Khost: KVM-based Near Native MCU Firmware Rehosting

Chunlin Wang, Yicheng Yang, Yuan Zhang, Haoyu Xiao, Yifan Zhang, Jiarun Dai
Fudan University, China

Abstract

Microcontroller Unit (MCU)-based devices constitute a critical layer of the Internet of Things (IoT) infrastructure, so ensuring their security is of paramount importance. Rehosting-based dynamic MCU firmware analysis is an effective approach to securing these devices. However, existing rehosting frameworks commonly suffer from substantial performance overhead due to emulation or diminished execution scope.

To address these limitations, we propose **Khost**, a near-native, scope-preserving rehosting framework. It extends the KVM by introducing a lightweight extended CPU, an auxiliary page table, and a software-based interrupt controller, enabling MCU firmware to be rehosted on high-performance platforms with minimum overhead. It also provides a memory-mapped I/O (MMIO) monitor for quick peripheral interactions and a wrapper for firmware to enable coverage collection and configure the existing fuzzing engines flexibly. Evaluations on two standard benchmarks show that Khost reduces overhead by 90.0% to 95.5% for complex computational tasks and by up to 98.5% for MCU system-level operations, compared to QEMU. Furthermore, fuzzing on 12 real-world firmware with Khost achieves up to 197.5× higher throughput and improves basic block coverage by 6x compared to existing fuzzing tools. Additionally, Khost successfully uncovers 5 previously unknown bugs.

1 Introduction

MCUs, as compact and energy-efficient computing units, are widely used in embedded devices [41, 65], such as industrial controllers, healthcare machines, smart home devices, etc. These devices serve as a cornerstone in the development and proliferation of the IoT ecosystem. However, hundreds of vulnerabilities [10, 64] have been discovered in MCU firmware recently, resulting in denial of service, privacy breaches, data corruption, and even physical damage to devices.

Dynamic analysis technologies [22, 38, 69], such as fuzzing, play an important role in detecting vulnerabilities in traditional programs. However, some obstacles hinder the practicality of these technologies for MCU firmware. First, MCU firmware is typically designed for specific purposes, such as controlling sensors or managing communication protocols. As a result, it is tightly coupled with customized hardware, making it challenging for execution or analysis on general-purpose computers or servers. Second, the limited hardware resources of MCU devices make it hard to deploy an efficient dynamic analysis environment directly. Third, MCU devices are highly diverse and deployed in large quantities, making large-scale analysis costly due to the substantial hardware acquisition expenses. Moreover, conducting analysis directly on hardware devices also causes irreversible hardware damage.

Firmware rehosting [12, 19, 66, 67], which runs firmware in a virtual environment provided by general computers or servers, has been proven to be a viable approach to overcome the above limitations. However, as shown in Table 1, existing rehosting approaches expose their limitations. First, translation-based approaches create virtual execution environments with a QEMU-like emulator (①②③), causing poor performance due to the overhead from dynamic binary translation and software Memory Management Unit (soft-MMU). The performance becomes worse when combined with hardware-based peripheral interactions. Second, High Level Emulation (HLE)-based approaches (③④) require substantial human effort to write replacement functions for Hardware Abstraction Layers (HALs). Even worse, some low-level system instructions in HALs (e.g., `svc`) are essential for RTOS-like firmware, but cannot be accurately emulated[1], thereby narrowing the scope of firmware execution. Moreover, Native Execution-based approaches (④) convert the firmware into an ARM-based Linux application via binary rewriting to improve rehosting efficiency. However, such approaches cannot solve the incompatibilities between different instruction set architectures (ISAs), thus have to compromise the accuracy of instruction execution and system modeling, and diminish the execution scope.

[1] SafireFuzz [52] acknowledges this issue in the paper.

Table 1: The comparison among different firmware rehosting approaches from the following perspectives: (1) the efficiency in handling the rehosting tasks; (2) the scope of target firmware behaviors that the rehosting framework can accurately handle; and (3) the degree of automaticity in performing the rehosting tasks without human intervention.

No.	Method (ISA + Peripheral)	Related tool	Efficiency	Scope	Auto.
①	QEMU-like Emulator + Hardware Peripheral	Avatar [68], GDBFuzz [17]	low	medium	low
②	QEMU-like Emulator + Peripheral Model	P^{2}IM [21], Fuzzware [50]	medium	medium	high
③	QEMU-like Emulator + Hand-Written Library Function	HALucinator [11]	medium	low	medium
④	Native Execution + Hand-Written Library Function	SafireFuzz [52]	very high	low	medium
⑤	KVM-based Emulator + Peripheral Model	**Khost**	high	medium	high

Kernel-based Virtual Machine [27], or KVM, is a hardware-assisted virtualization technology for general-purpose processors that enables near-native execution performance. Many modern ARM 64-bit high-performance processors support KVM and provide backward compatibility with 32-bit operations, allowing them to execute most operations typically found in MCUs. Therefore, we can leverage KVM to efficiently rehost MCU firmware on high-performance processors while preserving the original firmware behavior. First, the absence of an MMU in MCUs significantly undermines KVM's correctness and performance in memory handling. Second, the instruction set profiles used by MCUs and high-performance processors are incompatible, hindering the direct execution of MCU firmware on KVM. Third, the Nested Vectored Interrupt Controller (NVIC), which is essential for managing asynchronous events in MCUs, is not supported on high-performance platforms. Finally, the lack of built-in instrumentation for coverage collection and efficient peripheral interaction mechanisms significantly impedes the effectiveness of fuzzing MCU firmware under KVM.

In this paper, we present **Khost**, a near-native and scope-preserving MCU firmware rehosting framework based on KVM, which enables MCU firmware to be rehosted on high-performance ARM platforms. Khost addresses the above challenges through the following components: (1) an auxiliary page table, which creates a virtual MMU to bridge the memory management differences between MCUs and high-performance processors; (2) an extended CPU, for handling MCU-specific operations with minimal overhead; (3) a software-based NVIC that allows the MCU asynchronous events to be correctly processed during execution; (4) an MMIO Monitor, for configuring existing peripheral models and providing quick peripheral interactions; (5) a wrapper for firmware to enable coverage collection and configure the existing fuzzing engines flexibly.

We evaluate the rehosting performance of Khost with benchmark workloads from CoreMark-PRO [13] and Simbench [59]. The results demonstrate that Khost reduces overhead by 90.0% to 95.5% on handling complex computational tasks compared to QEMU under the same test environment. And it achieves up to a 98.5% reduction in overhead when performing system-level operations. Furthermore, we assess the performance of fuzzing with Khost on 12 real-world firmware. The results show up to 197.5× improvement in fuzzing throughput and a 6x increase in basic block coverage compared to HALucinator [11], as well as 68.6× and up to 3.4x improvements, respectively, over Fuzzware [50]. In all, Khost successfully uncovers 5 previously unknown bugs.

In summary, we make the following contributions:

- We design a near-native, scope-preserving rehosting framework that extends KVM to rehost MCU firmware on high-performance ARM platforms, while enabling coverage collection, rapid peripheral interactions, and flexible configuration of existing fuzzing engines.
- We implement our approach in a prototype tool called Khost. And its source code can be accessed publicly at: https://github.com/seclab-fudan/Khost
- Evaluations on two benchmarks demonstrate that Khost outperforms QEMU in handling complex computational tasks and system-level tasks. And it achieves higher throughput and code coverage than existing approaches when fuzzing on 12 real-world firmware, while uncovering 5 new bugs.

2 Background

2.1 MCU Firmware & Rehosting

MCU firmware [21, 58] refers to the low-level software that directly manages and coordinates hardware components in embedded devices. It is commonly deployed in resource-constrained environments designed for low cost and energy efficiency, such as Drones, 3D printers, and programmable logic controllers (PLCs). Typically, MCU firmware is implemented as a monolithic binary that integrates peripheral drivers, a lightweight operating system (OS) library, and application-specific logic. For example, the firmware in a smart door lock contains drivers for motor control and wireless communication modules, a lightweight security protocol stack, as well as mechanisms for user authentication, access logging, and handling remote commands.

MCU firmware rehosting refers to the process of building a virtual environment that closely approximates the hardware dependencies of a given MCU firmware image [19]. This process is essential for various firmware analysis techniques, including fuzz testing. The central aim of rehosting is to ensure that the behavior of the firmware running in the virtual environment closely reflects its behavior on the real hardware [26]. Achieving such behavioral fidelity requires precise modeling of two fundamental elements: (i) the ISA, and (ii) the hardware peripherals. This necessity stems from the inherent architectural and peripheral-level discrepancies between embedded MCU systems and the analyst's host platform.

2.2 Memory Types for ARM

The memory of the ARM is typically divided into several regions, each with its own set of attributes. These attributes include access permissions, such as read and write access at different privilege levels, as well as the memory type and cache policy. In the ARMv8-A architecture, memory regions are classified into two mutually exclusive types [47]: `Normal` and `Device`. Every region in the address space must be configured as one of these two types. The `Normal` memory is used for all code and most data regions, including RAM, Flash, and ROM. It supports attributes such as cacheable and is weakly ordered, allowing both the processor and compiler to perform aggressive optimizations.

In contrast, `Device` memory is used for memory-mapped peripherals, where accesses may have side effects. For instance, reading from a timer register, which is a typical MMIO register, can yield different values on each read, making the access non-repeatable. Device memory is further classified into four subtypes: Device-nGnRnE, Device-nGnRE, Device-nGRE, and Device-GRE. Each subtype imposes progressively fewer constraints on memory access behavior. Among them, Device-nGnRnE is the most restrictive, disallowing all forms of optimization such as access merging (G), reordering (R), and speculative reads (E), which results in the lowest memory access performance.

Incorrect memory type or attribute assignments can undermine OS execution, leading to unexpected behavior, exceptions, or even system crashes. For example, misclassifying `Normal` memory as `Device` disables caching and out-of-order execution, leading to significant performance degradation.

