

BoardRunner: Automatic Firmware Rehosting using High-Fidelity Compositional Device Models

Muhammad Hammad Bashir
The Pennsylvania State University
University Park, PA, USA
hammad.bashir@psu.edu

Michael Rooney
Purdue University
West Lafayette, IN, USA
Rooney13@purdue.edu

Colin Smith
The Pennsylvania State University
University Park, PA, USA
cts5520@psu.edu

Dongyan Xu
Purdue University
West Lafayette, IN, USA
dxu@purdue.edu

Arslan Khan
The Pennsylvania State University
University Park, PA, USA
arslankhan@psu.edu

ABSTRACT

Analyzing embedded systems firmware for security vulnerabilities is challenging because physical testbeds are costly and difficult to scale. Firmware rehosting, which executes firmware in an emulator, provides an alternative, but existing approaches face trade-offs among fidelity, performance, and scalability. To address these limitations, BOARDRUNNER uses compositional device modeling, allowing individual peripherals to be handled by independently selected modeling backends while preserving their dependencies. BOARDRUNNER provides reusable Virtual I/O (VIO) models that capture the high-level operational semantics of commodity peripherals across different MicroController Units (MCUs). To adapt these models to new targets, BOARDRUNNER passively observes Hardware-In-The-Loop (HITL) executions and uses automaton learning and LLM-guided synthesis to generate mappings from target-specific MMIO interactions to reusable VIO semantics. To reduce the execution overhead of peripheral modeling, BOARDRUNNER uses FASTDYN, a high-speed rehosting engine that integrates device-model handling directly into JIT-translated firmware execution. We evaluate BOARDRUNNER across five MCU boards and 70 Software Development Kit (SDK) firmware workloads, including DMA-intensive workloads, demonstrating end-to-end rehosting through a semi-automated pipeline. On end-to-end macrobenchmarks, FASTDYN achieves a 12.74× average speedup across four workloads compared with HALucinator. Finally, we demonstrate that BOARDRUNNER-generated models can be reused as fuzzing harnesses for real-world firmware targets, providing a high-fidelity execution substrate for downstream dynamic security analysis.

CCS CONCEPTS

• Security and privacy → Embedded systems security; Software security engineering.

KEYWORDS

firmware rehosting, emulation, LLM, peripheral modeling, embedded security

ACM Reference Format:

Muhammad Hammad Bashir, Michael Rooney, Colin Smith, Dongyan Xu, and Arslan Khan. 2026. BoardRunner: Automatic Firmware Rehosting using High-Fidelity Compositional Device Models. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3830454.3846632>

1 INTRODUCTION

Embedded systems are foundational to modern life, powering everything from critical infrastructure to personal devices, and are often built around low-cost microcontroller units (MCUs) that execute monolithic firmware in a flat address space. Unfortunately, MCUs provide very limited analysis capabilities, leaving various vulnerabilities [11, 13, 30], including remote code execution [7, 20, 37], in the wild. Firmware rehosting has emerged as a promising alternative, enabling firmware to be executed and analyzed within a software-based emulator without the need for physical hardware. However, existing rehosting frameworks suffer from fundamental limitations that impede their practicality and performance.

Software-based emulation frameworks [4] require manual peripheral modeling, which scales poorly due to the sheer diversity of embedded hardware. While automatic rehosting methods [27, 41, 52] attempt to infer peripheral behavior by analyzing the firmware, they struggle with complex devices and provide low emulation fidelity. For instance, a General Purpose Input/Output (GPIO) pin is often emulated only superficially, while complex peripherals like cryptographic and neural-network accelerators are simply too opaque to model entirely in software.

To mitigate this issue, Hardware-in-the-Loop (HITL) rehosting [33] forwards peripheral access requests to a physical device. For security analysis, this approach highlights a critical trade-off that is not merely an execution-performance issue. Low-fidelity software abstractions may allow firmware to execute, but can fail to preserve the stateful peripheral and protocol behavior needed to exercise security-relevant functionality. Conversely, HITL preserves these exact semantics, but the constant synchronization



between the emulator and the hardware imposes a severe performance penalty, making repeated dynamic analyses such as fuzzing prohibitively expensive.

To bridge this gap, high-level rehosting approaches, such as HALucinator [9], attempt to reduce modeling effort by mapping firmware functions directly to reusable high-level peripheral models. However, because HALucinator operates at a higher firmware abstraction layer, it requires manual, target-specific mappings between firmware abstractions and the high-level models. Furthermore, it struggles to compose independently handled peripherals whose states depend on one another (e.g., through clock, power, or interrupt-control logic). Together, these bottlenecks suggest that high-fidelity rehosting should be compositional: each peripheral should use the modeling strategy best suited to it, while preserving dependencies among independently handled devices.

