP are the foundational standards in general-purpose computing, facilitating communication across the internet. The protocol suite includes multiple protocols essential for networked communication. The samples that contain a TCP/IP stack are 1) a *HTTP Server* with a static IPv4 address, 2) a *UDP Server* that uses DHCP to configure its IPv4 address, 3) a *TCP Echo Server*, and 4) a *TCP Echo Client*. Note that the last two images were taken from the SEMU data set.
(2) Bluetooth Low Energy (BLE) is a low-power Bluetooth standard optimized for close proximity communication. It operates on distinct physical channels: The advertisement channel, which is used for advertising, scanning, and connection establishment, and the data channel, which is used for actual data transmission. For the BLE stack, the applications are 1) a *BLE Heart Rate Monitor*, which is built on zephyr-os, and 2) a *nimBLE* application by the nuttx operating system.
(3) IEEE 802.15.4 is a radio-based protocol suite for establishing WPAN networks. It operates on the two lowest layers and is typically used with other high-level standards like 6LoWPAN or Zigbee. The samples implementing this stack are 1) the contiki-ng *Radio Ping*, which is an ICMPv6 ping application, 2) a *Constrained Application Protocol (CoAP) Client* that tries to interact with a corresponding server, and 3) an *SNMP Server* that features a known CVE and is part of the FUZZWARE sample set.

For FUZZWARE and HOEDUR, we configured firmware samples using the built-in tool to generate appropriate configurations, using provided templates wherever possible and documenting minimal manual adjustments in the configurations. For instance, certain nRF52840 DK samples required specific MCU values in user or factory information configuration registers located in RAM. This additional memory area was manually configured for samples using this feature. All samples are published with our prototype.

Table 2: Tested evaluation samples.

Vendor	MCU	Protocol Suite	ENS	OS	Sample	Source
STM32	nucleo-f767zi	Ethernet	LwIP	FreeRTOS	HTTP Server	[44]
STM32	nucleo-f767zi	Ethernet	NetXDuo	ThreadX	UDP Server	[44]
STM32	f429	Ethernet	LwIP	Raw	TCP Echo Server	[53]
STM32	f429	Ethernet	LwIP	Raw	TCP Echo Client	[53]
Nordic	nrf52840dk	BLE	nordic softdevice	zephyr	BLE Heart Rate Monitor	[33]
Nordic	nrf52840dk	BLE	nimBLE & nordic softdevice	nuttx	nuttx nimBLE	[33]
Nordic	nrf52840dk	IEEE 802.15.4	uIP	contiki-ng	Radio Ping	[11]
Nordic	nrf52840dk	IEEE 802.15.4	uIP	contiki-ng	CoAP Client	[11]
Texas Instruments	cc2538	IEEE 802.15.4	uIP	contiki-ng	SNMP Server	[11]

Table 3: Setup of the EmNetTest data set.

ENS	CVE-ID	Type	EOS	MCU
FreeRTOS-plus-TCP	2018-16523	Div-by-zero	FreeRTOS	MPS2
	2018-16524	OOB Read		
	2018-16526	OOB Write		
	2018-16601	Integer Underflow		
	2018-16603	OOB Read		
Contiki-ng	2021-21281	OOB Read	Contiki-ng	TI cc2538dk
	2022-26053	OOB Write		
PicoTCP	2020-17441	OOB Read	FreeRTOS	STM32 F769
	2020-17442	Integer Overflow		
	2020-17444	Integer Overflow		
	2020-17445	OOB Read		
	2020-24337	Infinite Loop		

To enable the comparison with EmNetTest, we use their dataset, which consists of twelve CVEs across three ENSs. As EmNetTest tests standalone ENSs compiled for the host system rather than full firmware images, a direct coverage-based comparison is not feasible in our rehosting setup. We used the following setup to replicate their evaluation in our rehosting environment. First, we selected three different MCUs to compile the network stacks to and integrated each ENS with an embedded OS. We then backported the vulnerabilities used by EmNetTest (see Table 3).

5.3 Experiments

We perform four experiments to assess Pemu's ability to improve the coverage of existing approaches and to test whether Pemu allows a fuzzer to detect bugs nested deep within the network stack.

Experiment 1: SEmu. First, we performed an ablation study on SEmu. As SEmu only supports a limited chipset, we selected the samples of their evaluation dataset that exhibit network behavior and performed an ablation study for these samples. We fuzzed each sample five times for 24 hours with the two versions of SEmu: SEmu_base and SEmu_Pemu. Afterward, we collected the basic blocks each run was able to cover and evaluated them.

Experiments 2 and 3: Fuzzware & Hoedur. Next, we performed two similar ablation studies for Fuzzware and Hoedur. We fuzzed each sample of the full sample set five times for 24 hours with the three different versions (e.g., Fuzzware_base, Fuzzware_rand, and Fuzzware_Pemu; the same applies to Hoedur). We then again collected the basic blocks that each run was able to cover and evaluated them.

Experiment 4: State-of-the-art comparison. To compare Pemu with EmNetTest, we ran Hoedur_Pemu for 72 hours with samples containing the backported vulnerabilities and then analyzed the results. We chose Hoedur as the base rehosting platform because its usage of multiple input streams allows for the most stable emulation [41].