2.3 ARM Cortex-A/M

ARM is one of the most widely used processor architectures, especially in embedded systems [67]. It defines three main profiles: Cortex-A, Cortex-R, and Cortex-M. Cortex-R targets real-time tasks, while Cortex-A and Cortex-M are designed for high-performance and low-power scenarios, respectively [46, 52]. Notably, within the embedded market, 32-bit ARMv7-A and ARMv7-M variants remain widely

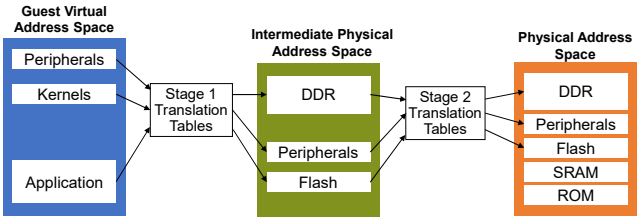

Figure 1: Two-Stage Address Translation of KVM on ARM architecture [2]

used [28, 52, 65]. ARMv7-A supports two execution modes: (1) ARM mode, which uses fixed-length 32-bit instructions aligned on 4-byte boundaries. (2) Thumb mode, which supports both 16-bit and 32-bit instructions, allows for denser code, making it well-suited for systems with limited memory. And two modes are dynamically switchable. In contrast, ARMv7-M is tailored for MCU devices and only supports the Thumb-v2 instruction set.

The more recent ARM architectures, such as ARMv8-A, introduce the AArch64 extension, which provides a 64-bit instruction set. Although processors based on ARMv8-A are primarily intended for high-performance platforms such as mobiles, desktops, and servers, they retain support for 32-bit execution through the AArch32 instruction set. This enables backward compatibility with legacy ARM software, allowing legacy 32-bit ARM software to run within a 64-bit OS.

2.4 KVM Technology

KVM [27] is a hardware-assisted virtualization technology that has been integrated into the Linux kernel. It directly uses hardware extensions provided by processors to achieve near-native performance. However, to use KVM, processors must support hardware virtualization features, such as Intel VT-x, available in most Core and Xeon processors, and ARM Virtualization Extensions, implemented in ARMv8-A cores like Cortex-A57 and Cortex-A72. Both the host, which refers to the physical machine and system responsible for providing virtualization, and the guest, representing the virtual machine (VM) or firmware environment being virtualized, must conform to the same architectural profile.

In the ARM architecture, Exception Levels (EL0 to EL3), first introduced in ARMv8-A, define a hierarchical model of execution privileges. EL0 is designated for user applications, EL1 for the OS kernel, EL2 for the hypervisor (a privileged layer responsible for managing VMs), and EL3 for the secure monitor with the highest privilege level. The `trap` (to hypervisor) and `eret` mechanisms, functionally analogous to `vm_exit` and `vm_enter` in x86 virtualization, are fundamental to enforcing isolation and enabling virtualization. These mechanisms allow the hypervisor managing VMs and hardware resources to intercept sensitive operations initiated by guest OSes. Such operations primarily involve the execution

of privileged instructions, access to MMIO, and interactions with system registers. When these events occur, the processor triggers a `trap` that saves the guest state and transfers control to the hypervisor. Then, the hypervisor handles the event, typically by emulating the faulting instruction or delegating the operations to a user-space device model provided by emulators such as QEMU. Subsequently, execution is returned to the guest via the `eret` operation. This trap-based design enforces strong isolation, preventing guests from directly manipulating sensitive system resources. Unfortunately, frequent `trap` operations incur significant performance overhead due to context switching and instruction emulation [27].

Memory virtualization is a critical mechanism for enforcing isolation and ensuring the OS's security in KVM. As shown in Figure 1, memory accesses in an ARM virtualization environment are subject to a two-stage address translation process. The OS-controlled Stage 1 translation utilizes page tables to map virtual addresses to what the OS perceives as physical addresses, known as Intermediate Physical Addresses (IPAs), which are not actual physical memory. Instead, a Stage 2 translation managed by the hypervisor maps IPAs to real physical addresses. This gives the hypervisor full control over the VM's memory view, including access to memory-mapped system resources and their placement within the VM's address space. This fine-grained control of memory access is crucial for isolation and sandboxing, ensuring that a VM can only access the resources allocated to it and remains unaware of resources assigned to other VMs or the hypervisor itself.

3 Challenges

Efficient and scope-preserving MCU firmware rehosting is essential for security analysis and dynamic debugging. Processors based on the ARMv8-A architecture are widely deployed and support KVM technology, while maintaining compatibility with 32-bit operations. It enables them to support most functionalities of the ARMv7-M architecture. Leveraging this feature, we can extend KVM to rehost firmware on high-performance ARMv8-A processors, achieving near-native execution efficiency and facilitating downstream tasks such as fuzzing. However, several challenges hinder the direct use of KVM for rehosting firmware on ARMv8-A platforms.

3.1 Ineffective Memory Access Handling

As discussed in Section 2.2, correctly distinguishing between Normal and Device memory and assigning appropriate attributes is crucial for the correctness and performance. Improper configuration can cause inconsistent behavior and lead to significant performance degradation.

In KVM, both Stage 1 and Stage 2 memory mappings define attributes such as memory type and access permissions. The host OS combines these attributes to determine the final effective value, typically selecting the more restrictive one. Specially, when the Stage 1 MMU is disabled, all memory regions are treated as Device-nGnRnE [47]. The absence of an MMU in the MCU means that its firmware cannot configure Stage 1 mappings, leaving the Stage 1 address translation disabled. As a result, all memory regions accessed by the MCU firmware are treated as Device-nGnRnE, implying that the data cache is disabled and no read/write optimizations are applied. Consequently, all memory accesses follow the strictest policy, leading to a significant number of `trap` events.

3.2 Incompatible Instruction Behavior

As described in Section 2.3, ARMv7-M and ARMv8-A adopt different instruction sets. Some instructions on ARMv7-M are not available on ARMv8-A or within its KVM environment, hindering the correct rehosting of MCU firmware. These instructions can be broadly categorized into two types. On the one hand, certain instructions rely on special registers unique to ARMv7-M, such as the Interrupt Program Status Register, Base Priority Register, and CONTROL register, which are essential for firmware execution. For example, the CONTROL register defines a task's privilege level. It is crucial for managing the execution of both unprivileged and privileged threads, especially in real-time operating system (RTOS)-based MCU firmware. Without proper support or emulation, the firmware may fail to enforce privilege separation, resulting in security vulnerabilities or improper execution of system calls.

On the other hand, certain memory access instructions that utilize post-indexed addressing are not supported in KVM. Post-indexed addressing is commonly used for efficient sequential memory access, such as traversing arrays. It consists of two sub-operations: address calculation and result write-back. For example, when executing the `ldr r2, [r1], #4` instruction, the CPU first uses the value in r1 (a base address register) as the memory address to load data into r2, and then updates r1 by adding 4. Post-indexed operations are also used for MMIO regions on MCUs, such as configuring interrupt priorities in the System Control Space (SCS), a special MMIO region. However, MMIO accesses in KVM typically trigger a `trap` to the hypervisor for further handling. To ensure simplicity and performance, KVM supports only basic addressing modes without write-back for MMIO. Consequently, the base register cannot be updated synchronously, potentially causing inconsistencies in the MCU state and breaking the accuracy of firmware rehosting.

3.3 Incompatible Interrupt Behavior Handling

Correctly capturing and handling asynchronous events via the interrupt controller is essential for MCU firmware execution, as many peripheral operations, such as timers, serial communication, and sensor inputs, rely on precise and timely interrupt responses. However, a fundamental difference exists between the interrupt handling mechanisms of ARMv7-M and

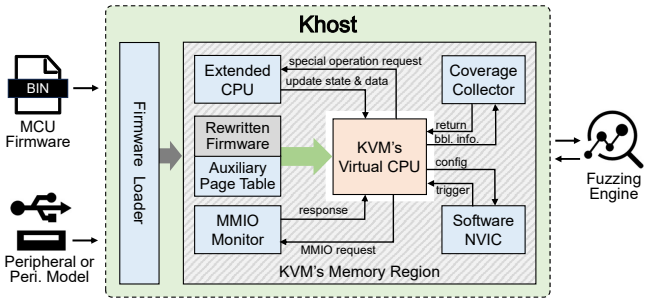

Figure 2: The overview of Khost

ARMv8-A. Specifically, ARMv7-M adopts the NVIC, which assigns a dedicated vector address to each interrupt source, enabling direct and low-latency dispatch to the corresponding handler. In contrast, ARMv8-A utilizes the Generic Interrupt Controller (GIC), which delivers all interrupts through a shared entry point, requiring software to identify and dispatch each interrupt. This discrepancy renders ARMv8-A inherently incompatible with the fine-grained and asynchronous interrupt handling typically expected by MCU firmware.

3.4 Lacking Fuzzing Support

Fuzzing is one of the most common downstream applications of firmware rehosting. However, no suitable system-level mechanisms currently exist on ARM to support MCU firmware fuzzing on KVM. One major limitation is the collection of code coverage, which is essential for guiding input generation and improving fuzzing efficiency. Existing frameworks typically rely on lightweight IR-based instrumentation (e.g., AFL-QEMU [70]) or hardware-assisted tracing. Yet, these techniques are hard to apply to MCU firmware under ARM KVM due to native execution and limited tracing support. Recent works (e.g., μAFL [37], GDBFuzz [17]) leverage debugging infrastructures, but frequent context switches and intrusive breakpoint handling impose substantial overhead.

In addition, MCU firmware frequently receives inputs through peripherals, which are typically used by fuzzing tools to deliver test inputs. Under KVM, each peripheral interaction triggers a `trap` operation, introducing substantial overhead. When combined with the cost of code coverage collection, the overall performance impact becomes significant. This severely limits the practicality of existing approaches in large-scale fuzzing campaigns or time-constrained testing scenarios, where high throughput and low latency are essential.

4 Design

4.1 Overview

To bridge the gap between ARMv7-M and ARMv8-A, and efficiently run MCU firmware on ARMv8-A platforms via KVM, we design a novel rehosting framework called Khost. As shown in Fig 2, there are six components in Khost. First, the Firmware Loader is responsible for loading the firmware, initializing other components, and setting up the execution environment for rehosting and fuzzing. Second, to address challenges in memory access handling, Khost allocates Auxiliary Page Table (**APT**), which is combined with firmware, ensuring proper memory isolation and attribute configuration required by KVM. Third, the extended CPU (**eCPU**) is used to handle MCU-specific operations, helping the KVM's virtual CPU (vCPU) to maintain the system state. Furthermore, the software NVIC is used to capture and handle asynchronous events during firmware execution. Finally, to enable efficient fuzzing for MCU firmware under KVM, we design the MMIO Monitor and the Coverage Collector for Khost. The former manages MMIO access in a flexible and scalable manner, allowing users to configure custom software peripherals (e.g., those used in QEMU) or MMIO interface models (e.g., those generated by Fuzzware). The latter is designed to collect coverage during fuzzing with Khost.

To mitigate the overhead caused by frequent `trap` operations during rehosting or fuzzing in the KVM environment, all components of Khost, except the Firmware Loader, are placed in KVM memory regions mapped to unused high-address spaces reserved in the firmware. This mapping enables the firmware running on the KVM's vCPU to access rehosting components directly via system calls, thereby significantly reducing the performance overhead compared to `traps`.

