future internet

MDPI

Review

IoT Firmware Emulation and Its Security Application in Fuzzing: A Critical Revisit

Wei Zhou [1,*], Shandian Shen [1] and Peng Liu [2,*]

[1] School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China; shenshandian@hust.edu.cn
[2] College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA
* Correspondence: weizhou_sec@hust.edu.cn (W.Z.); pliu@ist.psu.edu (P.L.)

Abstract: As IoT devices with microcontroller (MCU)-based firmware become more common in our lives, memory corruption vulnerabilities in their firmware are increasingly targeted by adversaries. Fuzzing is a powerful method for detecting these vulnerabilities, but it poses unique challenges when applied to IoT devices. Direct fuzzing on these devices is inefficient, and recent efforts have shifted towards creating emulation environments for dynamic firmware testing. However, unlike traditional software, firmware interactions with peripherals that are significantly more diverse presents new challenges for achieving scalable full-system emulation and effective fuzzing. This paper reviews 27 state-of-the-art works in MCU-based firmware emulation and its applications in fuzzing. Instead of classifying existing techniques based on their capabilities and features, we first identify the fundamental challenges faced by firmware emulation and fuzzing. We then revisit recent studies, organizing them according to the specific challenges they address, and discussing how each specific challenge is addressed. We compare the emulation fidelity and bug detection capabilities of various techniques to clearly demonstrate their strengths and weaknesses, aiding users in selecting or combining tools to meet their needs. Finally, we highlight the remaining technical gaps and point out important future research directions in firmware emulation and fuzzing.

Keywords: IoT devices; MCU; firmware; vulnerability detection; peripheral modeling; firmware emulation; fuzz testing; hardware-in-the-loop

Academic Editors: Olivier Markowitch and Jean-Michel Dricot

Received: 28 November 2024
Revised: 25 December 2024
Accepted: 2 January 2025
Published: 6 January 2025

Citation: Zhou, W.; Shen, S.; Liu, P. IoT Firmware Emulation and Its Security Application in Fuzzing: A Critical Revisit. *Future Internet* **2025**, *17*, 19. https://doi.org/10.3390/fi17010019

1. Introduction

A microcontroller unit (MCU) is a compact computer on a single integrated circuit, containing one or more CPUs (processor cores), memory like flash, ROM, or RAM, and programmable peripherals. MCUs are widely used in constrained IoT devices due to their low power consumption, compact size, cost-effectiveness, and ability to integrate peripherals for specific tasks. A recent report [1] states that over half of constrained IoT devices use MCUs. However, due to resource constraints, on-chip MCUs lack the security mechanisms present in general computers like x64, x86, and ARM Cortex-A, as well as systems like Windows and Linux [2,3]. For instance, most microcontrollers (MCUs) lack a memory management unit (MMU), which translates virtual memory addresses to physical ones. As a result, firmware on MCUs typically cannot support task isolation or address space randomization protection. In addition, firmware on MCU-based devices, often written in memory-unsafe languages like C/C++, is prone to memory bugs. Due to market pressure and cost constraints, these devices are frequently released without thorough code

auditing or security testing, leaving many vulnerabilities. Consequently, MCU-based devices are easily compromised, posing significant risks. The American Information Security Conference in April 2023 reported that over 50% of Distributed Denial of Service (DDoS) attacks stem from the growing number of compromised IoT devices [4].

The rise in vulnerabilities within MCU firmware has led researchers to adapt general software vulnerability detection techniques, particularly for fuzzing. Fuzzing has proven to be a highly effective tool for vulnerability detection over the past decade [5], uncovering thousands of security bugs in various software applications. However, directly applying fuzzing tests to MCU based devices is impractical for firmware due to the slow fuzzing speeds resulting from limited hardware resources. Researchers have attempted dynamic testing in emulated environments, but fuzzing firmware without the original device is difficult due to its tight integration with hardware interfaces (i.e., peripherals). Unlike traditional system-based programs that use well-defined I/O interfaces like POSIX APIs (e.g., `read(fd, buf, len)`), firmware must directly interact with peripheral registers and interfaces. For instance, while a Linux program reads input using `scanf` or `fread(buf, 1, size, fd)`, MCU firmware must first poll peripheral status registers (Line 3), read input from registers (Line 5), and check for errors (Line 6), as shown in Listing 1. The diverse configurations of peripherals across devices make universal emulation difficult. Consequently, there are still significant research gaps in developing automatic, high-fidelity firmware emulation environments and efficient fuzzing techniques.

Listing 1. Code snnippnet of external input reading from UART peripheral in a real-world MCU firmware.

```
1 uint8_t read(...) {
2     ...
3     while (!(USART1->SR & USART_SR_RXNE));
4     uint8_t data = USART1->DR;
5     if (USART1->SR & USART_SR_FE)  return -1;
6     else return data;
7 }
```

Our paper reviews MCU firmware emulation and its fuzzing applications, focusing on works from top international computer security conferences and journals over the past decade (2014–2024), including IEEE S&P, CCS, USENIX Security, NDSS, ACSAC, ASIA CCS, RAID, DSN, ESORICS, IEEE TIFS, and IEEE TDSC. Previous surveys only include works up to 2022 [6–8]. With recent advancements in embedded device emulation technology, especially from 2021 (as shown in Figure 1), it is crucial to revisit the state-of-the-art in MCU-based firmware emulation and its applications. Unlike previous surveys, such as those by Feng et al. [8], which summarize vulnerability detection methods across all firmware types, including Linux-based and MCU-based, on real devices and emulators, our study specifically targets MCU firmware emulation and its fuzzing applications. We deliberately exclude Linux-based firmware and real device fuzzing applications to focus on MCU firmware. This allows for a more detailed discussion of these tools and enables direct comparisons, as they share a common dataset. Moreover, while earlier works categorize studies based on technologies and applications, as seen in Fasano et al. [6] and Wright et al. [7], our approach begins by identifying the key features of MCU firmware to reveal the core challenges impacting emulation and fuzzing. We then reorganize the research on embedded device firmware emulators around these challenges, emphasizing how each study addresses them.

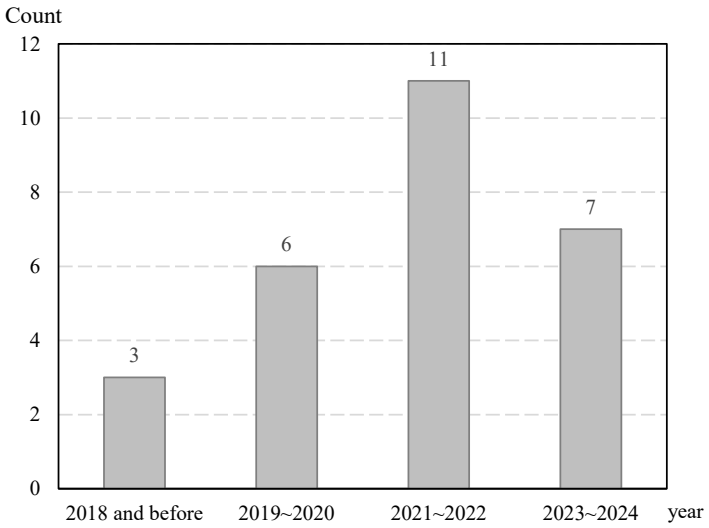

Figure 1. Number of papers related to MCU firmware emulators published in international conferences within 10 years.

The remainder of this article is structured as follows: Section 2 summarizes the software and hardware environment of MCU firmware, highlighting key differences from general software. Section 3 explores the challenges and requirements for full system emulation and enhanced fuzzing applications. Section 4 classifies recent work based on the challenges addressed, summarizing methods, advantages, and drawbacks. Section 5 compares the emulation fidelity and bug detection capabilities of existing approaches. Section 7 identifies technical gaps and suggests future research directions in firmware emulation and fuzzing. Finally, Section 8 concludes the article.

2. Feature of MCU Firmware

This section summarizes the background information of MCU firmware , highlighting three key features that differentiate it from general software as show in in Table 1.