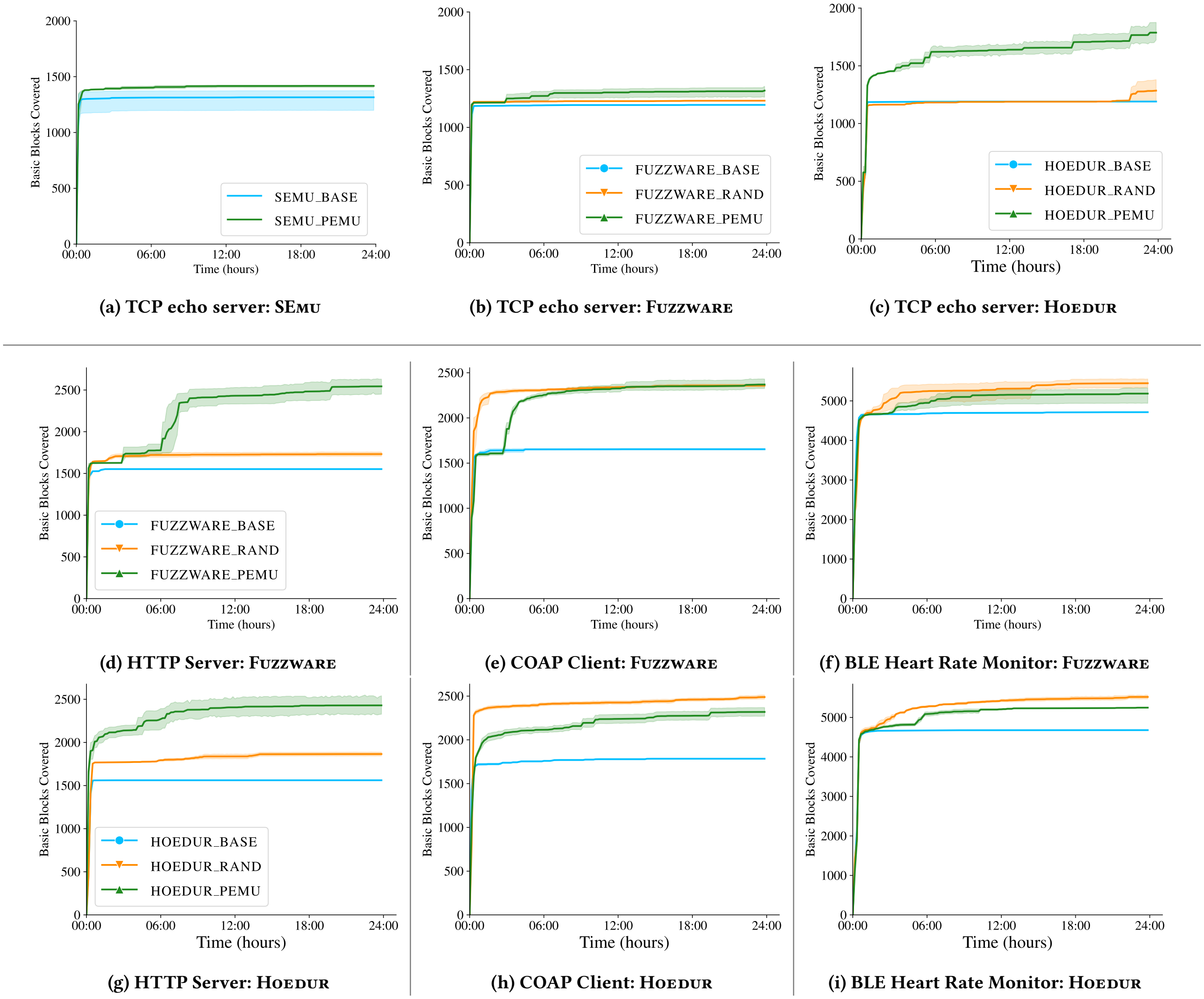
Experiment 5: Applicability beyond network stacks. To evaluate whether Pemu's techniques generalize beyond network-based communication protocols, we extended it with support for the Modbus protocol [46], which is widely used in industrial applications. A Modbus frame consists of a device ID, a function code, a length field, a variable-length payload, and a CRC checksum. To this end, we fuzzed the heat-press firmware from the P2IM dataset [13], which utilizes the Modbus protocol. Each configuration with Fuzzware_rand and Fuzzware_Pemu was run five times for 24 hours. Before starting the fuzzing campaign, we reverted previous modifications introduced by P2IM that had disabled the Modbus checksum verification to ensure realistic conditions.

5.4 Results

In the following, we summarize the experimental results in relation to the research questions. In addition to the selected plots in Figure 6, further plots can be found in the appendix. A more detailed breakdown of the results can be found in Table 4.

5.4.1 RQ1. To address the first research question on the applicability of Pemu across multiple rehosting platforms, we integrated it with three distinct platforms: Hoedur, Fuzzware, and SEmu (see Section 4). This demonstrates Pemu's compatibility with platforms that support (custom) network peripherals or basic block hooks and affirms the design choice of platform independence. As shown in Figure 6, Pemu visibly improved the coverage on all platforms despite their divergent approaches, demonstrating its versatility and applicability in embedded firmware fuzzing.

5.4.2 RQ2 – SEmu. The results in Figure 6a (and Figure **??** in the appendix) display the coverage achieved by SEmu_Pemu (i.e., SEmu with Pemu) and the coverage by SEmu_base (i.e., SEmu without any



(a) TCP echo server: SEmu

(b) TCP echo server: Fuzzware

(c) TCP echo server: Hoedur

(d) HTTP Server: Fuzzware

(e) COAP Client: Fuzzware

(f) BLE Heart Rate Monitor: Fuzzware

(g) HTTP Server: Hoedur

(h) COAP Client: Hoedur

(i) BLE Heart Rate Monitor: Hoedur

Figure 6: Selected coverage plots from the different ablation studies.

modifications). The coverage achieved by SEmu_Pemu outperforms the baseline SEmu_base by 8.5% on average. Due to SEmu's internal architecture, it cannot send more than one packet per emulation run. Still, this improvement highlights the benefits of network-aware fuzzing.