4.2 Auxiliary Page Table

MMU-based memory isolation and attribute configuration are essential when using KVM to rehost a firmware. However, MCU firmware typically operates based on physical addresses, lacking an MMU. This fundamental difference introduces an inherent incompatibility with KVM. As a result, all memory regions of firmware are assigned as Device-nGnRnE attributes, rendering conventional MCU firmware rehosting on KVM impractical. Therefore, we design APT to address this issue by constructing a virtual MMU for the MCU firmware and correctly configuring the memory attributes.

APT Allocation is a critical step. Specifically, we design the APT Allocator, a part of Firmware Loader used to complete this task. Figure 3 illustrates the allocation process of the APT. Before loading the firmware into KVM's memory, the allocator creates a virtual Stage 1 MMU by allocating the APT, emulating an address translation layer for MCU firmware. This virtual MMU abstracts the firmware's physical address as GVAs and establishes direct mappings from GVAs to IPAs. Each GVA is mapped to an IPA with a zero offset, thereby preserving the physical address layout expected by the firmware. A user can configure the memory attributes on demand. Otherwise, the Allocator applies a default configuration following the memory attribute guidelines

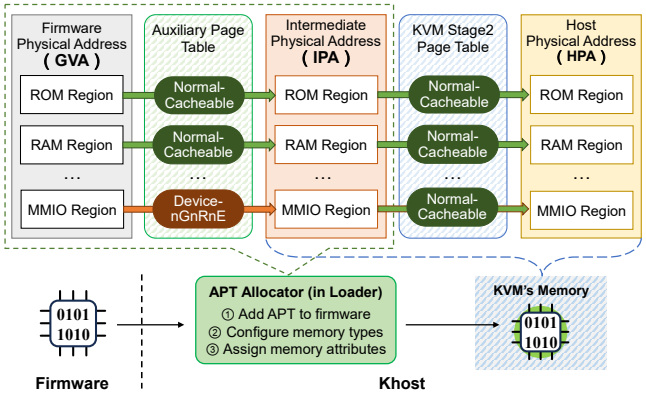

Figure 3: The allocation process of the Auxiliary Page Table

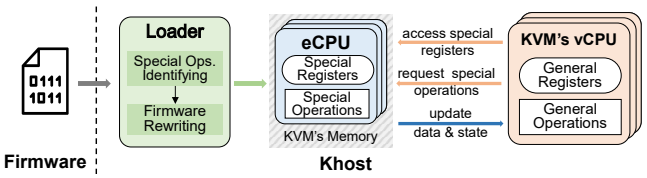

Figure 4: The workflow of the Extended CPU

from the ARM official document [1]. Specifically, the ranges `0x00000000`∼`0x1FFFFFFF` and `0x20000000`∼`0x3FFFFFFF` correspond to ROM / Flash and on-chip SRAM, which are marked as Normal-cacheable memory. The ranges `0x40000000`∼`0x5FFFFFFF` and `0xE0000000`∼`0xE00FFFFF` denote MMIO regions for peripherals and the SCS, respectively, which are assigned the Device-nGnRnE attribute.

A specific configuration of MMU based on APT is essential for other components of Khost. In non-MCU firmware, the Stage 1 MMU maintained by the firmware assigns readable and writable attributes to MMIO regions. When rehosted under KVM, these regions will not be configured in the Stage 2 MMU, so any MMIO access triggers a trap to the hypervisor for handling. This frequent trapping introduces substantial performance overhead. To mitigate this, Khost adopts an optimized strategy. During the allocation of APT, the MMIO region is marked as Device memory without read or write permissions in the Stage 1 MMU. Meanwhile, the MMIO region is registered as Normal memory when configuring the Stage 2 MMU of KVM. Consequently, when the firmware accesses the MMIO region, a data abort exception can be triggered, rather than a trap operation. This exception is then handled in the data abort handler of firmware, allowing Khost to intercept and emulate peripheral behaviors through APT with minimal overhead.

4.3 The Extended CPU

The difference in instruction sets hinders the rehosting of MCU firmware on ARMv8-A with KVM. To settle this issue, we design the eCPU in Khost. As shown in Figure 4, after initialization, the eCPU is loaded into the memory of KVM. It intercepts vCPU requests for some operations unsupported by ARMv8-A or KVM, emulates the corresponding hardware behaviors, and updates the relevant state and data accordingly. These operations include: (1) access to special registers in the MCU; and (2) post-indexed operations on MMIO.

Access to special registers unsupported by ARMv8-A causes undefined behavior or system-level faults that may immediately crash the VM of KVM. Such operations can not be captured by the exception handling mechanism of KVM, obstructing Khost from handling them directly. Fortunately, these registers can only be accessed via the `msr`, `mrs`, and `cps` instructions [1]. Consequently, Khost handles MCU-specific register accesses by the following steps. First, it utilizes the Firmware Loader to statically identify instructions that access these registers by analyzing the bytecode[2]. For example, given a 4-byte Thumb instruction, if the upper 2 bytes, after a bitwise AND with `0b1111111111110000`, equal to `0b1111001110000000`, and the lower 2 bytes, after a bitwise AND with `0b1111001100000000`, equal to `0b1000000000000000`, the instruction can be recognized as an `msr` instruction. And the bits [3:7] of the lower two bytes encode the ID of the special register. Next, it extracts some essential information to initialize the eCPU, including instruction addresses and register indices, then rewrites the original instructions as `svc` instructions. During execution, when the firmware attempts to access a special register, KVM saves the current context of the vCPU and triggers a supervisor call. The eCPU intercepts this call with the help of APT, handles the operation like hardware, and updates the corresponding state and data to the vCPU.

Post-indexed operations on MMIO are unsupported in KVM, which impedes the correctness of firmware rehosting. As mentioned in Section 4.2, with the special configuration for MMU, each MMIO access can trigger a data abort exception. Therefore, the eCPU first intercepts the data abort exception and identifies MMIO access by parsing the values in the DFSR (Data Fault Status Register) and DFAR (Data Fault Address Register). Specifically, the DFSR indicates the cause of the exception, while the DFAR records the address accessed by the firmware when an exception happens. If the access is determined to be an access to MMIO, the eCPU notifies the MMIO Monitor (see Section 4.4) to handle it, then updates the results to the vCPU. Otherwise, the original data abort handling routine of the firmware is executed.

To handle MCU-specific operations, directly using the virtual CPU structure from the QEMU framework is suboptimal. This is mainly because QEMU is designed to emulate a wide range of processor functionalities, including the full regis-

[2]Capstone may misidentify some MCU-specific instructions (https://github.com/capstone-engine/capstone/issues/2418), so we do not use it in this stage.

ter state and additional data structures tailored for dynamic binary translation. These components are tightly coupled, which hinders the CPU structure from being loaded into the memory of KVM. By contrast, the eCPU in Khost focuses on cooperating with the vCPU of KVM to maintain the system state of the MCU. It adopts a lightweight design and is solely responsible for emulating a small subset of system registers dedicated to ARMv7-M, while the majority of registers are managed by KVM, thereby introducing minimal overhead in both execution time and memory consumption.

4.4 Interrupt Handling

The GIC in ARMv8-A systems exhibits fundamental differences from the NVIC utilized in MCUs, particularly in terms of structural organization, interrupt prioritization, and delivery semantics. These architectural disparities render the GIC inherently incompatible with the NVIC event-driven interrupt model and preclude native support for the asynchronous, low-latency interrupt semantics expected by firmware originally designed for NVIC-based systems.

To address this challenge, we implement a software-based NVIC in Khost. During firmware initialization, it configures interrupt priorities and enables specific interrupt lines by emulating writes to control registers. When an interrupt is triggered during execution, either by fuzzing input or simulated peripheral events, the NVIC checks whether the interrupt is both pending and enabled. If it is, the NVIC retrieves the corresponding handler address from the vector table, stacks the current context, and updates the program counter to perform exception entry. This design faithfully emulates the interrupt-driven execution model of MCU hardware, enabling accurate behavior reproduction and supporting downstream analyses such as fuzzing and debugging.

It is critical to accurately identify which interrupts are valid for firmware running and to understand when an interrupt is required by the firmware. First, certain firmware has vendor-provided default interrupt handlers, many of which are implemented as infinite loops intended to halt execution when unexpected events occur. If a round-robin strategy is used to trigger interrupts during fuzzing, these default handlers may be inadvertently invoked, causing the firmware to hang or stall. Thus, to mitigate this issue, Khost's NVIC inspects each handler's instructions before triggering an interrupt. Specifically, if a handler contains the `0xFEE7` opcode (commonly representing an infinite loop) and its total instruction count is fewer than five, the NVIC suppresses the triggering of that interrupt to prevent unintended stalls. Second, it is necessary to fire an interrupt under certain conditions, such as when the firmware enters an idle state by executing a `wfi` instruction. Instead of simply replacing `wfi` with a `nop` instruction as done by Fuzzware, Khost's NVIC detects such situations and automatically injects an interrupt to wake the processor, thereby mimicking realistic hardware behaviors. These strategies significantly enhance the stability of dynamic analysis and avoid unnecessary exploration of firmware behaviors.

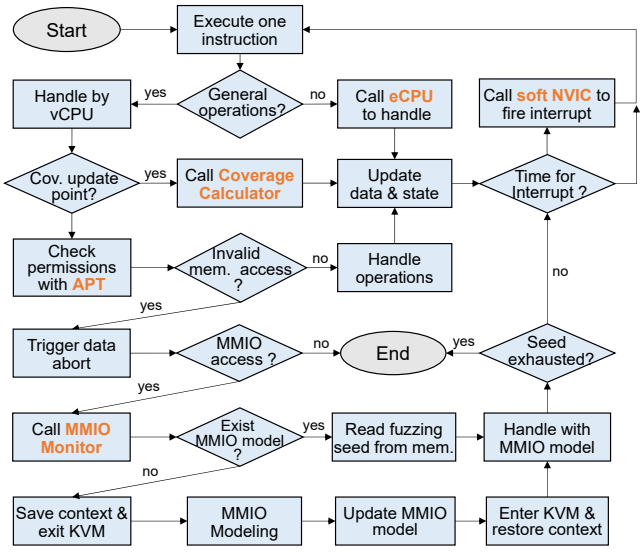

Figure 5: The workflow of fuzzing one test case with Khost.