Table 1. Comparison among MCU Firmware, General Purpose OS-based Firmware and General Software.

	Hardware Dependence	**OS Type**	**Layer Boundaries**	**Driver Diversity**	**Driver Interfaces**
General Software	Low	Few	Clear	Low	Standard
General-purpose OS Based Firmware	Less (Application) High (Driver)	Few	Clear	Moderate	Standard
MCU-based Firmware	High	Many	Blur	No	Custom

2.1. Embedded Firmware Classification and Microcontroller-Based Firmware

Firmware refers to all software on an embedded device, including drivers, systems, and applications. It can be categorized into three types based on the system types used [7].

- **General-purpose OS-based firmware.** This firmware uses adapted versions of traditional operating systems like Linux and Windows for embedded environments, maintaining most desktop functionalities. They rely on high-end CPU architectures like x86 and ARM Cortex-A, allowing general OS functions such as memory isolation of processes. These systems can only run on high-end CPU-based embedded devices like routers, with system and application components compiled separately.
- **Specialized Embedded OS-based firmware.** This category of systems employs specialized embedded operating systems, predominantly real-time operating systems

(RTOS), which have lower computing power but higher real-time requirements. Examples of such RTOS include Zephyr, FreeRTOS, and RIoT. They are typically found in commercial IoT products like small drones. In these systems, the OS and applications are compiled into a unified binary.

- **Bare-metal firmware.** This category includes firmware without an OS abstraction (also called Bare-metal Firmware). The application code directly accesses hardware and is also statically linked into a single binary file, as seen in small sensors.

Due to limited computing and storage resources, only specialized embedded operating systems or bare-metal firmware can run on MCUs. In this paper, "MCU firmware or firmware" refers to this specialized embedded OS or bare-metal firmware.

2.2. Features of MCU Firmware

F1: Diverse hardware dependency environments. Unlike general software that interacts directly with users or external files, firmware relies on hardware peripherals for external interaction, making it tightly to these components. Figure 2 illustrates the hardware environment of a drone based on ARM Cortex-M4 MCU, showcasing various peripherals. These peripherals are categorized into on-chip (e.g., SPI, I2C, UART) and off-chip (e.g., WiFi, Button). Firmware accesses off-chip peripherals indirectly through on-chip peripherals. Firmware interacts with peripherals in four ways (highlighted with red lines):

① **Peripheral Registers R/W:** In most MCUs, such as ARM-based ones, peripheral hardware registers are memory-mapped (MMIO), allowing the processor to access them for status, configuration, and data queries. In a few MCUs, such as Intel 8051, these registers are accessed via port instructions.

② **IRQ:** Actively verifying peripheral registers reading values wastes CPU cycles when no events occur. Many MCUs support configuring peripherals to trigger interrupts for events like timer completions or new data arrivals, allowing firmware to handle them promptly in interrupt service routines (ISRs).

③ **DMA:** To enhance high-throughput data transfer, some MCUs offer Direct Memory Access (DMA), enabling peripherals to transfer data directly to and from main memory, bypassing peripheral register reading via processor for faster large data transfers.

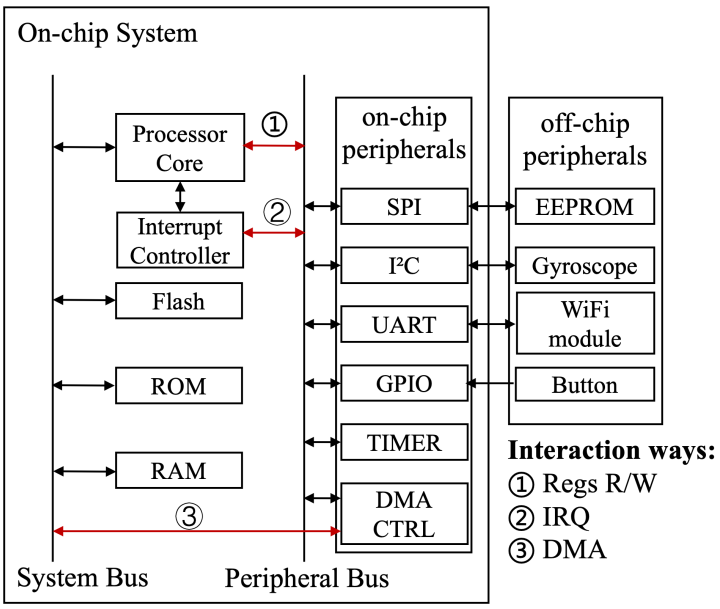

Figure 2. Hardware components of a drone based on ARM Cortex-M4 MCU.

Firmware functionality varies significantly across devices, such as drones and door locks, due to different peripheral controls and methods that can change at runtime. Consequently, peripheral usage by firmware is highly varied, involving numerous methods.

F2: Blurring of boundaries and interface among application, library, and driver code. As illustrated in Figure 3, in general, MCU firmware consists of three layers: the driver layer, the operating system layer, and the application layer. The application layer, developed by programmers, contains the device's functional logic. The system layer includes real-time operating systems like FreeRTOS, Zephyr, and MbedOS for task scheduling and control, along with protocol libraries for external interactions. The driver layer manages hardware peripherals, with most code provided by manufacturers through board support packages (BSP) or open-source hardware abstraction layers (HALs). However, unlike general OSs and software, the layer boundaries in firmware are often blurred, with many components being optional, as shown in Figure 3. For instance, FreeRTOS [9] allows library or application code to interact with hardware directly, bypassing system calls like bare-metal firmware. Moreover, driver code can invoke system calls, such as system queue operations, making it difficult to distinguish driver layers from other code layers.

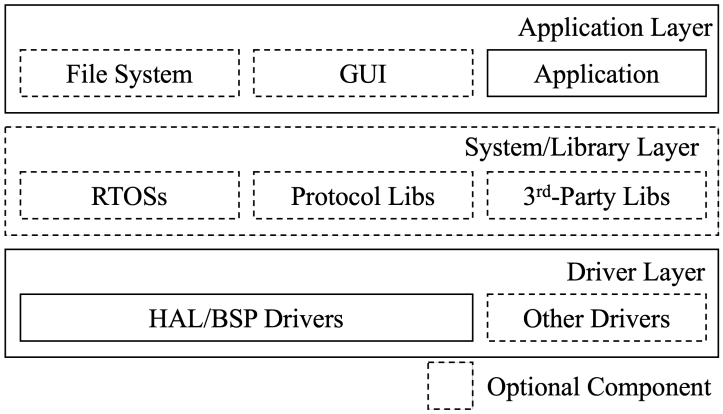

Figure 3. Common components and layers in MCU firmware.

F3: Fragmentation of MCU Firmware Ecosystem. MCU-based devices are tailored for specific tasks, such as sensors or drones, and their firmware is fragmented, unlike the dominant OSs like Linux or Windows. Over 20 IoT operating systems exist, each with its own compatible libraries, such as RT-Thread [10], which offers over 700 packages with multiple versions. Different MCU vendors provide their own HAL/BSP libraries, like STM32CubeMX [11], which are specific to their MCUs. These libraries often implement different driver interfaces for peripheral controls, leading to incompatibility across various HAL/BSP functions and OS drivers. Developers can also integrate third-party libraries or create custom driver code for unique peripherals, compounding to the fragmented software stack in the MCU firmware ecosystem.

3. Challenges Faced by Firmware Emulation and Its Application in Fuzzing

This section highlights the challenges and the ideal goal of firmware emulation and its application in fuzzing, focusing on three aspects of support for firmware emulation approaches. First, CPU emulation, including CPU registers and instruction set translation, is crucial for executing across various hardware architectures. Second, emulating peripheral interactions is challenging due to the diverse peripherals and lack of general system interfaces in MCU-based devices. Lastly, since firmware emulation aims for dynamic testing, it must support fuzzing tests.

3.1. CPU Emulation Support

Although MCU devices use diverse CPU architectures, the number is typically limited to fewer than ten, such as ARM, MIPS, and RISC-V. Tools like *QEMU* [12] and *Unicorn* [13] already support a wide range of commonly used embedded architectures. However, devices used in specialized fields like industrial or automotive applications, such as ECUs, may employ esoteric or custom architecture-specific instructions. Therefore, to emulate MCU firmware, support for any given CPU architecture is necessary.