To address these challenges, we design BOARDRUNNER, a compositional approach to automatic firmware rehosting. BOARDRUNNER shares HALucinator’s conceptual goal of mapping firmware interactions to reusable models, but it overcomes the limitations of high-level emulation by operating directly at the Memory-Mapped I/O (MMIO) boundary. Instead of relying on manual target-specific mappings, BOARDRUNNER makes inter-peripheral dependencies explicit, allowing different peripherals to use different modeling backends while maintaining a consistent state across their dependencies.

BOARDRUNNER provides a library of Virtual I/O (VIO) models that 1) capture the high-level operational semantics of commodity peripherals and 2) standardize inter-peripheral interactions. To automate the adaptation of these models to new firmware interfaces, BOARDRUNNER introduces VIOMAPPER. VIOMAPPER passively observes HITL interactions and iteratively generates target-specific mappings from low-level registers to VIO models using automaton learning and LLM-guided synthesis. The learned mapping can subsequently be used to either eliminate the need for physical hardware entirely, or construct a hybrid HITL/VIO rehosting configuration.

Lastly, to reduce the execution overhead of peripheral modeling, BOARDRUNNER uses FASTDYN, a high-speed emulation engine that integrates device-model dispatch and analysis instrumentation directly into Just-In-Time (JIT)-translated firmware code. FASTDYN supports both software-based models and HITL passthrough, enabling flexible compositional rehosting.

BOARDRUNNER therefore provides a faithful execution substrate for dynamic security analysis of hardware-coupled firmware, enabling repeated analyses such as fuzzing and protocol testing. As shown in our evaluation, BOARDRUNNER successfully rehosts SDK firmware workloads across five MCU boards and four real-world firmware targets. For peripheral classes covered by the VIO library, rehosting is semi-automated; supporting a previously unsupported peripheral class requires implementing a reusable VIO model once.

In summary, our contributions are as follows:

- We design *Compositional Device Modeling*, an automatic rehosting technique that maps low-level firmware interactions to reusable, high-fidelity Virtual I/O models while preserving inter-peripheral dependencies. This enables faithful execution of hardware-coupled firmware without requiring a complete platform-specific software model.

- We design FASTDYN, BOARDRUNNER’s high-performance execution engine, which integrates device modeling directly into the translated firmware execution loop. FASTDYN makes high-fidelity compositional rehosting efficient enough for repeated dynamic analysis. On end-to-end macrobenchmarks, FASTDYN achieves a 12.74× average speedup across four workloads compared with HALucinator.
- We evaluate BOARDRUNNER across five MCU boards and 70 SDK firmware workloads, as well as four real-world firmware targets. Beyond functional rehosting, we demonstrate that the generated models can be reused as fuzzing harnesses for real firmware stacks, showing that BOARDRUNNER provides the execution fidelity required by downstream dynamic security analysis.

BOARDRUNNER code will be public post-acceptance.

2 BACKGROUND

2.1 Microcontroller Peripherals

Microcontrollers (MCUs) are equipped with a rich set of peripherals that enable interaction with other controllers and the physical environment. These peripherals are typically accessed through memory-mapped I/O (MMIO), where peripheral registers are mapped into the processor’s address space and accessed using regular load and store instructions. The semantics and layout of these registers are defined by the hardware vendor and documented in the platform’s reference manual or datasheet. In addition to on-chip peripherals, MCUs often communicate with external devices via standard buses such as Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I²C), each providing its own protocol for device addressing and data exchange.

2.2 QEMU and Dynamic Binary Translation

QEMU [4] executes guest code using dynamic binary translation. During execution, blocks of guest instructions are translated into host instructions and cached for subsequent execution. For MCU firmware, memory-mapped peripheral accesses originating from the translated code are ultimately dispatched to the corresponding device model or external handler. Firmware-analysis frameworks often interpose on this execution to inspect or modify machine state or to handle peripheral interactions. When such operations leave the translated execution path, for example through a debugger interface or an external handler, each transition introduces additional synchronization and dispatch overhead. In contrast, instrumentation inserted directly into the translated code can execute within the normal JIT execution path.

3 RELATED WORK

MMIO-Level Peripheral Modeling. Prior work models peripheral behavior directly at the MMIO boundary. Symbolic approaches [5, 12, 22, 23, 40, 51] infer values that satisfy firmware expectations, while fuzzing-based approaches [8, 17, 38, 39] explore peripheral inputs to sustain or expand execution. These techniques are effective for code exploration, but generally model what the firmware expects rather than the higher-level semantics of the underlying

device. Trace-driven approaches [21, 41] learn peripheral behavior from observed hardware interactions, but remain limited by behaviors exercised during trace collection. Specification-guided approaches [27, 31, 52] derive richer semantics from datasheets or software artifacts, but depend on their availability and completeness. In contrast, VIOMAPPER uses observed MMIO interactions to map target-specific behavior onto reusable high-level VIO semantics.