4.5 Fuzzing with Khost

Khost is a flexible and scalable framework for MCU firmware rehosting, supporting downstream tasks such as fuzzing and dynamic debugging. To demonstrate its capability in facilitating downstream tasks, we primarily showcase its application in firmware fuzzing, one of the most prevalent and significant applications of rehosting. In Section 3.4, we have discussed that coverage collection and peripheral interactions can introduce significant overhead under a KVM environment, limiting the feasibility of large-scale fuzzing for MCU firmware. We design a fuzzing wrapper in Khost's Firmware Loader to address this performance bottleneck. The fuzzing wrapper automatically configures the fuzzer, rewrites the firmware to enable coverage collection, and encapsulates it with all rehosting components. Figure 5 illustrates the workflow of fuzzing one test case in Khost.

Coverage Collection based on heavyweight binary rewriting, as employed in SafireFuzz [52], often disrupts the firmware's original address space layout, which can lead to unintended execution behavior. To collect coverage while preserving the firmware's space address layout, we utilize the following method. Before loading the firmware into KVM, the wrapper first extracts essential metadata from the firmware, including the length of the basic block and the type of instructions within each block. Then, it analyzes each basic block to identify the instructions that do not alter the program counter (PC). Finally, it replaces either a 4-byte instruction or a pair of consecutive 2-byte instructions with a 4-byte unconditional branch instruction jumping to the Coverage Calculator. Simi-

lar to prior work [26], Khost doesn't handle table branches, short blocks, or instructions operating on PC-relative data to preserve the original execution semantics. According to the statistics in Section 6, these cases only comprise 6.9%∼12.1% of all basic blocks in the dataset, so their impact on coverage collection is expected to be minor.

The Coverage Calculator consists of a series of code trampolines placed in an unused region of the firmware's high-address space, with each trampoline corresponding to a specific basic block. When the control flow is redirected from the coverage update point of the firmware to the trampoline. The trampoline first saves firmware context according to the ARM calling conventions and calculates the coverage data. Then, it updates the coverage bitmap shared between Khost and the fuzzing engine. Next, it re-executes the original instruction(s) that are replaced by the branch instruction. Finally, it recovers the control flow of the firmware.

Peripheral interaction is also critical for firmware rehosting, and numerous works [21, 42, 50, 68, 74] have been proposed to tackle this challenge. Specifically, we implement an MMIO Monitor, which can handle MMIO access from firmware by configuring the existing peripheral modeling approach flexibly. During rehosting, it loads the peripheral model configuration file generated by the modeling module (initially empty) and parses it into memory. Upon the firmware accessing an MMIO region during execution, it can respond according to the loaded model. If the accessed address is not yet modeled, the monitor dynamically invokes the modeling module to generate and update the peripheral model.

In KVM, intensive peripheral interactions trigger numerous trap operations, leading to substantial runtime overhead. Khost employs a new strategy to mitigate this issue. As described in Section 4.3, eCPU intercepts MMIO accesses from the firmware and forwards them directly to MMIO Monitor for further handling. By replacing trap operations with direct firmware-level exception handling, this strategy substantially reduces the overhead of peripheral interactions during firmware rehosting.

5 Implementation

We implement the prototype of Khost on the ARM Linux platform. The framework comprises approximately 15,000 lines of code written in C, C++, Python, Rust, and assembly. The key implementation details are as follows.

Khost identifies the base address and entry point of the firmware by integrating the vector-based algorithm from IDA-Pro [25]. To extract essential information and facilitate lightweight rewriting for firmware, Khost integrates Capstone [48] and Keystone [49]. To enable fuzzing under KVM, Khost rewrites the firmware to insert instrumentation code following the approach proposed in SURGEON [26], and configures AFL++ as the default fuzzing engine.

To handle peripheral accesses during rehosting, Khost's MMIO Monitor integrates the peripheral modeling technique introduced in Fuzzware [50], which has demonstrated excellent performance in dealing with peripheral access during rehosting. Since Fuzzware's modeling approach requires a state snapshot as input to Angr for generating and updating MMIO models, we implement a transformer that constructs such snapshots from the KVM's VM context whenever an MMIO access occurs.

6 Evaluation

In this section, we demonstrate the effectiveness of Khost by answering the following questions.

RQ1: How does the performance of our tool compare with existing firmware rehosting solutions? (6.2)

RQ2: What are the respective performance improvements brought by APT and loading rehosting components into KVM? (6.3)

RQ3: How does Khost perform compared to previous methods in fuzzing MCU firmware? (6.4)

6.1 Experiment Settings

Environments. We conduct our experiments on three different platforms: (1) `KunPeng Server`, an ARM server equipped with two 32-core Cortex-A72 processors clocked at up to 2.4 GHz, and 256 GB of DDR4 RAM running at 2400 MT/s. It serves as our primary evaluation platform. (2) `Dell Server`: an x86 server featuring four 24-core Intel® Xeon® Platinum 8260 processors running at up to 3.9 GHz, and 128 GB of DDR4 RAM at 2933 MT/s. It is used to evaluate the cross-architecture rehosting capabilities of QEMU. (3) `Raspberry Pi 4B Board`, an ARM development board with a quad-core Cortex-A72 processor clocked at up to 1.8 GHz, and up to 8 GB of LPDDR4 RAM at 3200 MT/s. It is used to demonstrate the lightweight deployment capability of Khost and to evaluate its effectiveness on a lower-power ARM device. All platforms run a 64-bit Ubuntu 22.04 LTS OS to ensure a consistent experiment environment.

Dataset. To evaluate the performance of Khost in both rehosting and fuzzing, we collect three distinct datasets. To answer **RQ1** and **RQ2**, we conduct experiments using test cases from the following benchmarks: (1) `CoreMark-PRO` [13], an industry-standard benchmark designed to measure system performance on complex computational workloads. Each benchmark test case is compiled into two variants. One for Cortex-M4 firmware using STM32CubeIDE [54], which integrates an arm-none-eabi-gcc toolchain. The other is compiled into an ARM Linux application using the arm-linux-gnueabi-gcc toolchain, serving as a reference to analyze runtime overhead. Table 7 (in Appendix) summarizes key information about this benchmark. (2) `Simbench` [59], a comprehensive micro-benchmark suite for system-level performance evaluation of

Table 2: The basic information of real-world firmware.

Sources	Firmware Name	MCU Type
HALucinator	atmel_6lowpan_rx	Atmel-Asfv3
	tcp_echo_client	STM32F4
	tcp_echo_server	FRDM-K64F
P^2IM	PLC	STM32F429ZI
	Drone	STM32F103RB
	Heat-Press	SAM3X8E
SafireFuzz	Stm32_Sine	STM32F103RB
	LibJPEG_decoding	STM32F429NIH
New	BlueMicro_BLE† [30]	NRF52832
	Avem [3]	STM32F103RET
	Zephyr-Modbus† [71]	FRDM-K64F
	nuttx-nsh† [45]	NRF52840

† RTOS-like firmware.

full-system emulators. Given that this dataset mainly covers system-level tasks tailored to specific ARM development boards, we do not construct a corresponding ARM Linux application variant. Basic information about the selected test cases is provided in Table 8 (in Appendix).

In addition, to answer **RQ3**, we need a real-world MCU firmware dataset. We initially plan to construct the dataset using all 12 real-world firmware used by SafireFuzz. However, we observe that four cases have the same MCU or exhibit similar behaviors with others (e.g., 6LoWPAN_Receiver and 6LoWPAN_Transmitter). Therefore, we exclude these four cases to make the dataset more representative. Further, to increase dataset diversity, we add four new real-world firmware from GitHub, which use different MCUs, and include three more complex RTOS-like firmware. Table 2 provides detailed information about this dataset.

6.2 Rehosting Performance

To evaluate the rehosting performance, we select QEMU as the primary baseline. On the one hand, as shown in Table 1, most existing firmware rehosting works are built on QEMU. On the other hand, it is infeasible to recompile firmware for native execution on ARMv8-A platforms due to MCU-specific operations (e.g., accessing PRIMASK/CONTROL registers). As a remediation, we replace these operations in benchmarks with equivalent ones on Linux and recompile them to measure the native execution speed. This helps us understand to what extent the other baselines differ from near-native execution.

To assess the individual contributions of APT and loading rehosting components into KVM, we construct two variants of Khost: one with APT disabled (Khost-NoAPT), and another with rehosting components loaded outside of KVM (Khost-OUT). All performance experiments are conducted by running each test three times on a single CPU core, with the average of the runs reported as the final result.

CoreMark-PRO: To evaluate the performance of Khost in handling complex computational tasks, we rehost test cases from the CoreMark-PRO. As shown in Table 3, compared with QEMU running on the same platform, Khost reduces time overhead by at least 90.0%. Additionally, in all test cases except the parser, Khost achieves performance that is nearly equivalent to native execution. The observed overhead in the parser results from differences in the complexity and optimization of memory allocator implementations between ARM-native applications and MCU firmware. On Linux, `malloc` / `free` leverage advanced mechanisms such as thread-local caching (tcache) and multiple memory arenas, ensuring highly efficient memory operations. In contrast, the MCU firmware relies on simpler `malloc_r` / `free_r` implementations that introduce global locking to ensure thread safety. As a result, each allocation or deallocation incurs significant synchronization overhead. Given that the parser heavily depends on dynamic memory allocation, this discrepancy results in a performance overhead exceeding 1x.

Remarkably, while QEMU running on a higher-performance x86 server also achieves reduced overhead, Khost further decreases overhead by an additional 16.6% to 30.8% compared to it. We further evaluate Khost on the Raspberry Pi 4B board. The results show that, despite running on a substantially less powerful platform, Khost achieves at least an 85.0% reduction in overhead.

Simbench: To better understand the performance of Khost in various system-level tasks, we do experiments on the Simbench. Table 4 presents the final results. We can see that Khost outperforms QEMU running on the same server in all cases of Memory, I/O, Control Flow, and Code Generation. Especially, Khost reduces overhead over 90% in mem-cold, coprocessor, inter-page-direct, intra-page-indirect, and small-blocks. However, in Exception-related tasks, Khost introduces roughly 10% overhead. This overhead primarily arises from Khost's modifications to the original exception handling logic via the eCPU and MMIO Monitor, which are necessary to intercept operations that KVM cannot handle. Fortunately, most of them occur infrequently during normal firmware execution.

Compared with QEMU on the Dell server, Khost outperforms in all system-level tasks, except for Exception cases and the mem-hot case. For Exception cases, the overhead also stems from Khost's modifications to the original handler. For mem-hot, QEMU gains an edge from both the Dell server's stronger hardware and its internal TLB-based optimizations for memory-intensive operations. Moreover, further experiments on the Raspberry Pi board demonstrate that Khost cuts down overhead up to 98.1% compared to QEMU on the KunPeng server, further demonstrating Khost's efficiency in MCU firmware rehosting.