3.2. Peripheral Interaction Emulation Support

As mentioned in Section 2, while CPU emulation can be manually achieved, the diverse configurations of individual devices make manual support for hundreds or thousands of peripherals impractical. For instance, *QEMU* supports only a few peripherals for fewer than ten MCU versions. Although *QEMU* lists the STM32F405 MCU, it actually supports only six on-chip peripherals, like UART and SPI, out of more than 20 available. Moreover, *QEMU* does not fully support peripheral configurations and interactions; for example, it supports SPI only via peripheral register read/write, lacking interrupt and DMA support. Initially, research like *Avatar* [14] proposed a hardware-in-the-loop solution that emulates only the CPU and forwards peripheral access to real hardware. However, this method suffers from performance issues, as peripheral operations on MCUs are much slower than virtual memory operations on servers (MCU CPU frequency is in the hundreds of MHz, compared to x64 CPUs operating over GHz) and cannot be used for parallel testing. Peripheral emulation should ideally be automatic, have high applicability (capable of emulating any MCU peripheral functions), and offer high fidelity (replicating the control and data flow of a real device without actual hardware).

3.3. Fuzzing Application Enhancement

Fuzzing refers to an automated software testing technique that providing invalid, unexpected, or random data as inputs to a computer program to test whether the program perform as expected, which has been widely used in vulnerability detections. In general software fuzzing, the process includes input generation, delivery, execution, feedback collection, and feedback-guided input mutation. It is divided into a front-end (input generation and mutation) and a back-end (execution environment and feedback). In firmware fuzzing within an emulation environment (see Figure 4), the emulator serves as the back-end, facilitating (1) test case reception, (2) emulated execution, (3) feedback collection, and (4) crash detection. The emulated execution environment relies on CPU and peripheral emulation. Inputs should be delivered when the firmware reads data from peripherals, as fuzzing aims to uncover software bugs, which are more exploitable than hardware bugs. Unlike general software with defined data I/O interfaces (as discussed in F2 and F3 in Section 2), identifying external input/output interfaces in black-box firmware is challenging. For feedback, fuzzers mainly use code coverage, requiring the emulator to support instruction emulation at the basic block level and collect path and basic block feedback [15]. For bug detection, general fuzzers detect bugs by using existing security mechanisms within the target system or an integrated sanitizer. The sanitizer identifies bugs by detecting undefined or suspicious behavior through instrumentation code inserted by the compiler at runtime. However, MCU firmware, often using RTOS or bare-metal resources, lacks these mechanisms and source code for instrumentation, leading to undetected memory corruptions, also known as “silent” bugs [16]. Therefore, the emulator needs to support more fine-grained memory monitoring for bug detection. An ideal fuzzing application on an emulator should have low false-positive and -negative rates for bug detection and testing at high speed.

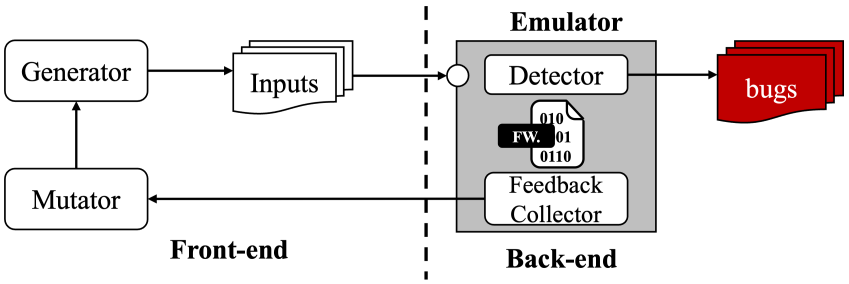

Figure 4. Firmware fuzzing in an emulated environment.

4. State-of-the-Art Firmware Emulation and Its Application in Fuzzing

In this section, we summarize and reorganize the tools from all the research included in Figure 1, categorizing them by the main challenges they address. As shown in Figure 5, since CPU emulation is a fundamental component of firmware emulation, the following works all depend on CPU emulation tools. Tools built on the same platform are generally compatible, as demonstrated by the compatibility between *Fuzzware* and *Halucinator*.

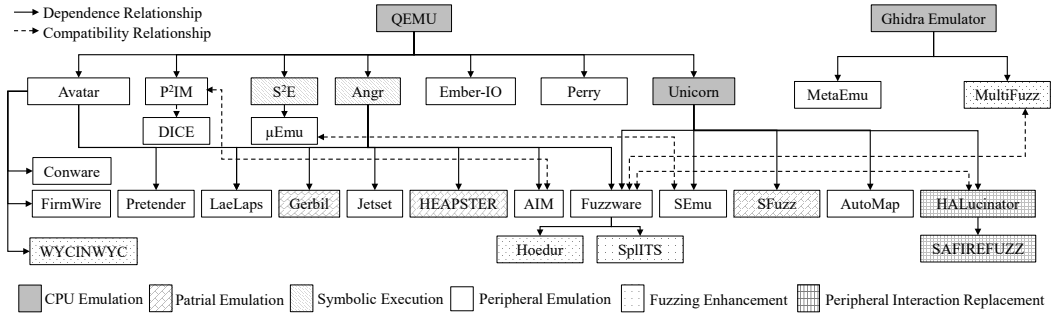

Figure 5. Dependence and compatibility among firmware emulators (the single-arrow line denotes dependence, showing one tool is built upon or enhanced by another; the dotted double-arrow line denotes compatibility, connecting tools that can be used together directly).

4.1. Solution to CPU Emulation Support

General binary emulators like *Simics* [17], *QEMU* [12], and *Unicorn* [13] support most common CPU architectures. *QEMU* and *Unicorn* also offer memory mapping and peripheral forwarding, making them popular for full-system emulation with peripheral interaction emulation tools. However, these emulators struggle with unknown CPU architectures, requiring significant manual effort to add new ones within their Intermediate Representation (IR) frameworks. The *Ghidra* Emulator [18], developed by the NSA, supports more instruction set architectures than *QEMU* and allows users to add new ones using SLEIGH language definitions. For instance, *MetaEmu* [19] addresses the challenge of uncommon processor architectures in automotive firmware through the *Ghidra* Emulator.

In addition, *SAFIREFUZZ* [20] pinpoint instruction translation as a key factor slows down emulation. Instead of translating instructions for a general processor, it executes firmware on more powerful hardware within the same instruction set family. By using high-level peripheral function replacement and dynamic binary rewriting, it replaces peripheral interactions and filters ARM Cortex-M specific instructions, allowing ARM Cortex-M firmware to run on ARM Cortex-A servers without CPU emulation.

4.2. Solutions to Peripheral Interaction Support

Aside from the hardware-in-the-loop solutions, three approaches have been proposed for handling diverse peripheral interaction in an emulated environment as summarized in Table 2.

4.2.1. High-Level Peripheral Interaction Replacement

For Linux-based firmware, researchers can hook the Linux system call interface and redirect it to driver function replacements for full-system emulations, as seen in *Firmadyne* [21], *FirmAE* [22], and *ECMO* [23]. Inspired by this, although MCU-based firmware lacks a standard system call interface and has diverse OS kernels, researchers found that hardware abstraction layer (HAL) interface functions are more common. They proposed intercepting firmware calls to the HAL and replacing them with emulator functions, as performed by *HALucinator* [24]. However, this approach requires a library matching tool to identify HAL functions in the firmware and manual creation of replacement functions, limiting its applicability and increasing labor costs.

4.2.2. Peripheral Interaction Modeling

To emulate diverse peripheral hardware with minimal manual effort, existing research proposes various automatic peripheral interaction modeling methods.

Hardware Record-based method. *Pretender* [25] enhances hardware-in-the-loop solutions for full-system emulation by recording real hardware feedback and using machine learning to create peripheral emulation models. However, *Pretender* still needs the same hardware as the firmware, which is often hard to access for practical firmware analysis. Similarly, *Conware* [26] relies on hardware peripheral operation logs to build peripheral models.