5.4.3 RQ2 – Fuzzware & Hoedur. Next, we discuss the results for Fuzzware and Hoedur—which both support the entire sample set—together due to the similarity of their results. On average, Pemu is able to improve the coverage of the Fuzzware baseline by 40.7%, while it improves the baseline of Hoedur by 39.2%. To highlight different aspects, we further analyze the results by protocol stack:

TCP/IP. For the four samples that implement a TCP/IP network stack (i.e., *HTTP Server*, *UDP Server*, *TCP Echo Server*, and *TCP Echo Client*), the results in Table 4 show a consistent pattern: For both Fuzzware and Hoedur, the ablations with Pemu clearly outperform their respective baseline by achieving 32.5% more average coverage for Fuzzware and 48.2% for Hoedur. When looking at the combined number of basic blocks covered over the five fuzzing runs, Pemu even improves basic-block coverage by 39% and 61.3%, respectively. Notably, the second ablation (Fuzzware_rand and Hoedur_rand) surpasses the baseline in both cases but is consistently outperformed by the configurations using Pemu. These results confirm the hypothesis that random inputs progress significantly worse in network stacks that enforce strict semantic constraints on their input. In a TCP/IP stack, these constraints are already present on the IPv4 layer (i.e., the second lowest layer), including checks for IPv4 addresses, a static protocol value, and a header checksum. While some implementations do not enforce the checksum, allowing randomized data to progress marginally

Table 4: Breakdown of the basic block coverage of each sample. base refers to the baseline (SEmu_base, Fuzzware_base, and Hoedur_base), rand to the ablation configuration Fuzzware_rand and Hoedur_rand, and Pemu to the full configuration (SEmu_Pemu, Fuzzware_Pemu, and Hoedur_Pemu). The column *rel imp* measures the relative improvement of Pemu over the baseline. We highlighted the results that are part of the summarized plots in Figure 6.

	Target	#BB in target	#BB AVG				#BB MAX				#BB combined			
			base	**rand**	**Pemu**	***rel imp***	**base**	**rand**	**Pemu**	***rel imp***	**base**	**rand**	**Pemu**	***rel imp***
SEmu	TCP Echo Server	5,212	1,315	–	**1,417**	*8%*	1,375	–	**1,425**	*4%*	1,375	–	**1,425**	*4%*
	TCP Echo Client	5,333	1,342	–	**1,467**	*9%*	1,406	–	**1,475**	*5%*	1,406	–	**1,475**	*5%*
Fuzzware	TCP Echo Server	5,212	1,195	1,231	**1,318**	*10%*	1,196	1,234	**1,374**	*15%*	1,196	1,234	**1,378**	*15%*
	TCP Echo Client	5,333	1,227	1,302	**1,661**	*35%*	1,228	1,317	**1,732**	*41%*	1,228	1,327	**1,764**	*44%*
	UDP Server	5,451	1,928	2,160	**2,336**	*21%*	1,938	**2,394**	2,375	*23%*	1,938	**2,396**	2,390	*23%*
	HTTP Server	6,579	1,552	1,732	**2,543**	*64%*	1,552	1,770	**2,683**	*73%*	1,552	1,774	**2,705**	*74%*
	BLE Heart Rate Monitor	17,469	4,715	**5,446**	5,185	*10%*	4,717	**5,608**	5,358	*14%*	4,723	**5,631**	5,380	*14%*
	nuttx nimBLE	20,260	6,580	6,588	**6,620**	*1%*	6,592	6,596	**6,710**	*2%*	6,602	6,596	**6,711**	*2%*
	CoAP Client	5,666	1,653	2,359	**2,369**	*43%*	1,671	2,382	**2,492**	*49%*	1,671	2,400	**2,542**	*52%*
	SNMP Server	4,578	1,445	2,253	**2,267**	*57%*	1,450	2,321	**2,347**	*62%*	1,450	2,354	**2,370**	*63%*
	Radio Ping	4,778	923	**2,109**	2,079	*125%*	929	**2,145**	2,108	*127%*	929	**2,185**	2,151	*132%*
Hoedur	TCP Echo Server	5,212	1,190	1,285	**1,787**	*50%*	1,198	1,663	**1,915**	*60%*	1,200	1,665	**1,947**	*62%*
	TCP Echo Client	5,333	1,218	1,242	**1,828**	*50%*	1,222	1,296	**1,957**	*60%*	1,227	1,299	**2,049**	*67%*
	UDP Server	5,451	1,945	2,390	**2,655**	*37%*	1,946	2,431	**2,667**	*37%*	1,946	2,460	**2,672**	*37%*
	HTTP Server	6,579	1,561	1,865	**2,429**	*56%*	1,562	1,915	**2,800**	*79%*	1,562	1,931	**2,802**	*79%*
	BLE Heart Rate Monitor	1,7469	4,677	**5,517**	5,246	*12%*	4,689	**5,635**	5,255	*12%*	4,689	**5,661**	5,319	*13%*
	nuttx nimBLE	2,0260	6,619	**6,627**	6,619	*0%*	6,637	**6,644**	6,642	*0%*	6,639	**6,646**	6,643	*0%*
	CoAP Client	5,666	1,784	**2,492**	2,319	*30%*	1,786	**2,572**	2,403	*35%*	1,786	**2,612**	2,435	*36%*
	SNMP Server	4,578	1,440	**2,519**	2,328	*62%*	1,440	**2,633**	2,368	*64%*	1,440	**2,758**	2,402	*67%*
	Radio Ping	4,778	1,332	**2,203**	2,083	*56%*	1,332	**2,277**	2,115	*59%*	1,332	**2,349**	2,149	*61%*

further, most packets fail at the IPv4 address validation. In contrast, Pemu either leverages DHCP packets to dynamically assign the firmware an IPv4 address or extracts the firmware's IPv4 address by parsing outgoing packets. Consequently, Pemu is able to progress considerably further through the network stack.