Table 3: Execution time (in seconds) of running CoreMark-PRO test cases. Khost-NoAPT: rehosting firmware on Khost with APT disabled; Values in parentheses except Khost-NoAPT indicate the percentage relative to the QEMU running on KunPeng Server.

Test cases	KunPeng Server				Dell Server	Raspberry Board
	QEMU	Khost	Khost-NoAPT	Native	QEMU	Khost
cjpeg	31.555	**2.155 (-93.2%)**	243.827 (+112.1x)	2.129 (-93.3%)	10.166 (-67.8%)	2.541 (-91.9%)
core	114.514	**6.228 (-94.6%)**	804.825 (+128.2x)	5.002 (-95.6%)	37.535 (-67.2%)	8.085 (-92.9%)
liner	73.388	**7.305 (-90.0%)**	1013.403 (+137.7x)	6.346 (-91.4%)	28.627 (-61.0%)	9.533 (-87.0%)
nnet	119.623	**7.251 (-93.9%)**	355.451 (+48.0x)	6.295 (-94.7%)	40.031 (-66.5%)	9.384 (-92.2%)
parser	152.491	**14.932 (-90.2%)**	974.565 (+64.3x)	7.346 (-95.2%)	55.020 (-63.9%)	22.923 (-85.0%)
radix	115.505	**5.289 (-95.4%)**	274.636 (+50.9x)	5.135 (-95.6%)	40.810 (-64.7%)	7.277 (-93.7%)
sha	371.892	**16.865 (-95.5%)**	1994.610 (+117.3x)	16.265 (-95.6%)	78.501 (-78.9%)	22.228 (-94.0%)
zip	29.169	**2.908 (-90.0%)**	291.191 (+99.1x)	2.497 (-91.4%)	9.522 (-67.4%)	3.608 (-87.6%)

6.3 Ablation Study

As mentioned in Section 3.1, due to the absence of the MMU in MCU firmware, KVM is unable to properly configure and manage memory, resulting in all memory regions being marked as Device-nGnRnE, which severely degrades rehosting performance. Khost addresses this issue by introducing APT. In addition, frequent exits from KVM during rehosting introduce significant overhead due to context switching. To mitigate this, we adopt a strategy where eCPU and MMIO Monitor are loaded into the KVM address space, enabling inter-module interactions via system calls instead of traps. To better understand the impact of these two optimizations, we evaluate the performance using those two invariants: Khost-NoAPT and Khost-OUT. The gray-shaded columns of Table 3 and Table 4 record the results.

CoreMark-PRO: After disabling APT, the rehosting overhead of Khost increases by at least 48.0× on the nnet case, with four out of eight test cases exhibiting slowdowns of over 100×. For the parser test case, the overhead reaches as high as 137.7×. These results underscore the critical role of APT in preserving both capability and efficiency when rehosting MCU firmware on KVM. Since this dataset is specifically designed for complex computational tasks, and special register accesses and device interactions are rare, we do not use it to evaluate Khost-OUT.

Simbench: Disabling APT significantly increases rehosting overhead, especially for inter-page direct calls and small-block code generation, with overheads rising over 140× and 179×, respectively. These results highlight the critical role of APT in system-level tasks. With the special configuration of APT, any privileged operation, such as MMIO, triggers a Stage 1 MMU exception without exiting from KVM. To forward such accesses outside KVM, we insert the `bkpt` instruction. As shown in Table 4, moving rehosting components outside KVM introduces a 35.8× overhead for device-access and over 200× for exception handling. Compared to exits caused by Stage 2 MMU aborts, exits triggered by `bkpt` incur lower overhead on device-access, since fewer registers and system state need to be saved. However, exception handling, particularly data aborts, remains costly due to the additional processing required during KVM exits. These results highlight the importance of moving rehosting components directly within KVM to minimize overhead and maintain efficient system-level execution.

In total, the ablation experiment effectively illustrates the significant contribution of APT and the strategy to load rehosting components inside KVM.

6.4 Fuzzing Experiments

To evaluate the fuzzing performance of Khost, we compare it against three representative tools: (1) `HALucinator`, a Unicorn[3]-based fuzzer utilizing HLE; (2) `Fuzzware`, a Unicorn-based fuzzer with automated peripheral modeling; and (3) `SafireFuzz`, which combines binary rewriting with HLE. SURGEON is not included in the evaluation since its performance is worse than Fuzzware according to the paper [26], particularly in terms of basic block coverage.

As HALucinator and Fuzzware are designed to run on the x86 platform, we extend them to support running on the ARM platform by: (1) adding memory access interception to QEMU on AArch64 to capture peripheral operations[4], and (2) modifying the toolchain to build native libraries for ARM to accelerate rehosting. As both tools rely on AFL/AFL++ fuzzing backends, we configure them with AFL++ (v4.32) and use the same backend for Khost, ensuring a fair comparison. To ensure completeness and provide a reference, we also conduct fuzzing experiments with HALucinator and Fuzzware on the Dell server, with detailed results presented in

[3]A QEMU derivative that uses the same translation-based emulation.

[4]QEMU inlines memory read/writes on AArch64 platform, which causes the memory hook to fail, obstructing the capture of peripheral operation events:https://github.com/unicorn-engine/unicorn/issues/1737

Table 4: Execution time (in seconds) for Simbench test cases. Khost-OUT represents the results obtained when moving rehosting components outside of KVM. The gray-shaded columns are used for ablation evaluation. The values in parentheses represent percentages relative to QEMU running on KunPeng Server, except gray-shaded columns.

Type	Test cases	KunPeng Server				Dell Server	Raspberry Board
		QEMU	Khost	Khost-NoAPT	Khost-OUT	QEMU	Khost
Mem.	mem-hot	32.74	**14.50 (-55.7%)**	1220.71 (+83.2x)	14.61 (+0x)	11.03 (-66.3%)	19.36 (-40.9%)
	mem-cold	685.84	**10.36 (-98.5%)**	239.05 (+22.1x)	10.42 (+0x)	345.20 (-49.7%)	29.01 (-95.8%)
I/O	co-processor	109.45	**3.17 (-97.1%)**	292.83 (+91.4x)	3.17 (+0x)	47.15 (-56.9%)	4.18 (-96.2%)
	device-access	23.48	**10.07 (-57.1%)**	397.52 (+38.5x)	370.50 (+35.8x)	10.91 (-53.5%)	18.19 (-22.5%)
Except.	dfault	145.36	**134.98 (-7.1%)**	241.22 (+0.8x)	30974.33 (+228.5x)	64.24 (-55.8%)	190.20 (+30.8%)
	ifault	128.26	**134.01 (+4.5%)**	13532.83 (+100.0x)	29309.95 (+217.7x)	57.28 (-55.3%)	191.16 (+49.0%)
	syscall	239.58	**265.72 (+10.9%)**	27400.59 (+102.1x)	59759.00 (+223.9x)	109.10 (-54.5%)	377.25 (+57.5%)
	undef	235.74	**266.45 (+13.0%)**	27277.27 (+101.4x)	53718.35 (+200.6x)	108.42 (-54.0%)	377.44 (+60.1%)
Contr. Flow	inter-page-direct	596.25	**10.85 (-98.2%)**	1535.12 (+140.5x)	10.86 (+0x)	215.60 (-63.8%)	14.49 (-97.6%)
	inter-page-indire.	117.80	**21.27 (-81.9%)**	566.18 (+25.6x)	21.43 (+0x)	48.94 (-58.5%)	29.30 (-75.1%)
	intra-page-direct	77.10	**10.85 (-85.9%)**	1009.63 (+92.0x)	10.86 (+0x)	32.50 (-57.8%)	14.49 (-81.2%)
	intra-page-indire.	174.37	**14.11 (-91.9%)**	629.25 (+43.6x)	14.11 (+0x)	103.70 (-40.5%)	18.83 (-89.2%)
Code Gen.	large-blocks	32.80	**7.10 (-78.4%)**	233.10 (+30.4x)	7.10 (+0x)	15.26 (-53.5%)	9.47 (-71.1%)
	small-blocks	354.80	**5.15 (-98.5%)**	923.30 (+179.3x)	5.16 (+0x)	158.22 (-55.4%)	6.88 (-98.1%)

Section B (in Appendix). In contrast, SafireFuzz is tightly coupled with LibAFL (v0.71), so we adopt LibAFL for it and similarly configure Khost with the same fuzzing engine in our experiments to ensure a fair and direct comparison. Besides, we use four files as initial seeds, which contain 512 bytes of zeros, 512 bytes of ones, a 512-byte shifting-one bit, and an empty file, respectively.

As discussed earlier, certain operations replaced by HLE-based approaches are essential for the correct execution and rehosting of firmware. Therefore, we count the number of basic blocks reached by Fuzzware and Khost during the experiments, including those substituted by HLE-based methods. For all fuzzing experiments, we perform five 24-hour runs with each process pinned to a designated core, and report the average result.

6.4.1 Throughput & Coverage

We first analyze the throughput and basic block coverage during fuzzing, with the overall results summarized in Table 5 and the coverage over time visualized in Figure 6.

Khost vs. HALucinator. We use the default harnesses provided by HALucinator, but four firmware blobs fail to rehost. For the remaining eight firmware, Khost achieves significantly higher fuzzing throughput under identical conditions, with speedups of up to 197.5× compared to HALucinator (atmel_6lowpan_rx). In terms of basic block coverage, Khost outperforms HALucinator on all evaluated targets, achieving an increase in coverage ranging from 41% to 602% (24% to 418% after excluding the basic blocks in HALs). Conducting a Mann–Whitney U test on execution speed and coverage further confirms that the observed differences are statistically significant across all targets ($p < 0.05$).

Khost vs. Fuzzware. Fuzzware and Khost can successfully rehost and fuzz all firmware. Compared to Fuzzware, Khost achieves up to a 68.6× improvement in fuzzing throughput (nuttx-nsh), while improving basic block coverage by 1% to 335% . For tcp_echo_client and tcp_echo_server, Khost exhibits lower throughput than Fuzzware but achieves up to a 335% increase in coverage. This is due to the limitation of Fuzzware's NVIC controller, which cannot bypass invalid interrupts that can lead to a firmware hang. It causes the execution to prematurely hit the exit condition threshold (i.e., up to 150,000 basic blocks without any MMIO access). In contrast, Khost can automatically skip such interrupts during fuzzing, allowing it to explore more than four times as many basic blocks in these cases. In addition, as shown in Figure 6, the higher throughput of Khost enables it to achieve faster coverage improvement and cover more basic blocks on most firmware. We also manually analyze the peripheral models automatically generated during fuzzing. The results show that the higher throughput allows Khost to discover more peripherals. For example, on Zephyr-Modbus, Khost generates three additional models compared to Fuzzware.