Human Heuristic-based method. *P*^{2}*IM* [27] introduces the first hardware-independent automated method for creating peripheral emulation models. It infers peripheral register types from firmware peripheral access patterns and establishes feedback models based on these types. While effective for simple interactions, its reliance on human assumptions limits its success with complex peripherals. *DICE* [28] uses a similar approach by modeling DMA access patterns to automate DMA emulation models, supporting firmware emulation with DMA-based peripherals. However, it is also constrained by human assumptions, making it unsuitable for complex devices like Ethernet and USB DMA. *FirmWire* [29] offers human-crafted peripheral models for baseband firmware.

Symbolic Constraints-based method. Peripheral modeling using symbolic contrarians is a common approach in current emulation solutions to address the limitations of human heuristics-based approach. *Laelaps* [30] uses symbolic execution to handle unknown peripheral reads by returning symbolic values and exploring program paths. However, it requires continuous symbolic execution, leading to high overhead. To address this, *µEmu* [31] automatically determines execution paths by analyzing execution states. It uses symbolic execution to learn path constraints on valid execution state and create a peripheral feedback knowledge base, which is used during dynamic execution, eliminating the need for continuous symbolic execution and reducing overhead. *Jetset* [32] employs a search strategy with incremental control flow graphs to guide symbolic execution for firmware execution, similar to *µEmu*. *Fuzzware* [33] pinpoints models based on single or few path constraints for overly restricting reachable paths. It uses symbolic execution to generate broad peripheral feedback constraints, allowing more mutations during fuzz testing to explore additional paths. This increases code coverage compared to previous models but also results in more false positives than real hardware. Note that continuous symbolic execution for peripheral input can lead to path explosion and high overhead, so current solutions use local symbolic execution within functions handling peripheral input without exploring all paths.

Fuzzing Feedback-based method. *Ember-IO* [34] introduces a feedback peripheral access method using a fuzz testing feedback scheme. Instead of generating complete peripheral models, it generates all hardware inputs through fuzz testing, caching only those values

that achieve higher code coverage to accelerate execution. This method increases code coverage quickly but also results in more false positives and negatives.

External Information-based method. In 2022, Zhou et al. [35] noted that previous methods for generating peripheral models relied solely on firmware, which lacks complete peripheral logic, such as interrupt timing. This limitation leads to false positives in firmware-oriented peripheral modeling. To enhance the accuracy of peripheral emulation models without hardware reliance, they introduced *SEmu*. *SEmu* uses natural language processing (NLP) to learn peripheral behavior from hardware manuals, converting this information into structured conditional-action rules. By executing and linking these rules during runtime, *SEmu* dynamically creates a peripheral model for each accessed peripheral, achieving nearly 100% consistency with real hardware at the basic block level for simple peripherals. Similarly, *Perry* [36] develops high-fidelity peripheral emulator models using external open-source peripheral drivers.

4.2.3. Partial Emulation

For research on specific vulnerabilities or code snippets in firmware where full system emulation is not required, researchers [37–39] propose slicing the code snippet without peripheral interactions. For instance, *HEAPSTER* [37] focuses on detecting heap vulnerabilities within firmware by emulating only a limited number of functions related to heap operations.

Table 2. Summary of peripheral interaction emulation solution.

Method		Years	Tools	Fidelity *	Automation
Hardware-in-the-loop		2014	*Avatar* [40]	High	Low
		2022	*AutoMap* [41]	High	Low
High-level Replacement		2020	*HALucinator* [24]	Moderate	Low
		2023	*SAFIREFUZZ* [20]	Moderate	Low
Peripheral Modeling	Symbolic Constraints-based	2020	*Laelaps* [30]	Moderate	Low
		2021	*µEmu* [31]	Moderate	Moderate
		2021	*Jetset* [32]	Moderate	Moderate
		2022	*Fuzzware* [33]	Moderate	High
		2022	*MetaEmu* [19]	Low	Low
		2024	*AIM* [42]	Moderate	Moderate
	Hardware Record-based	2019	*Pretender* [25]	Moderate	Moderate
		2021	*Conware* [26]	Moderate	Moderate
	Human Heuristic-based	2020	*P²IM* [27]	Moderate	Moderate
		2021	*DICE* [28]	Moderate	Moderate
		2022	*FirmWire* [29]	Moderate	Low
	Fuzzing Feedback-based	2023	*Ember-IO* [34]	Moderate	High
	Manual-based	2022	*SEmu* [35]	High	Low
	Driver-based	2024	*Perry* [36]	High	Low
Partial Emulation		2019	*Gerbil* [38]	Low	Low
		2022	*HEAPSTER* [37]	Moderate	Low
		2022	*SFuzz* [39]	Low	Low

*: The fidelity level combines execution and data/memory fidelity, as shown in Figure 6.

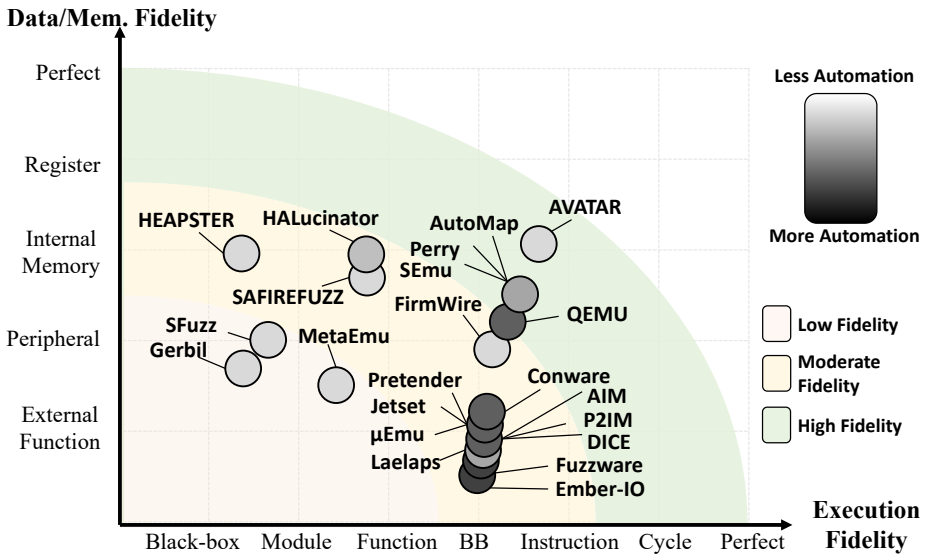

Figure 6. Fidelity of firmware emulators.

4.3. Solutions to Fuzzing Application Enhancement

Recent advancements in peripheral emulation technology have led researchers to redesign the general fuzzing technology to better accommodate real-world firmware-specific features including coverage feedback, input generation and mutation, and crash detection as summarized in Table 3.

Coverage Feedback. As mentioned in Section 2, peripheral interrupts can actively trigger ISR functions causing uncertain bursts in firmware execution. The exact point where the interrupt occurs is usually not crucial, but switching execution to the ISR at different points can create many meaningless new edges for edge-based coverage feedback fuzzers. To address this, *Ember-IO* [34] remaps edge-coverage feedback to separate edges in ISRs from normal execution, eliminating invalid edges from unpredictable interrupts. *MultiFuzz* directly uses basic block coverage to avoid these issues. Additionally, firmware often performs multi-byte magic string checks for command processing in protocols. *SplITS* [43] introduces feedback mechanisms for input-to-state mapping and seed retention for targeted replacement mutations, efficiently handling multi-byte comparisons.

Input Generation and Mutation. Since real-world firmware typically not only uses single peripheral but multiple peripherals at the same time, as shown in the drone in Figure 2. However, existing fuzzers for general applications typically offer only a single flat input. Thus, *Hoedur* [44] and *MultiFuzz* [45] enhanced general fuzzing techniques, especially for input generation and mutation, to address the multi-stream nature of firmware inputs from various peripherals.