High-Level Peripheral Modeling. High-Level Modeling (HLM) [9, 28] avoids reproducing complete register-level behavior by redirecting firmware interactions to reusable peripheral models. HALucinator [9] is the closest prior work to BOARDRUNNER, but requires identifying and mapping firmware-level hardware abstractions to these models. BOARDRUNNER instead operates at the MMIO boundary and uses VIOMAPPER to learn the mapping from target-specific register interactions to VIO operations. Moreover, BOARDRUNNER explicitly captures inter-peripheral dependencies, allowing independently modeled peripherals to maintain consistent shared state. *Hardware-Assisted and Hybrid Rehosting.* PROSPECT [25], Avatar [50], and Avatar2 [33] preserve peripheral fidelity by forwarding selected interactions to physical hardware, but frequent synchronization and state transfer can impose substantial overhead. Secureware [24] reduces hardware accesses through caching, while SURROGATES [26] and Inception [10] use specialized hardware to accelerate communication. BOARDRUNNER instead treats HITL as one backend within a compositional architecture, allowing different peripherals to use software models, replay, or hardware passthrough while FASTDYN preserves dependencies among them.

On-Device Firmware Analysis. On-device approaches preserve native peripheral behavior but remain coupled to physical hardware. μ AFL [29] relies on hardware tracing. GDBFuzz [16], TWFuzz [18], and SHiFT [32] use debugging, tracing, or semihosting mechanisms to improve fuzzing throughput. These approaches retain hardware fidelity, but physical execution limits scalability. BOARDRUNNER instead uses hardware execution to learn peripheral behavior and subsequently moves supported interactions into the rehosted environment.

4 OPERATIONAL MODEL

BOARDRUNNER operates in a semi-whitebox setting to enable downstream dynamic security analysis, including fuzzing and protocol testing. To initialize the environment, the analyst supplies the firmware binary, physical board access for Hardware-in-the-Loop (HITL) tracing, and basic platform configurations (e.g., CPU, memory map, peripheral addresses, and inter-peripheral dependencies). BOARDRUNNER also relies on register metadata to associate observed MMIO accesses with peripheral semantics. Our implementation uses CMSIS System View Descriptor (SVD) files, though equivalent data from vendor documentation or automated inference suffices. If this metadata is unavailable, the rehosting process requires additional manual effort.

For peripheral classes currently in the VIO library, VIOMAPPER automatically synthesizes the target-specific MMIO mappings. Unsupported peripheral classes require a one-time VIO model implementation. Ultimately, BOARDRUNNER produces a flexible compositional environment where analysts can mix VIO models, hardware

Table 1: Comparison of Peripheral Modeling Techniques.

Technique	Perf.	Fidelity	Scal.	Effort
Software-Based:				
- MLM	High	Low	Low	Low
- HLM	High	High	High	High
HITL	Low	High	Low	Low
BOARDRUNNER (Compositional)	High	High	High	Low

passthrough, and replay backends on a per-peripheral basis. Finally, because prior frameworks make different baseline assumptions (e.g., black-box or grey-box), we evaluate BOARDRUNNER against related work strictly within their respective operational contexts.

5 MOTIVATION

Firmware rehosting executes an existing firmware binary within an emulated environment that approximates its original hardware context. Table 1 gives an overview of existing peripheral modeling techniques. Software-Based Modeling creates a software model for the emulated peripherals, resulting in a fast emulation mechanism. Due to the diversity of MCU hardware, they usually employ an inference algorithm to automatically generate the software model. *MMIO-Level Modeling (MLM)* analyzes the firmware using techniques such as symbolic execution [5, 12, 22, 23, 40], fuzzing [8, 17, 38, 39], specification analysis [52], and machine-learning [21, 41] to generate a device model based on how the firmware interacts with the peripheral at the MMIO layer. While these approaches are very easy to use and require the least amount of manual work, they have many fundamental issues limiting their practicality. Firstly, the emulation fidelity is low, i.e., the generated model does not have a meaningful emulation. For example, when interfacing with a GPIO pin, the user cannot meaningfully manipulate or observe the pin’s state as they could on real hardware.