Khost vs. SafireFuzz. As shown in Table 2, five firmware could not be rehosted via SafireFuzz's binary rewriting approach, including one (atmel_6lowpan_rx) that is originally used in its evaluation. Khost achieves higher coverage than SafireFuzz across all firmware (a 16%∼349% improvement

Table 5: Basic block coverage (BBL) and throughput (measured as the number of test cases executed per second, Exec/s) during fuzzing on the KunPeng server. '-' indicates that the firmware could not be rehosted. Khost-OUT represents the results obtained when moving rehosting components outside of KVM. Gray-shaded columns denote the results obtained using LibAFL. The number on the left of '/' shows total coverage, while the number on the right shows coverage excluding basic blocks in HALs.

		HALucinator		**Fuzzware**		**Khost**		**Khost-OUT**		**SafireFuzz**		**Khost-LibAFL**	
FW.	BBLs	Exec/s	BBL	Exec/s	BBL	Exec/s	BBL	Exec/s	BBL	Exec/s	BBL	Exec/s	BBL
atmel.	6706	1.4	1476	58.7	2398 / 2086	274.7	3154 / 2802	19.2	2288	-	-	422.0	3177 / 2824
.client	7359	3.2	984	307.5	460 / 266	276.7	2000 / 1636	14.4	1029	3194.4	1052	382.4	1974 / 1625
.server	6895	3.9	745	318.1	459 / 266	332.6	1254 / 926	16.7	1072	3652.0	795	528.2	1244 / 925
PLC	2304	2.4	106	56.9	633 / 472	379.4	744 / 549	42.6	656	3744.0	143	726.8	721 / 541
Drone	2726	1.7	248	17.1	582 / 398	231.5	1436 / 1211	10.7	612	2910.0	239	131.2	1101 / 590
Heat.	1832	14.0	201	39.4	552 / 460	258.4	579 / 484	23.6	514	3701.0	201	400.2	571 / 476
.Sine	3518	10.8	416	18.9	437 / 435	1053.9	1364 / 1328	13.9	907	4163.6	416	1528.6	1520 / 1463
.JPEG.	5345	6.7	617	167.6	756 / 666	536.2	873 / 777	16.0	799	4241.2	174	917.4	879 / 782
Blue.	9475	-	-	32.0	733	157.0	762	11.3	754	-	-	116.4	762
Avem	1248	-	-	35.6	909	1080.5	916	33.4	896	-	-	1198.4	914
Zephyr.	8755	-	-	52.2	2037	118.9	2119	4.0	1454	-	-	160.6	2118
nuttx.	8986	-	-	21.2	2097	1473.2	2350	11.8	2044	-	-	868.0	2511

after excluding basic blocks in HALs), despite its lower throughput. The main reason is that SafireFuzz hooks numerous functions with `nop/return_zero` stubs, skipping costly error-handling and validation paths. While this boosts execution speed, it introduces semantic discrepancies that compromise the fidelity and behavioral accuracy of rehosting. This highlights a key limitation of rewriting and HLE-based methods: although manual approximations can improve throughput, they often do so at the expense of semantic correctness.

Ablation Study on Fuzzing. As discussed in Section 6.3, disabling APT incurs prohibitive overhead on system-level tasks. Moreover, during fuzzing, firmware I/O operations reduce throughput to impractical levels, rendering this ablation infeasible. Therefore, we only evaluate the impact of relocating rehosting components outside KVM. As shown in the Table 5, the fuzzing throughput drops to 4.0∼33.4 exec/s, even only about 1% of the original in Avem and nuttx-nsh, while coverage falls to 43%. These results highlight that employing APT and integrating rehosting components directly within KVM are essential for practical fuzzing. Besides, to quantify the overhead of rewriting-based coverage collection, we design a micro-benchmark based on Drone and measure the average execution time of Khost three times with identical inputs. We compare the results when the coverage instrumentation is enabled and disabled. The results show that the overhead caused by coverage collection is approximately 4.71% (13.15s vs. 13.77s).

Overall, compared to QEMU-based fuzzing frameworks, Khost significantly improves fuzzing throughput without sacrificing coverage. Compared to rewriting-based approaches, Khost better preserves MCU semantics, achieving higher coverage while maintaining comparable throughput.

Table 6: Unique crashes and bugs found during fuzzing.

Firmware	**Crashes**	**Total Bugs**	**New Bugs**
atmel_6lowpan_rx	2	1	1
tcp_echo_client	2	1	1
tcp_echo_server	2	1	1
PLC	6	4	0
Heat-Press	1	1	0
LibJPEG_decoding	1	1	1
Zephyr-Modbus	1	0	0
nuttx-nsh	1	1	1
Total	16	10	5

6.4.2 Bug Detection

We collect and deduplicate the crashes detected by Khost. As shown in Table 6, Khost uncovers 16 unique crashes in total, and we confirm 10 cases as real bugs, which include five previously-reported bugs and five new ones. Next, we highlight the noteworthy bugs as follows.

We have discovered some bugs original found by prior works [21, 50, 52]. For example, the bugs in PLC and Heat-Press are a combination of an improper validation of array index (CWE[5]-129) and an out-of-bound write (CWE-787). These bugs allow a remote attacker to overwrite data objects on the embedded device and cause denial-of-service (DoS) or data corruption.

Remarkably, among the five new bugs, four of them orig-

[5]Common Weakness Enumeration: https://cwe.mitre.org/

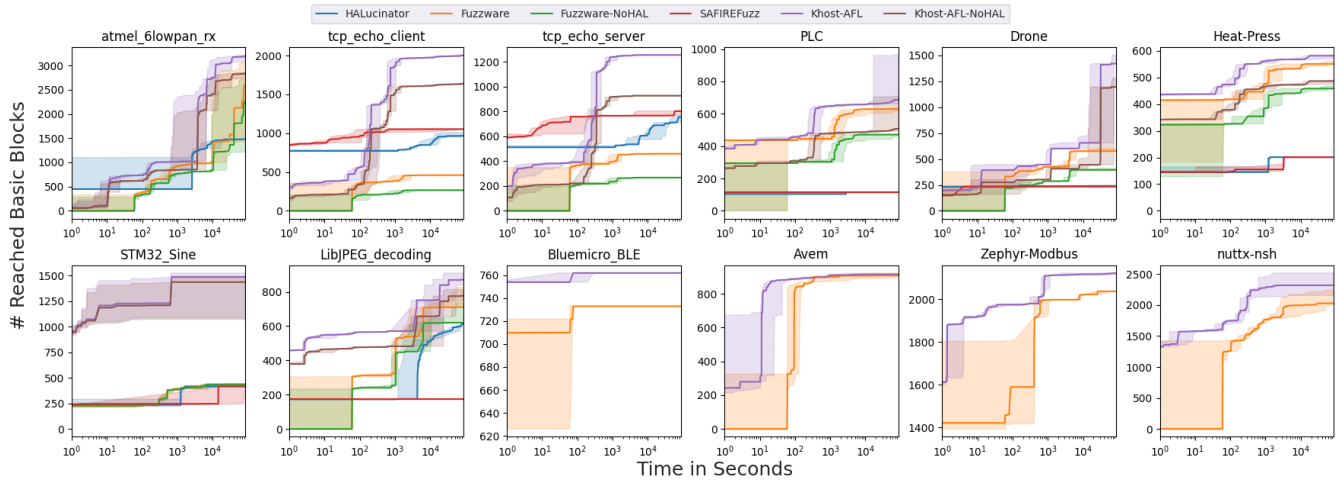

Figure 6: The median and the 95% confidence intervals of coverage over five 24-hour runs for HALucinator, SafireFuzz, Fuzzware, Fuzzware-NoHAL, Khost, and Khost-NoHAL on each test case.

inate from HAL functions in the firmware tcp_echo_client, tcp_echo_server, LibJPEG_decoding, and atmel_6lowpan_rx. For tcp_echo_client, it configures the global pointer of a handler according to the ID read from the device. Since only a small subset of IDs is valid, reading an invalid ID leaves the handler uninitialized, and the initialization function returns an error code. However, in the same parent function that calls the initialization function, a subsequent function invokes this handler without checking the returned code (CWE-252). As a result, it dereferences an uninitialized pointer (CWE-476), which can potentially cause a DoS attack. The bugs found in tcp_echo_server and LibJPEG_decoding are similar in nature. For atmel_6lowpan_rx, it configures the master module based on device input. When getting certain inputs, the module is set to NULL, but the subsequent routine still uses it without verifying the configuration results, leading to a NULL pointer dereference and a potential DoS attack. Existing tools fail to uncover these bugs due to their inherent limitations. Specifically, Fuzzware cannot bypass invalid interrupts that can lead firmware hang. SafireFuzz and HALucinator replace the corresponding HAL functions with manual stubs (most functions are just replaced as nop operations), which conceal the faulty logic and prevent effective analysis.

The fifth new bug is identified in nuttx-nsh, an RTOS-based firmware. It originates from a shell parser that lacks proper logic to filter unexpected input that tries to call up the background. When such input is received, the parser invokes the background, due to its inability to handle the terminal, infinite recursion happens, ultimately causing a stack overflow (CWE-674), leading to potential DoS or Remote code execution. This bug can also be detected by Fuzzware, but it is not discoverable by HALucinator or SafireFuzz, as they fail to rehost this firmware. Although we don't evaluate SURGEON, we think it can not handle the complex operations of this firmware properly according to our analysis on Section 8.

During the bug confirmation, we identify two primary sources of false positives. First, despite Khost optimizes the interrupt triggering mechanisms, some interrupts are still fired before their initialization routines finish. This leads to false positives in firmware such as atmel_6lowpan_rx, tcp_echo_client, tcp_echo_server, and PLC, where interrupt handlers access uninitialized resources. Second, when using Fuzzware's peripheral modeling approach, inaccuracies in device interactions also cause false positives in firmware such as Heat-Press and Zephyr-modbus. These results highlight that precise handling of asynchronous events and accurate peripheral modeling are critical for advancing practical firmware rehosting and fuzzing.