Bug Detection. Firmware often contains "silent" bugs that do not cause device crashes. While typical firmware fuzzing applications use basic crash detection, they often miss these silent memory corruptions. To address this, *µSBS* [46] statically instruments binaries to make memory corruptions observable. However, static binary rewriting for firmware is challenging and lacks accuracy, making these solutions suitable only for simple bare-metal firmware.

Table 3. Summary of fuzzing application enhancement solution.

Firmware Feature	Enhanced Component	Years	Tools
Random Peripheral Interrupt	Coverage Feedback	2023	*Ember-IO* [34]
Multi-stream Input	Input Generation & Mutation	2023	*Hoedur* [44]
Magic Value Check		2023	*SplITS* [43]
Multi-stream Input		2024	*MultiFuzz* [45]
Silent Bug	Bug Detection	2018	*WYCINWYC* [16]
		2020	*μSBS* [46]

5. Emulation Fidelity

Emulation fidelity is vital for firmware emulation, affecting its accuracy and applicability in security applications. We qualitatively classify and compare fidelity among firmware emulators, following the approach by Wright et al. [7]. This involves two dimensions: *execution fidelity*, which measures control flow similarity to real hardware, and *data/memory fidelity*, which assesses data consistency in the MCU device (memory and registers) and data output to device peripherals compared to real hardware. Figure 6 summarizes the fidelity distribution of these emulators and their degree of automation.

5.1. Execution Fidelity

Execution fidelity can be categorized into black-box, module-level, function-level, basic-block, instruction-level, cycle-level, and perfect. As illustrated in Figure 5, firmware emulators use *QEMU* [12], *Unicorn* [13], and *Ghidra* [18] for instruction set translation. They convert instructions from a target set to a host set at the basic block level, using an intermediate representation (IR) for translation. This approach provides execution fidelity at most the basic block and instruction levels, even without considering peripheral emulation. Consequently, no firmware emulator can achieve cycle-level fidelity, which accurately replicates the CPU instruction cycle as real hardware does, or perfect fidelity, where the emulator's internal execution mirrors that of the actual machine.

Avatar achieves the highest execution fidelity by forwarding all peripheral access to real hardware. Full system emulation solutions emulate all peripherals to execute entire firmware code, thus their execution fidelity relies on the accuracy of peripheral modeling. Solutions like *AutoMap*, *SEmu* and *Perry*, employ real hardware, hardware descriptions, or driver source code to construct individual models for each peripheral, offering higher fidelity compared to *QEMU*'s manually crafted peripheral emulation. Firmware-specific modeling solutions such as *Jetset*, P^2IM, and *Fuzzware*, while more automated, depend on general assumptions without detailed hardware information, leading to lower fidelity. *Conware*, which does not model peripherals but instead uses fuzzing input, results in even lower fidelity. Partial emulation solutions such as *SFuzz*, *Gerbil*, and *HEAPSTER* support only specific firmware functions, achieving module-level execution fidelity. For instance, *HEAPSTER* concentrates on heap operations using heuristic methods, with many functions returning constraint-solved values without actual execution. High-level function replacement solutions fall in the middle; they bypass all peripheral emulation by hooking and replacing the original driver function with a manually crafted function on the host. This approach misses the execution of peripheral-related instructions, thereby ensuring function-level execution consistency. Generally, firmware emulation based solely on the firmware itself tends to have lower execution fidelity. Achieving higher fidelity necessitates additional hardware information and the development of specific peripheral models for various peripherals.

5.2. Data/Memory Fidelity

Data/memory fidelity ordered from coarsest to finest granularity are as follows:

- **External Function:** Data output from firmware in an emulator leads to similar functions as real hardware.
- **Peripheral:** Data output from an emulator matches that of real hardware.
- **Internal Memory:** Data in the memory for running firmware in an emulator is consistent with real hardware.
- **Register:** Data in memory and registers for running firmware in an emulator is consistent with real hardware.
- **Perfect:** All data input, output, and memory space in the emulator matches real hardware at all times, which is virtually unobtainable.

Avatar interacts with real peripherals, ensuring high data memory fidelity. However, due to basic block translation, it handles peripheral interrupts only between blocks, not simultaneously with triggers, maintaining internal memory fidelity. In full system emulation, data fidelity relies on peripheral modeling methods. Solutions like *SEmu*, which use external info for individual peripheral modeling, offer higher fidelity than general firmware-specific models like P^2IM and *Fuzzware*. In partial emulation, fidelity depends on the consistency of emulated module functions. For example, while *HEAPSTER* only emulates heap-related functions, resulting in low execution fidelity, it excludes peripheral data and maintains memory consistency with real hardware. *FirmWire*'s function replacement and peripheral emulation specific to baseband firmware also achieve high fidelity but with less automation. Similarly, high-level replacement solutions using functional replacement functions can achieve higher data consistency than peripheral emulation solutions. As shown in Figure 6, higher execution fidelity enhances data and memory fidelity in full system emulation. In addition, partial emulations or HAL replacement solutions bypass driver code and involve fewer peripheral operations, resulting in data and memory consistency that closely resembles the real device.

Fidelity VS Automation: To achieve higher-fidelity peripheral modeling, simply analyzing firmware itself for general peripheral modeling is insufficient. Specific modeling for different peripherals requires hardware information. Extracting this information directly from the device can lead to hardware dependence issues, while obtaining it from hardware reference manuals or driver code is complex and error-prone. This complexity demands more human effort and reduces automation in high-fidelity modeling, as shown in Figure 6. In contrast, when focusing on a specific application domain, achieving high-fidelity peripheral emulation may not be essential. For instance, HAL-based solutions, as discussed in [24], can be utilized for testing application code exclusively. However, these solutions necessitate the manual implementation of replacement functions. Similarly, partial emulation can be employed to identify bugs that are less pertinent to peripheral operations, such as specific heap-related issues [37], which demand higher-fidelity emulation solely in memory. Nonetheless, the removal of peripheral-dependent code often requires manual intervention, which can impede automation due to indistinct boundaries and custom interfaces, as elaborated in Section 2. In summary, high-fidelity firmware emulation results in reduced automation, as illustrated in Figure 6 and Table 2.

6. Bug Detection Capability of Emulator's Fuzzing Application

Given that the main use of current firmware emulators is fuzzing, this section further compares and discusses their bug detection capabilities in fuzzing applications. We focus on emulators whose main purpose is fuzzing and which have been tested for at least 24 h over multiple targets. Emulators not primarily used for fuzzing, such *Gerbil* [38],

HEAPSTER [37], and *AIM* [42], are excluded. Similarly, *Conware* [26] and *Avatar* [14], which perform short-time and single-target fuzzing without a dedicated fuzzing process adoption, are also out of scope, as are *Pretender* [25], *Laelaps* [30], and *Jetset* [32]. In our study, we evaluate emulators that satisfy our predefined criteria by comparing their relative false-positive and false-negative rates. This comparison utilizes results derived from fuzzing experiments conducted on shared datasets. Specifically, we employ real-world firmware samples from P^2IM [27], which have been tested across all the tools under consideration. For instance, the *Fuzzware* tool research paper reports a higher false-positive rate, and subsequent studies confirm this using P^2IM samples. In contrast, *SEmu* tests the same dataset with no false positives, indicating a low false-positive rate. Meanwhile, *MultiFuzz* claims to find more bugs and achieve higher coverage on the same dataset as *Fuzzware*, suggesting a lower false-negative rate. The results are summarized in Table 4. Table 4 also outlines their fuzzing strategies, including the main functions in front-end and back-end components.

6.1. False Positives

False positives in firmware fuzzing with emulators often result from low-fidelity peripheral emulation, leading to increased manual effort for bug reproduction and confirmation.