IEEE 802.15.4. The results for the IEEE 802.15.4/6LoWPAN-based samples (i.e., *CoAP Client*, *Radio Ping*, and *SNMP Server*) differ from those of the TCP/IP samples. Both Fuzzware_rand and Fuzzware_Pemu cover up to 100% more basic blocks than the baseline Fuzzware_base. However, the difference in coverage between Fuzzware_rand and Fuzzware_Pemu is much smaller and, in some cases, yields almost identical results. This trend is similar for Hoedur; due to the overall better coverage of the Hoedur baseline compared to Fuzzware, the relative improvement is slightly smaller—on average 53%. Again, the improvement gains on the second ablation Hoedur_rand are less significant or slightly negative.

Generally speaking, these results reflect the differences in the lower layers of the TCP/IP and the IEEE 802.15.4 network stacks. In TCP/IP, the IPv4 layer (i.e., the second lowest layer) already acts as a stringent barrier, discarding all semantically and syntactically invalid packets. Conversely, in the IEEE 802.15.4 layer, stringent checks like checksum validation are deferred to the transport layer (i.e., the layer above IPv6). Header compression further simplifies the validation of packets as valid, even without explicit addresses. Consequently, random input can more easily penetrate the MAC layer, 6LoWPAN adaption layer, and IPv6 network layer. The complexity of these layers, due to compression and fragmentation and a multitude of option headers, provides a more expansive code path for coverage than TCP/IP. As Pemu prioritizes methodical exploration of header fields, it requires additional time to achieve similar coverage. However, manual inspection of the results revealed that, for example, Fuzzware_rand was unable to induce coverage in the application layer (i.e., SNMP and CoAP). At the same time, Fuzzware_Pemu consistently reached the respective handlers.

For Hoedur, the difference between Hoedur_rand and Hoedur_Pemu is exacerbated by performance reasons owed to the cross-language setup, significantly slowing the execution speed. Compared to unmodified Hoedur the performance overhead ranges from a factor of 2 to 10. This overhead can vary depending on how many packets Pemu crafts per execution. As input lengths grow over time, execution duration increases accordingly, leading to a higher per-execution overhead of Pemu. However, to utilize the multi-stream aspect provided by Hoedur, our implementation is forced to request input from Hoedur in small chunks. As the fuzzing input can only be supplied within the Rust context of Hoedur, this necessitates crossing the language border multiple times for a single network packet. We discuss this implementation aspect further in Section 6.

Bluetooth Low Energy (BLE). The two samples that implement a BLE network stack (i.e., *nimBLE example* and *BLE Heart Rate Monitor*) yield similar results to those of the IEEE 802.15.4 stack. Unlike the previous samples, both BLE samples include a proprietary component by the MCU vendor, which contains the complex logic of the BLE stack. This component explains the overall higher base coverage of both Fuzzware and Hoedur in Table 4. For the *BLE Heart Rate Monitor*, both Fuzzware_rand and Fuzzware_Pemu achieve up to 15% more coverage than Fuzzware's baseline. For Hoedur, this improvement is very similar, averaging 12%. This—compared to the other protocol suites—modest improvement reflects the complexity of the sample and the BLE protocol suite in general. Decreased emulation stability often limits the amount of packets delivered to the firmware to a single packet. This restricts network-related coverage primarily to the BLE advertising channel (i.e., advertising packets, scan packets, and connection packets) without establishing a connection that would enable access to the application logic of BLE.

The *nuttx nimBLE* sample also poses a similar problem regarding its complexity: contrary to the previous sample, however, the lowest BLE layer only accepts frame types that contain the firmware's correct BLE address. Hence, packets with the wrong address are discarded immediately. For Fuzzware, the emulation was only in one of the five runs stable enough for Pemu to successfully parse a packet and extract the firmware's BLE address. This enabled Fuzzware_Pemu to outperform the other Fuzzware_base and Fuzzware_rand regarding total and average coverage (see Table 4). However, for Hoedur, this success was not replicated, resulting in nearly identical results across all three ablations.

Evaluation of Analysis Timing. To evaluate the performance of Pemu's analysis component, we measure the detection timing of new packet types (i.e., protocol detection) and values (e.g., addresses and nonces), using Fuzzware_Pemu as an example. In this setup, we introduce an initial three-hour delay before enabling Pemu to ensure that the fuzzer reaches the point where packets are actually parsed by the firmware, which is an essential prerequisite for any analysis. Across all runs, Pemu consistently classified each firmware's network stack correctly, enabling valid packet generation in all cases. For most targets, the plots in Figure 9 visibly exhibit an increase in coverage around the three-hour mark, corresponding to the startup of Pemu. Note that all detection times reported below are relative to this initial three-hour delay.

The median time to detect the first protocol is 47 minutes, while the median time for the *last* protocol detection is 12 hours and 33 minutes. These results show that Pemu can detect protocols reasonably fast and can continuously expand the sequence of packets it sends. The delay between protocol detections is a result of our design: after each new configuration is found, the fuzzer is given a configurable window of uninterrupted fuzzing, potentially uncovering new firmware states that will feed back information into Pemu 's analysis. Taking the UDP echo firmware as an example, Pemu detected DHCP in its first analysis iteration and added two DHCP packets to its configuration. After further fuzzing, Pemu's analysis added a UDP packet to the configuration. Next, Pemu detected that the firmware also parses ARP packets and expands the configuration. At this point, Fuzzware_Pemu is able to fuzz the UDP echo server's logic. During subsequent analysis runs, Pemu also added an IGMP and an ICMP4 packet to the configuration, as the firmware also implements handling for these protocols. This step-by-step expansion illustrates the versatility of Pemu and shows the importance of our dynamic, feedback-driven analysis approach.