7 Discussion

Co-processor operations. ARM-based MCUs support up to 16 co-processors to extend the main processor's capabilities [26, 46]. Instructions issued by the main CPU can be forwarded to the appropriate co-processor for execution if required. Hardware floating-point support via co-processors requires no special handling, as KVM can directly forward these instructions to the host ARMv8-A's floating-point unit for native execution. Although the ARMv8-A architecture supports backward compatibility with 32-bit execution through the AArch32 instruction set, it lacks native support for some MCU-specific co-processor instructions. These include: ① co-processor data processing (`cdp/cdp2`); ② register transfer instructions between ARM and co-processor (`mcr/mcr2`, `mcrr/mcrr2`, `mrc/mrc2`, `mrrc/mrrc2`); and ③ memory transfer instructions (`ldc/ldc2`, `stc/stc2`). However, we don't observe any use of such co-processor operations in our experiments. If any requirements, one can add

related simulation logic to the eCPU.

Manual Effort. Although Khost can automatically rehost stripped firmware, certain corner cases may require limited user intervention. On one hand, a vector-based method is integrated to locate a firmware's base address and entry point. However, prior work [63] has shown that such a method is not always reliable. On the other hand, IDA-Pro is used for basic block extraction, which achieves both high efficiency (3 seconds per firmware in our experiment) and accuracy (according to existing work [15]). However, some basic blocks may be occasionally missed by its older versions. To handle these cases, Khost features a configuration mechanism to enable users to supply the missing information.

Hardware Platforms. Our insight is to extend KVM to rehost MCU firmware on high-performance platforms. Currently, our prototype is implemented on high-performance platforms equipped with ARMv8-A processors that provide 32-bit compatibility and EL2-level hardware virtualization. To our knowledge, platforms that satisfy the necessary hardware criteria are widely available, including the Raspberry Pi 4B, Honeycomb LX2 workstation, and Huawei Kunpeng servers. Although our prototype targets ARM, the approach is not inherently limited to it. As prior studies show [32], some architectures, such as MIPS, also satisfy the key requirements of our approach. Therefore, with modest engineering effort to accommodate architecture-specific features, Khost can be extended to support other instruction set architectures as well.

Sanitizers. As highlighted in prior work [44], the effects of memory corruption are often less visible in MCU, which substantially reduces the effectiveness of traditional dynamic testing, such as fuzzing. To address this limitation, a fine-grained sanitizer is essential. Several approaches [23, 40] have been proposed. In contrast, our work focuses on improving efficiency while preserving the execution scope of MCU firmware rehosting. However, the design of a fine-grained memory detector is orthogonal to our objective. Therefore, we leave it as future work, where they can complement our rehosting framework and further enhance bug detection.

8 Related Work

Firmware rehosting is essential for analyzing embedded systems. Prior works [17, 29, 31, 34, 37, 43, 68] leverage hybrid emulation (known as hardware-in-the-loop) to rehost MCU firmware on desktop or server platforms, but tight hardware–firmware coupling limits their scalability. HALucinator [11] replaces functions in HALs with manually written stubs to overcome hardware dependence. While effective in some scenarios, they require significant human efforts and fail for proprietary or vendor-specific HALs, which are prevalent in real-world firmware. Para-rehosting [36] and LEMIX [55] apply similar HALs replacement methods, but they require access to the source code of the firmware.

To reduce manual effort, a series of works [4, 21, 24, 42, 50, 53, 60, 74] focus on automatically generating peripheral models based on access patterns, symbolic execution, or intercepting runtime communication between firmware and physical devices. Recent works [20, 61, 62] concentrate on modeling interrupt behavior to reduce false positives during rehosting. Other approaches build models from documentation [75] or extract device semantics directly from driver source code [35].

Besides, Icicle [8] enhances instrumentation techniques for architectural diagnostics, while MetaEmu [7] aims to broaden architectural coverage in rehosting. These works primarily target emulator extensibility, whereas Khost focuses on improving rehosting performance. SafireFuzz [52] and SURGEON [26] achieve performance gains by transforming ARMv7-M firmware into ARM Linux applications for execution on ARMv8-A platforms. Nevertheless, both approaches exhibit inherent limitations. For SafireFuzz, it neglects the architectural differences between ARMv7-M and ARMv8-A, resulting in potential inaccuracies in instruction-level semantics. For example, it can not support `svc` instructions. SURGEON models several MCU-specific behaviors, but some of its designs are still problematic. First, it uses floating-point registers to model the banked stack pointer registers. However, these registers are global and may also be used by the firmware for floating-point operations. Consequently, any modification to these registers can corrupt the global state. Second, it replaces incompatible operations, including `svc` instructions, with `bkpt`, which introduces additional runtime overhead and interferes with debugging the rehosted firmware to some extent. In addition, according to SURGEON's implementation, several MCU-specific operations, such as instructions that manipulate primask, basepri, and `cpsie` instruction, are simply rewritten as `nop`[6], which alters the firmware semantics.

Concerning MCU firmware analysis, Inception [14] lifts firmware to LLVM IR for further analysis. CO3 [39] and SymEx-VP [57] adopt concolic analysis for MCU firmware. FirmXRay [63] and FirmUp [16] apply static techniques to detect bugs in firmware binaries. Other fuzzing-oriented works like Farrelly et al. [18] demonstrate firmware fuzzing without peripheral modeling, while SFuzz [6] uses program slicing to facilitate fuzzing. Hoedur [51] and Multifuzz [9] focus on mapping fuzzing input to peripheral channels.

Additionally, several efforts [5, 33, 56, 72, 73] have explored rehosting and analyzing Linux-based firmware. They generally target systems with a more full-featured OS and MMU support. In contrast, our work focuses on MCU firmware, which typically operates in a bare-metal or RTOS environment with limited resources, no MMU, and highly customized peripheral interfaces.

[6] https://github.com/HexHive/SURGEON/blob/main/src/rewriter/transplantationinstrumentor.py

9 Conclusion

This paper proposes Khost, a near-native rehosting framework tailored for MCU firmware. Khost enables MCU firmware rehosting on high-performance platforms with ARMv8-A processors, by extending the KVM to introduce a lightweight extended CPU, an auxiliary page table, and a software-based interrupt controller. Khost also provides an MMIO Monitor for quick peripheral interactions and a firmware wrapper to enable efficient coverage collection and flexible fuzzing engine integration. Experiments on Coremark-PRO and Simbench show that Khost reduces execution overhead by 90.0% to 95.5% for complex computational tasks and by up to 98.5% for system-level operations, compared to QEMU. Furthermore, fuzzing with 12 real-world firmware, Khost achieves up to 197.5× higher throughput and improves basic block coverage by 6x, compared to existing fuzzing tools. Additionally, Khost successfully uncovers 5 previously unknown bugs.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116) and the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Yuan Zhang is the corresponding author.

Ethical Considerations

Our research is conducted by following the ethical guidelines and standards outlined in the conference's call for paper. First, fuzzing technologies inherently have dual-use potential, as they may enable adversaries to discover vulnerabilities more efficiently. However, their responsible use and disclosure substantially improve system security. Khost is designed strictly for enhancing the security testing of MCU firmware. Second, all experiments are conducted on publicly accessible datasets, including CoreMark-PRO, Simbench, and some real-world firmware. Third, all identified bugs mentioned in Section 6.4.2 are from non-safety-critical firmware. Within one week of bug discovery, we have responsibly reported all newly-discovered bugs with the mitigation to the Product Security Incident Response Team (PSIRT) of ST and Nuttx via email, strictly following the security policy of the STM32 project and Nuttx project. Till now, four maintainers have responded, and two confirmed the reported bugs and have fixed them in the latest release. To avoid potential impacts, firmware users can update to the latest version.

Open Science

Our research adheres to the principles of Open Science, ensuring accessibility, transparency, and reproducibility. All artifacts, including datasets, scripts, the source code of our tool, and patches to existing tools for ARM platform support mentioned in Section 6, are publicly available at: https://doi.org/10.5281/zenodo.17976459.

References

[1] Arm Limited (or its affiliates). ARMv7-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0403/ee/?lang=en. Accessed: June 1, 2025.

[2] Arm Limited (or its affiliates). Armv8-A virtualization. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20virtualization.pdf?revision=a765a7df-1a00-434d-b241-357bfda2dd31. Accessed: May 21, 2025.

[3] avem labs. A 6 Axes sensor, Quaternion and Euler Angles Compute-PID Controller. https://github.com/avem-labs/Avem. Accessed: May 30, 2025.

[4] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 746–759, 2020.

[5] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, volume 1, pages 1–1, 2016.

[6] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 485–498, 2022.

[7] Zitai Chen, Sam L Thomas, and Flavio D Garcia. Metaemu: An Architecture Agnostic Rehosting Framework For Automotive Firmware. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 515–529, 2022.

[8] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. Icicle: A Re-designed Emulator for Grey-box Firmware Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–88, 2023.

[9] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5359–5376, 2024.

[10] Valasek Chris and Miller Charlie. Remote exploitation of an unaltered passenger vehicle. https://illmatics.com/Remote%20Car%20Hacking.pdf. Accessed: May 21, 2025.

[11] 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 *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218, 2020.

[12] Abraham Anthony Clements, Logan Carpenter, William Moeglein, and Christopher Michael Wright. Is Your Firmware Real or Re-hosted? Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2021.

[13] The Standard Performance Evaluation Corporation. EEMBC CoreMark® benchmark. https://www.eembc.org/coremark-pro/. Accessed: May 21, 2025.

[14] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX security symposium (USENIX security 18)*, pages 309–326, 2018.

[15] Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, Manu Sridharan, et al. IDAPro for IoT Malware analysis? In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

[16] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUP: Precise Static Detection of Common Vulnerabilities in Firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

[17] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing Embedded Systems using Debug Interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1031–1042, 2023.

[18] Guy Farrelly, Michael Chesser, and Damith C Ranasinghe. Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 401–414, 2023.

[19] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia conference on computer and communications security*, pages 687–701, 2021.

[20] 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*, 21(4):3866–3882, 2023.

[21] Bo Feng, Alejandro Mera, and Long Lu. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

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

[23] 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):3420–3432, 2020.

[24] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the Analysis of Embedded Firmware through Automated e-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, 2019.

[25] Hex-Rays. IDA Pro: Powerful Disassembler, Decompiler & Debugger. https://www.hex-rays.com/ida-pro, 2025. Accessed: May 30, 2025.

[26] Florian Hofhammer, Marcel Busch, Qinying Wang, Manuel Egele, and Mathias Payer. SURGEON: Performant, Flexible, and Accurate Re-Hosting via Transplantation. In *Workshop on Binary Analysis Research (BAR)*, 2024.