Low fidelity in peripheral emulation increases false positives. For instance, feeding fuzzing test cases to peripheral hardware responses can lead to false crashes. Tools like *Fuzzware* and *Ember-IO* might trigger peripheral interrupts before peripheral initialization and even feed fuzzing test cases to hardware-generated MMIO registers reading, such as SRs, causing uninitialized function pointers in interrupt service routines (ISRs) to lead to false crashes. In comparison, fine-grained peripheral models, which incorporated additional hardware information like *SEmu* and *Perry*, have relatively low false-positive rates. Similarly, *FirmWire* and *MetaEmu*, designed for specific applications, rely on manual configuration and tailored test constraints, resulting in fewer false positives. The accuracy of fuzzing enhancements relies on the fidelity of their emulators.

6.2. False Negatives

False negatives in firmware fuzzing can have serious consequences, particularly for security-sensitive devices in applications such as automotive, healthcare, and industry. False negatives can be caused by two reasons: limited code coverage and bug detection capabilities. Greater coverage means more parts of the firmware have been tested, which typically results in fewer false negatives, and the ability to detect more types of bugs in less time also reduces false negatives.

Coverage relies on both emulation capability and fuzzing strategies. If peripheral emulation fails, the firmware may stall, resulting in limited coverage. Improving emulation fidelity can increase coverage and reduce false negatives. Lower fidelity emulation can increase error handling coverage in firmware, but it may mislead fuzzing mutations to focus more on error handling rather than normal operations. For fuzzing strategies, many fuzzing applications rely on general-purpose fuzzers like AFL [47], AFL++ [48], or libafl [49]. Many fuzzing applications use general-purpose fuzzers such as AFL [47], AFL++ [48], and libafl [49]. To achieve higher coverage, recent research [34,43–45] has enhanced general fuzzing technologies for real firmware targets by improving input generation, mutation, and coverage feedback. For instance, *Hoedur* enhances libfuzzer [50] by handling multiple peripheral input streams and applying mutation techniques specific to firmware, such as recognizing peripheral input types and performing cross-input stream mutations, resulting in higher coverage than previous methods.

Table 4. Comparison of firmware emulation's fuzzing applications in bug finding capability.

Tools	Mutation and Generation	Input Interface/Constraints	Coverage Feedback	Bug Detection	FP	FN
HALucinator [24]	AFL	Specified Func./None	Edge	Basic Mem. Error+ Heap-checking	Moderate-low	Moderate
SAFIREFUZZ [20]	LibAFL	Specified Func./None	Non-colliding Edge	Basic Mem. Error	Moderate-low	Moderate
P²IM [27]	AFL	Identified Data IO/None	Edge	Basic Mem. Error	Moderate-high	Moderate
μEmu [31]	AFL	Identified Data IO/None	Edge	Basic Mem. Error	Moderate	Moderate
DICE [28]	AFL	Identified Data IO/None	Edge	Basic Mem. Error	Moderate	Moderate
Ember-IO [34]	AFL/AFL++	MMIO/Fuzz Cache	Functionally Equivalent Edge	Basic Mem. Error	High	Moderate
Fuzzware [33]	AFL/AFL++	MMIO/Modeling	Edge	Basic Mem. Error	Moderate-high	Moderate
SEmu [35]	AFL/AFL++	Identified Data IO/None	Edge	Basic Mem. Error	Moderate-low	Moderate
Perry [36]	AFL	Specified Func./None	Edge	Basic Mem. Error	Moderate-low	Moderate
Hoedur [44]	LibFuzzer + Δ	MMIO/Multi-Stream +Modeling	Functionally Equivalent Edge	Basic Mem. Error	Emulator Depended	Moderate-low
SplITS [43]	AFL + I2S	MMIO/Multi-Stream +Modeling	Edge + Comparison +Len.	Basic Mem. Error		Moderate-low
MultiFuzz [45]	AFL + I2S+ Len Extension	Specified Func./None	Block	Basic Mem. Error		Moderate-low
μSBS [46]		Fuzzer front-end Depended		Memory Sanitizer *		Moderate-low
SFuzz [39]	AFL++	Specified Func./None	Edge	Basic Mem. Error	Moderate	Moderate
FirmWire [29]	AFL++	Specified Func./Protocol	Edge + Per-task	Basic Mem. Error	Moderate-low	Moderate-low
MetaEmu [19]	LibAFL	Specified Function/None	Edge	Basic Mem. Error	Moderate-low	Moderate
AutoMap [41]	AFL	Identified Data IO/None	Edge	Basic Mem. Error	Moderate-low	Moderate-high

Δ: Len Extension + Cross-value + Cross-stream + Type-awareness. *: includes Null Pointer Dereference, Stack/Heap-based buffer overflow, format string, and double free detection.

Most firmware fuzzers includes basic memory error detection targeting memory segment permission violations, such as buffer overflows or access to unmapped memory. While studies suggest more detailed memory error detection and diverse bug identification, these solutions often come with high overhead or require source code, which is not feasible for real-world firmware [16]. Additionally, they are often specific to simple bare-metal firmware, making them unsuitable for complex firmware like *µSBS* [46].

7. Discussion and Future Research

In this section, we explore current approaches for improved usage and propose directions for future research, as outlined in Figure 7.

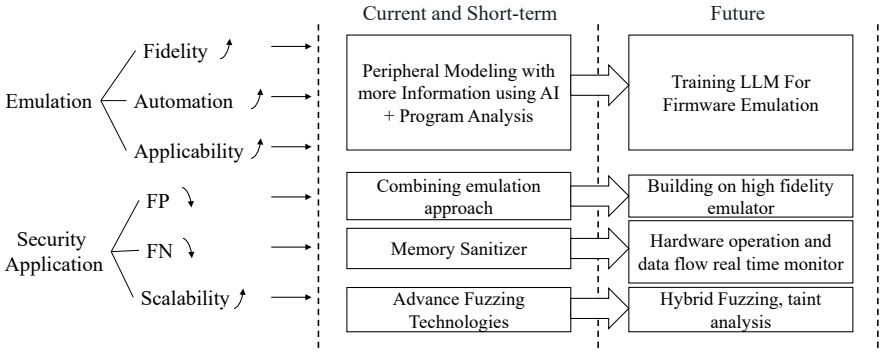

Figure 7. Roadmap for future research directions.

7.1. Selection and Combination of Different Emulation Approaches

With the rise of firmware emulation approaches, various solutions offer distinct advantages for different peripherals and application scenarios, affecting their performance and stability across test sets. Selecting or combining emulators to test target firmware can achieve higher fidelity and efficiency. For instance, while "hardware-in-the-loop" solutions have hardware dependence, it provides a realistic debugging environment through real hardware connections to accomplish high-fidelity emulation. If developers test a device with an unknown peripheral that current tools cannot emulate, they can forward this peripheral to the real hardware while keeping other peripherals in emulation. This approach facilitates testing the device efficiently compared to complete "hardware-in-the-loop" solutions.

HALucinator bypasses peripheral emulation by replacing high-level hardware abstraction layer (HAL) functions with host implementations. However, building a comprehensive database requires HAL source codes from major MCU vendors, limiting support for SoCs with proprietary SDKs. Combining HAL replacement with peripheral modeling approach could create an advanced firmware emulator with higher execution and data fidelity at the same time. Developers can use *HALucinator* to match HAL functions and hook them to host implementations, while peripheral modeling tools can emulate unknown peripherals during runtime.

For applications like baseband or automotive, specialized firmware emulators such as *FirmWire* and *MetaEmu* are preferable over general ones, as they offer more features tailored to these specific uses.

7.2. Basic Information Recovery for Emulation

Most current firmware emulators require developers to configure basic memory layout, start address, and other firmware information, assuming this can be found in the MCU hardware manual or in formats like ELF files with symbol tables. However, firmware dumped from real-world devices is often a pure binary without any information, even the MCU model. Unlike general software binaries, basic information such as memory

loading address, entry address, and other definitions vary with different MCU models. For accurate firmware dynamic emulation, knowledge of this information is essential. The absence of this basic information significantly limits the practicality of current academic dynamic analysis tools for real firmware and incurs much manual effort to identify these information before running the tools. Combining statistical analysis techniques, such as determining base addresses through fixed static global variable address analysis, was performed in [51]. The automation of the extraction and construction of a comprehensive firmware information knowledge base is a key technical challenge for firmware emulators.