The median time for detecting the first value is 1 hour and 35 minutes. Over the course of a fuzzing campaign, Pemu detects new protocol configurations an average of 3.75 times, while new values are detected 1.4 times on average. These values are typically either (static) addresses or identifiers like the transaction ID field in DHCP.

In the following, we provide a detailed analysis of the coverage across the individual network layers, focusing specifically on the network and transport layers. We exclude the two BLE samples, as the BLE protocol stack does not comply with the standard OSI model [16], making a comparison difficult. On average, Fuzzware_Pemu reaches the target's *network layer* 25 minutes after the three-hour delay, or 14 minutes after detecting the first protocol configuration. Overall, Fuzzware_Pemu successfully covers the network layer of all seven non-BLE samples. For the *transport layer*, the average time to first coverage is 3 hours and 5 minutes after discovering the initial protocol configuration. Reaching the transport layer usually coincides with a successful detection of the target's IP address, unless the target uses a dynamic IP address. Then, Pemu assigns this address by sending the appropriate DHCP packets. Fuzzware_Pemu reaches the transport layer for all cases where the sample has enabled at least one transport layer protocol. The only exception is the Radio Ping sample, which does not implement any functionality above the network layer. Comparing this to our baseline, Fuzzware_rand, we find it can reach the transport layer in only a single case, and only after requiring an average of 10 hours and 21 minutes.

5.4.4 *RQ3 Comparison with EmNetTest.*

We ran Hoedur_Pemu for 72 hours on the EmNetTest dataset and analyzed the results. Our evaluation showed that Pemu successfully detected *all* twelve known bugs in the data set (see Table 3), demonstrating both its correctness and effectiveness. All detected bugs were located on the network or transport layer, several requiring multiple subsequent packets to trigger the bug, which highlights Pemu's ability to handle protocol state and sequencing. Additionally, Pemu discovered three previously unknown bugs in the FreeRTOS-plus-TCP ENS, while EmNetTest did not detect any new issues in this stack. Furthermore, even though the fuzzer ran for 72 hours, all bugs were rediscovered within the first 24 hours.

5.4.5 *RQ4 Bug Finding Ability of Pemu.*

In addition to the twelve real-world bugs rediscovered throughout the EmNetTest experiments, we identified five previously unknown bugs during the evaluation of Pemu. Three of these bugs were found in the FreeRTOS-plus-TCP ENS. We are currently in the process of investigating and disclosing them to the vendor. The fourth bug is an out-of-bounds (OOB) write in LwIP. Pemu discovered it in one of the samples introduced in the SEmu dataset. In contrast, SEmu was unable to detect this bug that allows an attacker to control the target's PC by overwriting data structures behind the network buffer. Further analysis showed that the bug resulted from a misconfiguration during the creation of the sample by the SEmu authors. We discovered the fifth bug in the HAL of the STM32F767 MCU. It is an OOB read that arises due to improperly handled thread synchronization. The target needs to receive a valid TCP segment, which leads to a response from the firmware, to trigger the bug. Simultaneously, the firmware

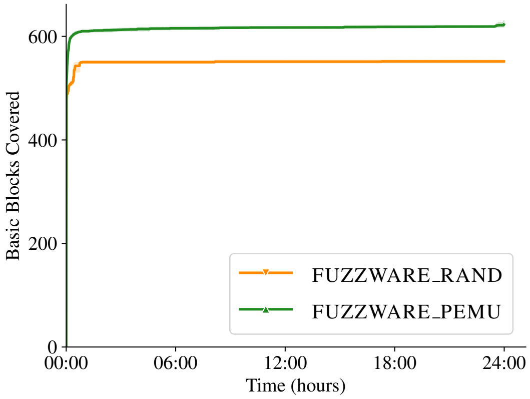


Figure 7: Coverage plot for the heat-press firmware, which utilizes the Modbus protocol.

automatically sends periodic ARP broadcasts to the network. If the transmission of the response and the periodic broadcast overlap, it can trigger a race condition, which leads to an OOB read that can leak data to an attacker. We disclosed this to the vendor, who acknowledged it and published a dedicated security advisory. This bug emphasizes the importance of holistic testing of network stacks in embedded firmware: although the bug does not reside in the network stack, triggering it requires the reception of a valid packet. Testing only the standalone network stack—like EmNetTest does—cannot reveal such bugs, highlighting the advantages of approaches that enable an end-to-end analysis.

5.4.6 Experiment 5: Applicability of Pemu beyond Network Stacks. To further assess Pemu's applicability beyond network stacks, we fuzzed the heat-press firmware [13], which communicates via the Modbus protocol. Figure 7 shows the coverage achieved by both ablations (Fuzzware_Pemu and Fuzzware_rand). As valid packets need to have a correct CRC checksum, Fuzzware_Pemu outperforms Fuzzware_rand by a significant margin. Manual analysis confirms that packets from Fuzzware_rand could pass the checksum verification occasionally; however, this happens too infrequently to produce meaningful new coverage. Statistically, the probability of a randomly generated packet having a correct CRC-16 checksum is very low. This case study demonstrates that Pemu is able to support structured communication protocols outside the network domain.