[27] Red Hat Inc. What is KVM? https://www.redhat.com/en/topics/virtualization/what-is-KVM. Accessed: May 24, 2025.

[28] Global Growth Insights. ARM Microprocessor Market Size, Share, Growth, and Industry Analysis. https://www.globalgrowthinsights.com/market-reports/arm-microprocessor-market-101545. Accessed: December 9, 2025.

[29] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 321–338, 2021.

[30] jpconstantineau. BlueMicro Firmware. https://github.com/jpconstantineau/BlueMicro_BLE. Accessed: May 30, 2025.

[31] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[32] The kernel development community. The Definitive KVM (Kernel-based Virtual Machine) API Documentation. https://www.kernel.org/doc/html/v5.13/virt/kvm/api.html. Accessed: May 29, 2025.

[33] 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 *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 733–745, 2020.

[34] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[35] Chongqing Lei, Zhen Ling, Yue Zhang, Yan Yang, Junzhou Luo, and Xinwen Fu. A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7085–7102, 2024.

[36] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware. *arXiv:2107.12867*, 2021.

[37] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1–12, 2022.

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

[39] Changming Liu, Alejandro Mera, Engin Kirda, Meng Xu, and Long Lu. CO3: Concolic Co-execution for Firmware. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5591–5608, 2024.

[40] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, Heyuan Shi, and Yu Jiang. Effectively Sanitizing Embedded Operating Systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[41] Knud Lasse Lueth. State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time. https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time. Accessed: May 20, 2025.

[42] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1938–1954. IEEE, 2021.

[43] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5323–5340, 2024.

[44] 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 *NDSS*, 2018.

[45] Nuttx. NuttX Shell in the NuttX RTOS. https://github.com/apache/nuttx. Accessed: May 30, 2025.

[46] Arm Limited (or its affiliates). Arm CPU Architecture: A Foundation for Computing Everywhere. https://www.arm.com/architecture/cpu/. Accessed: May 24, 2025.

[47] Arm Limited (or its affiliates). Learn the architecture - ARMv8-A memory systems. https://developer.arm.com/documentation/100941/0101/Armv8-A-memory-systems. Accessed: May 20, 2025.

[48] Nguyen Anh Quynh. Capstone: Next-Gen Disassembly Framework. https://www.capstone-engine.org/BHUSA2014-capstone.pdf, 2014. Black Hat USA.

[49] Nguyen Anh Quynh. Keystone: Next Generation Assembler Framework. https://www.keystone-engine.org/docs/BHUSA2016-keystone.pdf, 2016. Black Hat USA.

[50] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, 2022.

[51] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *32rd USENIX Security Symposium (USENIX Security 23)*. CISPA, 2023.

[52] Lukas Seidel, Dominik Christian Maier, and Marius Muench. Forming Faster Firmware Fuzzers. In *USENIX Security Symposium*, pages 2903–2920, 2023.

[53] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. Conware: Automated Modeling of Hardware Peripherals. In *Proceedings of the 2021 ACM Asia conference on computer and communications security*, pages 95–109, 2021.

[54] STMicroelectronics. Integrated Development Environment for STM32. https://www.st.com/en/development-tools/stm32cubeide.html. Accessed: April 30, 2025.

[55] Sai Ritvik Tanksalkar, Siddharth Muralee, Srihari Danduri, Paschal Amusuo, Antonio Bianchi, James C Davis, and Aravind Kumar Machiry. LEMIX: Enabling Testing of Embedded Applications as Linux Applications. *arXiv:2503.17588*, 2025.

[56] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, et al. Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5791–5808, 2023.

[57] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. SymEx-VP: an open source virtual prototype for OS-agnostic concolic testing of IoT firmware. *Journal of Systems Architecture*, 126:102456, 2022.

[58] Texas Instruments Incorporated. Mcu and cpu. https://www.ti.com/microcontrollers-mcus-processors/overview.html. Accessed: May 21, 2025.

[59] Harry Wagstaff, Bruno Bodin, Tom Spink, and Björn Franke. SimBench: A portable benchmarking methodology for full-system simulators. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 217–226. IEEE, 2017.

[60] Chunlin Wang and Hongliang Liang. Value Peripheral Register Values for Fuzzing MCU Firmware. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 718–729. IEEE, 2023.

[61] Jianqiang Wang, Qinying Wang, Tobias Scharnowski, Li Shi, Simon Woerner, and Thorsten Holz. AidFuzzer: Adaptive Interrupt-Driven Firmware Fuzzing via Run-Time State Recognition. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2025.

[62] Yuan Wei, Yongjun Wang, Lei Zhou, Xu Zhou, and Zhiyuan Jiang. IEmu: Interrupt modeling from the logic hidden in the firmware. *Journal of Systems Architecture*, 154:103237, 2024.

[63] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal irmware. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 167–180, 2020.

[64] Wikipedia. Stuxnet. https://en.wikipedia.org/wiki/Stuxnet. Accessed: May 21, 2025.

[65] WiseGuyReport. Global Standalone Embedded Systems Market Research Report. https://www.wiseguyreports.com/reports/standalone-embedded-systems-market. Accessed: May 29, 2025.

[66] 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.

[67] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of Embedded Systems: A Survey. *ACM Computing Surveys*, 55(7):1–33, 2022.

[68] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, volume 14, pages 1–16, 2014.

[69] Micha Zalewski. American Fuzzing Lop. https://lcamtuf.coredump.cx/afl/. Accessed: May 21, 2025.

[70] Michal Zalewski. American Fuzzy Lop v1.31b. https://github.com/google/AFL/releases/tag/v1.31b. First public release of QEMU instrumentation mode.

[71] Zephyr. A gateway between an Ethernet TCP-IP network and a Modbus serial line. https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/subsys/modbus. Accessed: May 30, 2025.

[72] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[73] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient Greybox Fuzzing of Applications in Linux-based IoT devices via Enhanced User-Mode Emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 417–428, 2022.

[74] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic Firmware Emulation Through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2007–2024, 2021.

[75] 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 *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3269–3283, 2022.

Appendix

A Basic Information of Dataset Used in Evaluating Rehosting Performance

To comprehensively evaluate the rehosting performance of Khost, we conduct experiments on the CoreMark-PRO dataset and Simbench dataset. The basic information of these two datasets is shown in Table 7 and Table 8, respectively.

Table 7: The basic information of CoreMark-PRO test cases.

Cases	Parameters	Description
cjpeg	-v0 -i100	JPEG compression workload for testing image encoding performance.
core	-v0 -i1	Computational kernels stressing arithmetic and memory operations.
liner	-v0 -i100	Linear algebra on 100×100 single-precision floating-point matrices.
nnet	-v0 -i1	Feed-forward neural network for machine learning computation.
parser	-v0 -i100	XML parsing workload for structured data processing.
radix	-v0 -i100	Radix sort stresses integer operations and memory access locality.
sha	-v0 -i1000	SHA-256 cryptographic hashing workload measuring bitwise operations and data throughput.
zip	-v0 -i10000	ZIP compression and decompression testing data processing performance.

B Fuzzing Experiments on Dell Server

As described in Section 6.4, we port HALucinator and Fuzzware to the KunPeng (ARM) server for a fair comparison. To ensure completeness of our evaluation and further validate the performance of Khost, we also run HALucinator and Fuzzware on the more powerful Dell (x86) server. The final results are summarized in Table 9. Compared to HALucinator, Khost achieves a 6.4×∼115.0× increase in throughput and up to a 7.0× improvement in coverage. When compared to Fuzzware, Khost achieves up to 49.4× higher throughput and 3.4× more basic block coverage. These results comprehensively demonstrate that Khost still outperforms existing tools, even when they run on more powerful servers.

Table 8: The basic information of Simbench test cases. We exclude four test cases incompatible with MCUs: three cases require MMU support, which is unavailable on most MCUs, and one uses the `swi` instruction, which is also unsupported in MCU firmware.

Type	Test cases	Iterations	Description
Memory	mem-hot	500M	Test memory access with high locality (hot data).
	mem-cold	2000M	Test memory access with low locality (cold data).
I/O	co-processor	1250M	Evaluate the performance of coprocessor operations.
	device-access	40M	Simulate memory-mapped device register accesses.
Exception	dfault	25M	Test data fault handling mechanisms.
	ifault	25M	Test instruction fault handling and recovery.
	syscall	50M	Simulate system call execution overhead.
	undef	50M	Test handling of undefined instructions.
Control Flow	inter-page-direct	1000M	Test direct memory accesses crossing page boundaries.
	inter-page-indirect	1250K	Test indirect memory accesses across pages.
	intra-page-direct	1000M	Test direct memory accesses within the same page.
	intra-page-indirect	1M	Test indirect memory accesses within the same page.
Codegen	large-blocks	100M	Test memory operations on large data blocks.
	small-blocks	40M	Test memory operations on small data blocks.

Table 9: Basic Block Coverage (BBL) and fuzzing throughput (the number of test cases that can be executed per second, Exec/s) of different tools when fuzzing 12 real-world MCU firmware. '-' indicates that the firmware could not be rehosted. The number on the left of '/' shows total coverage, while the number on the right shows coverage excluding basic blocks in HALs.

		HALucinator (x86)		**Fuzzware (x86)**		**SafireFuzz**		**Khost-AFL++**	
Firmware	Total BBL	Exec / s	BBL	Exec/s	BBL	Exec/s	BBL	Exec/s	BBL
atmel_6lowpan_rx	6706	3.1	1496	86.1	2600 / 2537	-	-	274.7	3154 / 2802
tcp_echo_client	7359	8.0	1005	431.8	460 / 266	3194.0	1052	276.7	2000 / 1636
tcp_echo_server	6895	8.8	732	501.6	458 / 267	3652.0	795	332.6	1254 / 926
PLC	2304	8.6	106	69.0	630 / 469	3744.0	143	379.4	744 / 549
Drone	2726	2.0	271	22.2	583 / 399	2910.0	239	231.5	1436 / 1211
Heat-Press	1832	34.8	201	41.5	550 / 458	3701.0	201	258.4	579 / 484
Stm32_Sine	3518	26.7	416	34.9	435 / 433	4163.6	416	1053.9	1364 / 1328
LibJPEG_decoding	5345	20.3	649	175.3	769 / 678	4241.2	174	536.2	873 / 777
BlueMicro_BLE	9475	-	-	67.1	734	-	-	157.0	762
Avem	1248	-	-	49.8	921	-	-	1080.5	916
Zephyr-Modbus	8755	-	-	67.0	2067	-	-	118.9	2119
nuttx-nsh	8986	-	-	29.2	2196	-	-	1473.2	2350