7.3. Automated High-Fidelity Emulation for Complex and Custom Peripherals

Firmware in safety-critical applications demands high-fidelity emulation for fuzzing, as even minor errors can have severe consequences. For instance, small mistakes in ADAS firmware can result in traffic accidents, and deviations in medical IoT device firmware could endanger lives. However, devices in automotive and industrial sectors often use complex or custom peripherals that current emulators cannot accurately replicate. For example, there is no existing tool that can automatically model complex peripherals like USB or WiFi with precision. As a result, firmware analysis for these devices relies heavily on statistical analysis or custom emulation models, requiring significant manual effort. Automating the accurate generation of complex peripheral emulation models without real hardware is a major technical challenge.

Advancements in AI technology offer new opportunities for achieving high-fidelity peripheral emulation more automatically. For instance, *Conware* converts logs of real peripheral interactions (such as interrupts, reads, and writes) into directed acyclic graphs (DAGs) and uses a novel graph-transformation technique to generate high-fidelity peripheral models. Advanced Reinforcement Learning [52] can also be applied to continually learn from recorded logs, enhancing peripheral modeling. Similarly, `SEmu` utilizes NLP to automatically generate peripheral models from hardware manuals, improving fidelity for complex peripherals. The development of large language models like GPT-4, which have shown greater capability than traditional NLP [53], may further enhance peripheral modeling by learning from hardware manuals. With the advancement of LLMs, we can train specialized models for firmware emulation using extensive firmware datasets, hardware manuals, and open-source MCU software stacks in future.

7.4. More and Effective Security Application Support

Currently, advanced fuzzing techniques like hybrid fuzzing [54], directed fuzzing [55], and stateful fuzzing [56], along with dynamic bug detectors such as *MSan* [57] and *UBSan* [58], are not widely adopted for firmware targets. This is primarily due to two reasons: first, the black-box nature of binaries makes instrumentation and context inference, such as state variables, challenging for directed, stateful fuzzing and sanitizers. Second, although symbolic execution has been integrated into emulation environments, it is mainly used for peripheral modeling rather than continuous hybrid fuzzing like *QSYM* [54]. This limitation arises from the need to modify machine code for improved symbolic execution performance, which is tightly coupled with host architecture. The diverse and distinct nature of firmware architectures presents challenges for such applications. To advance fuzzing technology, future efforts should combine it with statistical analysis for better information inference and focus on instrumenting intermediate representations rather than directly on binary code to support diverse CPU architectures.

Firmware emulation security applications are mainly limited to fuzz testing due to low data and memory fidelity. As emulation fidelity improves, more security analysis technologies, such as taint analysis for detecting vulnerabilities like command injections,

can be applied to firmware targets. Additionally, MCU devices are used in security-sensitive environments. Compliance checks with hardware operations are crucial, as errors that do not affect control or data flow can still cause serious issues, such as failing to engage brakes in an ASDS.

8. Conclusions

The increasing security vulnerabilities in microcontroller unit (MCU)-based devices have heightened the focus on vulnerability detection in MCU firmware. Current research emphasizes testing firmware in emulated environments to achieve higher levels of automation and efficiency, as well as to extract more internal information for identifying a broader range of security issues. Our paper revisits the latest MCU firmware emulation approaches, focusing on its key features and identifying core challenges in CPU emulation, peripheral emulation, and fuzzing application enhancement. We categorize existing emulators based on fidelity and bug detection capabilities to help users choose the right tools. Although advancements have been made, current emulators still face challenges in balancing fidelity, automation, and adaptability, indicating room for improvement. We also explore future research opportunities and how emerging technologies, such as LLM, can address these challenges. We posit that the integration of binary analysis with advanced AI technologies will significantly enhance the automation, fidelity, and applicability of firmware emulation and expanding security applications, making it a prominent area of interest in the near future.

Funding: Peng Liu was funded by National Science Foundation under grant number CNS-2019340, ECCS-2140175, the Department of Energy Office of Cybersecurity, Energy Security, and Emergency Response (DOE CESER) under grant number RC-40125b-2023. Wei Zhou and Shandian Shen were funded by the National Natural Science Foundation of China under grant number 62202188.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Foundation, E. IoT and Embedded Survey Report 2024. Available online: https://outreach.eclipse.foundation/iot-embedded-developer-survey-2024 (accessed on 21 December 2024).
2. Abbasi, A.; Wetzels, J.; Holz, T.; Etalle, S. Challenges in designing exploit mitigations for deeply embedded systems. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P), Stockholm, Sweden, 17–19 June 2019; pp. 31–46.
3. Zhou, W.; Jiang, Z.; Guan, L. Understanding MPU Usage in Microcontroller-based Systems in the Wild. In Proceedings of the Workshop on Binary Analysis Research (BAR) 2023, San Diego, CA, USA, 3 March 2023.
4. NSFoCus. Insight RSA 2023: Botnet Threat Situation Observation. Available online: http://blog.nsfocus.net/rsa-2023insight2/ (accessed on 19 November 2024).
5. Godefroid, P. Fuzzing: Hack, art, and science. *Commun. ACM* **2020**, *63*, 70–76. [CrossRef]
6. Fasano, A.; Ballo, T.; Muench, M.; Leek, T.; Bulekov, A.; Dolan-Gavitt, B.; Egele, M.; Francillon, A.; Lu, L.; Gregory, N.; et al. Sok: Enabling security analyses of embedded systems via rehosting. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (CCS), Online, Hong Kong, 7–11 June 2021; pp. 687–701.
7. Wright, C.; Moeglein, W.A.; Bagchi, S.; Kulkarni, M.; Clements, A.A. Challenges in firmware re-hosting, emulation, and analysis. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–36. [CrossRef]
8. Feng, X.; Zhu, X.; Han, Q.L.; Zhou, W.; Wen, S.; Xiang, Y. Detecting vulnerability on IoT device firmware: A survey. *IEEE/CAA J. Autom. Sin.* **2022**, *10*, 25–41. [CrossRef]
9. FreeRTOS. FreeRTOS LTS libraries. Available online: https://www.freertos.org/Documentation/03-Libraries/01-Library-overview/03-LTS-libraries/01-LTS-libraries (accessed on 21 December 2024).
10. RT-Thread. RT-Thread Packages. Available online: https://packages.rt-thread.org/ (accessed on 21 December 2024).
11. STMicroelectronics. STM32Cube Initialization Code Generator. Available online: https://www.st.com/en/development-tools/stm32cubemx.html (accessed on 21 December 2024).

12. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track, Anaheim, CA, USA, 10–15 April 2005; pp. 41–46.
13. Quynh, N.A.; Vu, D.H. Unicorn: Next Generation CPU Emulator Framework. Available online: https://www.blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf (accessed on 21 December 2024).
14. Muench, M.; Nisi, D.; Francillon, A.; Balzarotti, D. Avatar 2: A multi-target orchestration platform. In Proceedings of the Workshop on Binary Analysis Research (BAR) 2018, San Diego, CA, USA, 18 February 2018; pp. 1–11.
15. Kargén, U.; Shahmehri, N. Speeding up bug finding using focused fuzzing. In Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES) 2018, Hamburg, Germany, 27–30 August 2018; pp. 1–10.
16. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS) 2018, San Diego, CA, USA, 18–21 February 2018.
17. Magnusson, P.S.; Christensson, M.; Eskilson, J.; Forsgren, D.; Hallberg, G.; Hogberg, J.; Larsson, F.; Moestedt, A.; Werner, B. Simics: A full system simulation platform. *Computer* **2002**, *35*, 50–58. [CrossRef]
18. Agency, N.S. NSA's Research Directorat Ghidra. Available online: https://ghidra-sre.org/ (accessed on 19 November 2024).
19. Chen, Z.; Thomas, S.L.; Garcia, F.D. Metaemu: An architecture agnostic rehosting framework for automotive firmware. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, 7–11 November 2022; pp. 515–529.
20. Seidel, L.; Maier, D.C.; Muench, M. Forming Faster Firmware Fuzzers. In the Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, USA, 9–11 August 2023; pp. 2903–2920.
21. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. Towards automated dynamic analysis for linux-based embedded firmware. 23rd Annual Network and Distributed System Security Symposium (NDSS) 2016, San Diego, California, USA, 21- 24 February 2016; Volume 1, p. 1.
22. Kim, M.; Kim, D.; Kim, E.; Kim, S.; Jang, Y.; Kim, Y. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC) 2020, Online/Austin, TX, USA, 7–11 December 2020; pp. 733–745.
23. Jiang, M.; Ma, L.; Zhou, Y.; Liu, Q.; Zhang, C.; Wang, Z.; Luo, X.; Wu, L.; Ren, K. ECMO: Peripheral transplantation to Rehost embedded Linux kernels. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS), Online, Republic of Korea, 15–19 November 2021; pp. 734–748.
24. Clements, A.A.; Gustafson, E.; Scharnowski, T.; Grosen, P.; Fritz, D.; Kruegel, C.; Vigna, G.; Bagchi, S.; Payer, M. HALucinator: Firmware re-hosting through abstraction layer emulation. In Proceedings of the 29th USENIX Security Symposium, Online, 12–14 August 2020; pp. 1201–1218.
25. Gustafson, E.; Muench, M.; Spensky, C.; Redini, N.; Machiry, A.; Fratantonio, Y.; Balzarotti, D.; Francillon, A.; Choe, Y.R.; Kruegel, C.; et al. Toward the analysis of embedded firmware through automated re-hosting. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID) 2019, Chaoyang District, Beijing, China, 23–25 September 2019; pp. 135–150.
26. Spensky, C.; Machiry, A.; Redini, N.; Unger, C.; Foster, G.; Blasband, E.; Okhravi, H.; Kruegel, C.; Vigna, G. Conware: Automated modeling of hardware peripherals. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (AsiaCCS), Online, Hong Kong, 7–11 June 2021; pp. 95–109.
27. Feng, B.; Mera, A.; Lu, L. P^2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In Proceedings of the 29th USENIX Security Symposium, Online, 12–14 August 2020; pp. 1237–1254.
28. Mera, A.; Feng, B.; Lu, L.; Kirda, E. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 1938–1954.
29. Hernandez, G.; Muench, M.; Maier, D.; Milburn, A.; Park, S.; Scharnowski, T.; Tucker, T.; Traynor, P.; Butler, K. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In Proceedings of the Network and Distributed Systems Security Symposium (NDSS) 2022, San Diego, CA, USA, 24–28 April 2022.
30. Cao, C.; Guan, L.; Ming, J.; Liu, P. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC) 2020, Austin, TA, USA, 7–11 December 2020; pp. 746–759.
31. Zhou, W.; Guan, L.; Liu, P.; Zhang, Y. Automatic firmware emulation through invalidity-guided knowledge inference. In Proceedings of the 30th USENIX Security Symposium, Online, 11–13 August 2021; pp. 2007–2024.
32. Johnson, E.; Bland, M.; Zhu, Y.; Mason, J.; Checkoway, S.; Savage, S.; Levchenko, K. Jetset: Targeted firmware rehosting for embedded systems. In Proceedings of the 30th USENIX Security Symposium, Online, 11–13 August 2021; pp. 321–338.
33. Scharnowski, T.; Bars, N.; Schloegel, M.; Gustafson, E.; Muench, M.; Vigna, G.; Kruegel, C.; Holz, T.; Abbasi, A. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In Proceedings of the 31st USENIX Security Symposium, Boston, MA, USA, 10–12 August 2022; pp. 1239–1256.

34. Farrelly, G.; Chesser, M.; Ranasinghe, D.C. Ember-IO: Effective firmware fuzzing with model-free memory mapped IO. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (AsiaCCS), Melbourne, VIC, Australia, 10–14 July 2023; pp. 401–414.
35. Zhou, W.; Zhang, L.; Guan, L.; Liu, P.; Zhang, Y. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, 7–11 November 2022; pp. 3269–3283.
36. Lei, C.; Ling, Z.; Zhang, Y.; Yang, Y.; Luo, J.; Fu, X. A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers. In Proceedings of the 33rd USENIX Security Symposium, Philadelphia, PA, USA, 14–16 August 2024; pp. 7085–7102.
37. Gritti, F.; Pagani, F.; Grishchenko, I.; Dresel, L.; Redini, N.; Kruegel, C.; Vigna, G. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 1082–1099.
38. Yao, Y.; Zhou, W.; Jia, Y.; Zhu, L.; Liu, P.; Zhang, Y. Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution. In Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS) 2019, Luxembourg, 23–27 September 2019; pp. 638–657.
39. Chen, L.; Cai, Q.; Ma, Z.; Wang, Y.; Hu, H.; Shen, M.; Liu, Y.; Guo, S.; Duan, H.; Jiang, K.; et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, 7–11 November 2022; pp. 485–498.
40. Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D.; et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS) 2014, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 1–16.
41. Won, J.Y.; Wen, H.; Lin, Z. What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling. In Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID) 2022, Limassol, Cyprus, 26–28 October 2022; pp. 200–213.
42. Feng, B.; Luo, M.; Liu, C.; Lu, L.; Kirda, E. AIM: Automatic Interrupt Modeling for Dynamic Firmware Analysis. *IEEE Trans. Dependable Secur. Comput.* **2023**, *21*, 3866–3882. [CrossRef]
43. Farrelly, G.; Quirk, P.; Kanhere, S.S.; Camtepe, S.; Ranasinghe, D.C. SplITS: Split Input-to-State Mapping for Effective Firmware Fuzzing. In Proceedings of 28th European Symposium on Research in Computer Security (ESORICS) 2023, The Hague, The Netherlands, 25–29 September 2023; pp. 290–310.
44. Scharnowski, T.; Wörner, S.; Buchmann, F.; Bars, N.; Schloegel, M.; Holz, T. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, USA, 9–11 August 2023.
45. Chesser, M.; Nepal, S.; Ranasinghe, D.C. MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware. In Proceedings of the 33rd USENIX Security Symposium, Philadelphia, PA, USA, 14–16 August 2024; pp. 5359–5376.
46. Salehi, M.; Hughes, D.; Crispo, B. μSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID) 2020, San Sebastian, Spain, 14–15 October 2020; pp. 381–395.
47. Zalewski, M. American Fuzzy Lop. Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 21 December 2024).
48. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT) 2020, Online, 11 August 2020.
49. Fioraldi, A.; Maier, D.C.; Zhang, D.; Balzarotti, D. Libafl: A framework to build modular and reusable fuzzers. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, 7–11 November 2022; pp. 1051–1065.
50. LLVM. libFuzzer—A Library for Coverage-Guided Fuzz Testing. Available online: https://llvm.org/docs/LibFuzzer.html (accessed on 19 November 2024).
51. Wen, H.; Lin, Z.; Zhang, Y. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS), Online, USA, 9–13 November 2020; pp. 167–180.
52. Shakya, A.K.; Pillai, G.; Chakrabarty, S. Reinforcement learning algorithms: A brief survey. *Expert Syst. Appl.* **2023**, *231*, 120495. [CrossRef]
53. Baktash, J.A.; Dawodi, M. Gpt-4: A review on advancements and opportunities in natural language processing. *arXiv* **2023**, arXiv:2305.03195.
54. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
55. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, Texas, USA, 30 October–3 November 2017; pp. 2329–2344.

56. Ba, J.; Böhme, M.; Mirzamomen, Z.; Roychoudhury, A. Stateful greybox fuzzing. In Proceedings of the 31st USENIX Security Symposium, Boston, MA, USA, 10–12 August 2022; pp. 3255–3272.
57. Stepanov, E.; Serebryany, K. MemorySanitizer: Fast detector of uninitialized memory use in C++. In Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Francisco, CA, USA, 7–11 February 2015; pp. 46–55.
58. Developers, L. Undefined Behavior Sanitizer. Available online: https://clang.llvm.org/docs/UndeïňĄnedBehaviorSanitizer.html (accessed on 19 November 2024).