6 Discussion and Limitations

In the following, we discuss potential limitations of our approach and our prototype and explore directions for future research.

Low-Level Packet Transmission and DMA. Pemu is designed to extend any firmware rehosting platform that integrates a fuzzer for firmware testing. We assume the mechanism for receiving and transmitting low-level network frames to be known, i.e., we require a mechanism to control the contents of the low-level frames that the firmware receives (by writing them to the appropriate firmware memory) and a mechanism to read the frames that the firmware transmits (by reading them from firmware memory). The main mechanism to transmit data, such as Ethernet frames or over-the-air radio frames, is DMA. If the rehosting platform handles DMA, the transfer mechanism requirement will already be fulfilled. Fully automated DMA transfer modeling is an active research area orthogonal to our work. A first step in this direction is DICE [28], which enables detection and rehosting of DMA streams by analyzing MMIO-based configuration patterns. The approach is geared towards simple DMA interactions typically performed by peripherals like UART, SPI, or ADC. However, complex MCUs and peripherals (like network controllers) often perform more sophisticated DMA schemes that rely on RAM-based control structures (e.g., linked lists or array lists). DICE's modeling approach cannot capture these structures. Recent work by Scharnowski et al. [40] improves upon DICE, by supporting more soph isticated transfer mechanisms. This offers the possibility to attach Pemu in an entirely generic manner.

To stay independent of the progress of automated DMA modeling research, Pemu implements alternative options to integrate with the frame transmission of the rehosting platform. One option is an available implementation of a network peripheral that facilitates the reception and transmission of low-level frames to and from firmware memory. A second option is hooking into the hardware abstraction layer (HAL) functions. Here, Pemu provides hook implementations that can be configured in the rehosting environment to be invoked when a given packet reception or transmission function is called in firmware code.

Performance Impact in Hoedur. As discussed in Section 5.4.3, the cross-language interaction between Python and Rust, prevalent in Hoedur, introduces a measurable performance impact. The decision to implement Pemu in Python was mainly influenced by the current ecosystem of rehosting platforms: most rehosting platforms are either written in Python or make heavy use of it [10, 13, 28, 39, 52, 54].

Only recently, firmware fuzzers have started adopting Rust [9, 41]. Creating two separate implementations in Python and Rust would have been impractical, as it could have introduced inconsistencies and increased maintenance complexity. Additionally, it would have impacted comparisons across platforms.

Scaling with General Rehosting Progress. Like previous work that builds upon rehosting platforms, Pemu relies on the ability of the underlying rehosting platform to bring the firmware into a state in which it receives packets. As described in Section 3.4.1, we account for this via an iterative probing approach. However, if the rehosting platform hits a roadblock unrelated to networking that prevents it from ever discovering the first layer of network functionality, the *virtual network* of Pemu remains unused. Similarly, while Pemu provides a transparent input encapsulation mechanism, it still relies on the fuzzer to provide raw input for the application layer that is ultimately tested. As such, Pemu's benefit scales with further improvements to general rehosting platforms.

Manual Effort. The manual effort associated with using Pemu can stem from two sources. The first source is the integration of new protocols with Pemu. This is a one-time effort that largely depends on the protocol's complexity: structurally simple protocols like ARP, Modbus, or Ethernet can typically be added within minutes, whereas complex protocols that use compression or fragmentation may require several hours of effort. Notably, Pemu does not require exhaustive manual modeling of all protocol fields. Instead, only semantically critical fields (i.e., those essential for correct parsing) must be handled explicitly. All remaining fields can be controlled by the fuzzer. The second source of manual effort arises when configuring Pemu for use with a new firmware. In general, Pemu does not need any specific configuration, as it autonomously discovers new protocols and firmware-specific values like addresses and nonces. Instead, the main overhead arises when configuring the input channel through which the firmware receives its raw network frames. As discussed earlier, this effort is negligible if DMA modeling is available. Otherwise, it requires a one-time integration effort per MCU-HAL combination. First, the location of the HAL-specific abstraction function that handles the reception of network packets needs to be identified. This has to be done once per HAL as the function is consistent even across different MCUs. Second, an MCU-specific handler function—typically between 200 and 500 lines of code—that identifies the network packet buffer in RAM and injects the packet generated by Pemu needs to be written. As DMA-modeling for rehosting is an actively researched field, we consider this source of manual effort a temporary limitation imposed by the current capabilities of the underlying rehosting systems, rather than an inherent constraint of Pemu.

General Fuzzing Limitations. One of the most common limitations in general-purpose fuzzing is handling encryption, as mutations applied to encrypted data are unlikely to also produce validly encrypted data. To a certain degree, Pemu can help solve this for firmware fuzzing, as encryption algorithms can be integrated into Pemu in a similar manner to compression. However, Pemu cannot handle cases where a pre-shared secret or private key is required for authentication. In this case, reverse-engineering is necessary to either extract the secret or to replace, e.g., a certificate to enable Pemu to act as an authenticated server. Fortunately, there are many use cases where this is not necessary. For example, most embedded web servers do not require the client (in this case, the fuzzer + Pemu) to authenticate itself.

7 Related Work

Next, we discuss existing research on the two main areas on which the majority of our work focuses.

Firmware Rehosting and Fuzzing. Our work is based on the huge body of work in the field of embedded rehosting and fuzzing [6, 9, 10, 13, 15, 19, 28, 31, 38, 39, 41, 43, 51, 52, 54]. Especially Hoedur and Fuzzware by Scharnowski et al. [39, 41] and SEmu by Zhou et al. [54] serve as the base for the implementation of our approach. We expect that other rehosting methods can also benefit from the virtual network provided by Pemu. Beyond rehosting, research has also expanded to fuzzing embedded hardware [1, 8, 22, 26, 36] or using partial emulation to solve the hardware-dependency of firmware [20, 24, 29, 30].

Network Application Fuzzing. Network application fuzzing for general-purpose software has been extensively studied, leading to tools such as AFLNet [35], SGFuzz [3], StateAFL [32], Bleem [25], and Fuzztruction-Net [4], each introducing different methods to address this challenge. Existing approaches primarily target the application layer, leveraging modified network traffic seeds or OS functionalities to construct packets that encapsulate fuzzing input.

8 Conclusion

In this work, we introduced Pemu, the first automated approach for fuzzing the network stacks of embedded firmware. We tackle the challenges of ensuring well-formed encapsulation within network packets and awareness of the protocol state. Pemu focuses on the *in-depth* exploration of firmware by dynamically analyzing its network-related behavior and adapting to it to provide relevant fuzzing inputs encapsulated in seemingly real network packets. Due to its robust and automated analysis approach, it is possible to apply Pemu without requiring domain knowledge. To demonstrate how existing fuzzing methods can benefit from a virtual network stack, we integrated Pemu with three state-of-the-art rehosting platforms. In a comprehensive evaluation, we showed that our method provides an effective way of analyzing and fuzzing firmware using common types of embedded networking protocols.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy (EXC 2092 CASA – 390781972). This material is based upon work supported by the National Science Foundation under Award No. 2232915, 2146568, 2442984, and 2247954, as well as by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or ARPA-H.

References

[1] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing. In *USENIX Security Symposium*, 2021.

[2] Paschal C Amusuo, Ricardo Andrés Calvo Méndez, Zhongwei Xu, Aravind Machiry, and James C Davis. Systematically detecting packet validation vulnerabilities in embedded network stacks. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful Greybox Fuzzing. In *USENIX Security Symposium*, 2022.

[4] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. No Peer, no Cry: Network Application Fuzzing via Fault Injection. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[6] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[7] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[9] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. Multifuzz: A multi-stream fuzzer for testing monolithic firmware. In *USENIX Security Symposium*, 2024.

[10] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer.

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[11] Contiki-NG Team. Contiki-NG: The OS for Next Generation IoT Devices. https://github.com/contiki-ng/contiki-ng, as of September 10, 2025.

[12] Bo Feng, Meng Luo, Changming Liu, Long Lu, and Engin Kirda. AIM: Automatic Interrupt Modeling for Dynamic Firmware Analysis. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[13] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*, 2020.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11), 2020.

[16] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *sensors*, 2012.

[17] Harrison Green and Thanassis Avgerinos. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[18] Steve Heath. *Embedded Systems Design*. Elsevier, 2002.

[19] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[20] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[21] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[22] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave Jing Tian. PASAN: Detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications. In *USENIX Security Symposium*, 2021.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[24] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[25] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *USENIX Security Symposium*, 2023.

[26] Xiaoyue Ma, Qiang Zeng, Haotian Chi, and Lannan Luo. No more companion apps hacking but one dongle: Hub-based blackbox fuzzing of iot firmware. In *Conference on Mobile Systems, Applications and Services*, 2023.

[27] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[28] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[29] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *USENIX Security Symposium*, 2024.

[30] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A Multi-target Orchestration Platform. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[31] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[32] Roberto Natella. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering*, 27(7):191, 2022.

[33] Nordic Semiconductor. nRF Connect SDK: sdk-nrf. https://github.com/nrfconnect/sdk-nrf, as of September 10, 2025.

[34] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy. In *Annual Conference on Privacy, Security and Trust (PST)*, 2015.

[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[36] Wang Qinying, Chang Boyu, Ji Shouling, Tian Yuan, Zhang Xuhong, Zhao Binbin, Pan Gaoning, Lyu Chenyang, Payer Mathias, Wang Wenhai, and Beyah Reheem. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[37] John Romkey. Nonstandard for transmission of IP datagrams over serial lines: SLIP. RFC 1055, 1988.

[38] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *USENIX Security Symposium*, 2020.

[39] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[40] Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Wörner, Daniel Klischies, Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, Marius Muench, and Reviewing Model. GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays. In *USENIX Security Symposium*, 2025.

[41] Tobias Scharnowski, Simon Woerner, Felix Buchmann, Nils Bars, Moritz Schloegel, , and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *USENIX Security Symposium*, 2023.

[42] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[43] Lukas Seidel, Dominik Maier, and Marius Muench. Forming Faster Firmware Fuzzers. In *USENIX Security Symposium*, 2023.

[44] STMicroelectronics. STM32CubeF7 MCU Firmware Package. github.com/STMicroelectronics/STM32CubeF7/, as of September 10, 2025.

[45] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Adam Doupe, Tiffany Bao, Yan Shoshitaishvili, and Ruoyu Wang. Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation. In *USENIX Security Symposium*, 2023.

[46] George Thomas. Introduction to the modbus protocol. *The Extension*, 2008.

[47] Unicorn Engine. https://www.unicorn-engine.org/, as of September 10, 2025.

[48] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in Firmware Re-hosting, Emulation, and Analysis. *ACM Computing Surveys (CSUR)*, 54(1):1–36, 2021.

[49] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of Embedded Systems: A Survey. *ACM Computing Surveys (CSUR)*, 2022.

[50] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2013.

[51] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[52] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *USENIX Security Symposium*, 2021.

[53] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. SEmu GitHub Repository. https://github.com/MCUSec/SEmu, as of September 10, 2025, 2022.

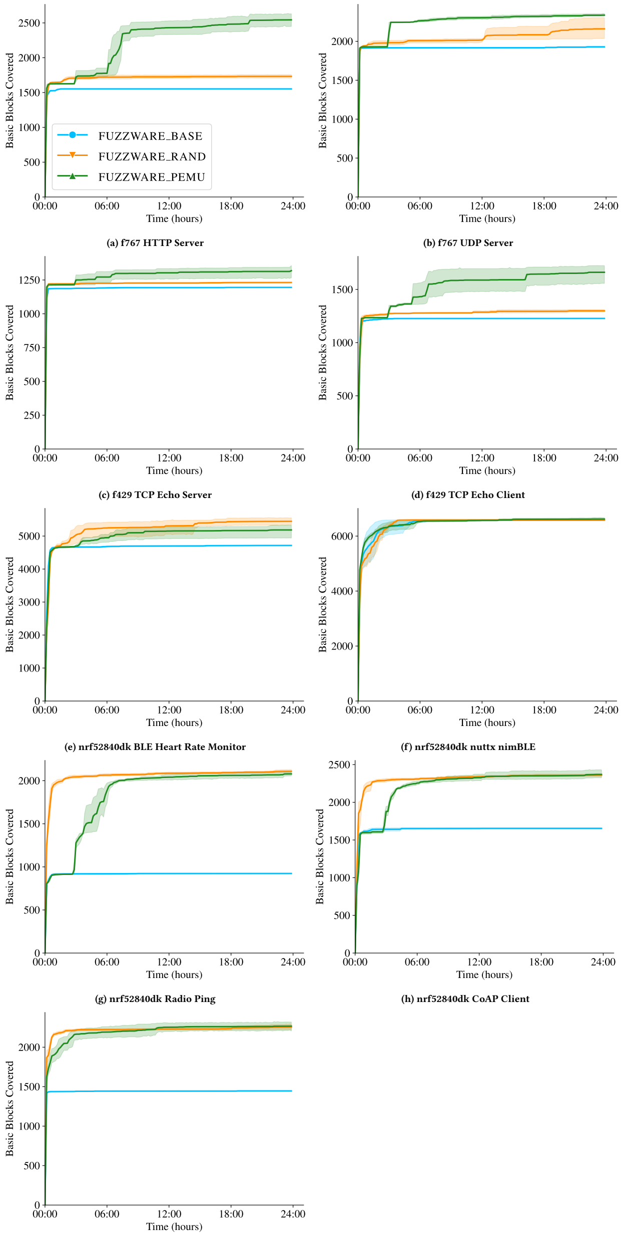
[54] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

A SEmu Results

We present the results of the SEmu experiment in Figure ??. The natural language processing approach by SEmu only supports a limited data set Therefore, we were only able to evaluate two samples.

B Fuzzware and Hoedur Results

The collected results of the evaluation with Fuzzware and Hoedur are displayed in Figure 9 and Figure 10. Both fuzzing platforms support the entire dataset. The plots show the median and the 95% confidence interval.



(a) f767 HTTP Server

(b) f767 UDP Server

(c) f429 TCP Echo Server

(d) f429 TCP Echo Client

(e) nrf52840dk BLE Heart Rate Monitor

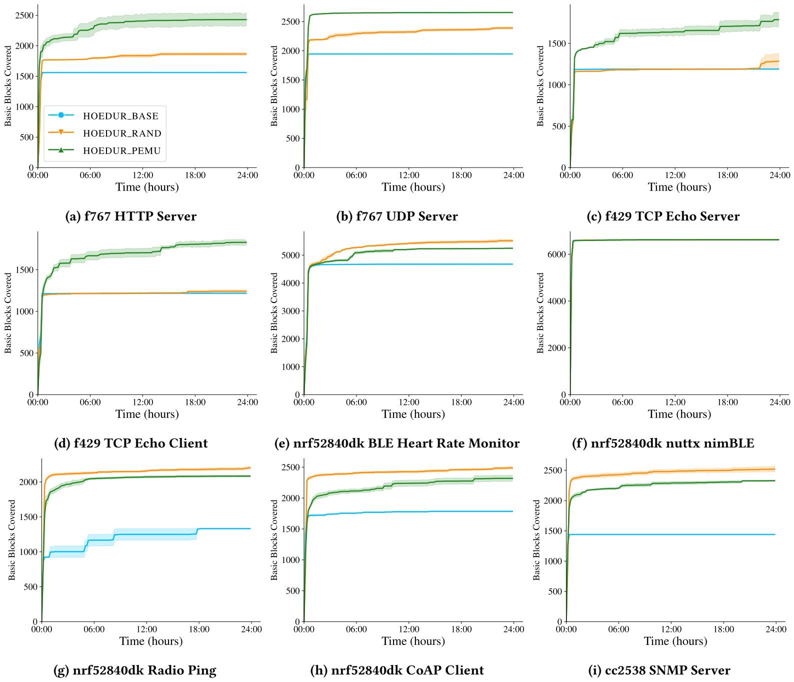
(f) nrf52840dk nuttx nimBLE

(g) nrf52840dk Radio Ping

(h) nrf52840dk CoAP Client

(i) cc2538 SNMP Server

Figure 9: Coverage plots of the ablation study on Fuzzware



(a) f767 HTTP Server

(b) f767 UDP Server

(c) f429 TCP Echo Server

(d) f429 TCP Echo Client

(e) nrf52840dk BLE Heart Rate Monitor

(f) nrf52840dk nuttx nimBLE

(g) nrf52840dk Radio Ping

(h) nrf52840dk CoAP Client

(i) cc2538 SNMP Server

Figure 10: Coverage plots of the ablation study on HOEDUR