Alternatively, *High-Level Modeling (HLM)* [9] targets a higher-level abstraction for MMIO interactions, thereby reducing the impact of peripheral-level diversity. For example, HALucinator uses binary-to-library matching to identify known HAL functions in stripped firmware and redirects them to reusable high-level peripheral models. However, non-standard implementations demand substantial manual effort to correctly map the abstract model to the target firmware.

Unfortunately, software-based modeling does not scale as the complexity of the device increases. Moreover, some peripherals, such as cryptographic and neural-network accelerators, are simply too opaque or complex to model entirely in software. Therefore, for such peripherals, Hardware-In-The-Loop (HITL) techniques are used, which forward the MMIO access directly to the hardware, alleviating the complexity of the peripheral. However, doing so requires costly hardware interactions, resulting in high overhead during execution.

Each of these approaches offers distinct advantages yet suffers from tradeoffs between fidelity, automation, or scalability. There is a need for a technique that unifies the strengths of all three

Table 2: Feature comparison among existing rehosting platforms and BOARDRUNNER. ●: full, ○: partial, ○: none.

Feature	Unicorn [34]	QEMU [4]	PANDA [14]	Avatar ² [33]	BOARDRUNNER
Multi-target Peripherals	○	○	○	●	●
Inline Device Modeling	○	○	○	○	●
Analysis Plugins	●	○	●	●	●
DMA Support	○	○	○	○	●

paradigms, i.e., automation of Low-Level modeling, flexibility and fidelity of High Level Models, with fallback capabilities of HITL.

Our Approach. BOARDRUNNER employs an MLM solution to map MMIO accesses to a high-fidelity HLM called VIO. Moreover, the high-fidelity design of VIO models uses well-defined interfaces, allowing a split configuration where some peripherals are served by VIO Models, whereas some peripherals are served by HITL.

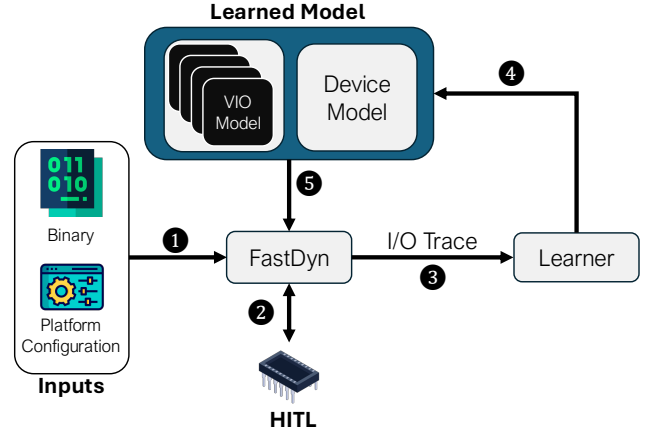
However, there are two major challenges to realizing such a design. Firstly, as the number of MMIO accesses increases, the firmware-peripheral interaction becomes a major source of bottleneck. Secondly, while peripherals mostly work independently, they still have some dependencies, especially in system-level peripherals, such as clock and power control. Therefore, we need a fast IO emulation mechanism that can route a single IO access to multiple device models.

To address these limitations, an ideal rehosting platform must offer *first-class rehosting support*. Specifically, it requires a robust architecture, such as a multi-target device model, capable of modeling the communication between firmware components and heterogeneous device-model backends. Furthermore, to reduce synchronization and dispatch overhead, the platform should support inline device modeling directly within the JIT translation path. Table 2 summarizes how the rehosting platforms we consider differ in their support for these capabilities.

6 OVERVIEW

BOARDRUNNER is a firmware rehosting framework that uses compositional device modeling to support scalable, high-fidelity rehosting. Rather than constructing a monolithic platform model, BOARDRUNNER decomposes the rehosting problem into per-device subproblems. For peripherals supported by the VIO library, VIOMAPPER automatically synthesizes and refines target-specific mappings using firmware execution traces and LLM-guided synthesis.

At a high level, Figure 1 illustrates the workflow of BOARDRUNNER. BOARDRUNNER takes the firmware and platform information encoded in a configuration file as its inputs. BOARDRUNNER uses its execution engine, FASTDYN, to execute the target firmware in a Hardware-in-the-Loop (HITL) configuration and trace the interaction patterns between the firmware and each device. These traces are then fed to VIOMAPPER, which synthesizes a mapping between the target-specific low-level register interactions and a Virtual I/O (VIO) model. Together, these components enable BOARDRUNNER to

**Figure 1: BOARDRUNNER Overview.** FASTDYN allows modifying firmware execution using instrumentation and emulates peripherals using its device models.

perform fine-grained, semi-automated firmware rehosting across heterogeneous hardware targets with limited manual intervention.

7 DESIGN

As mentioned in section 5, existing rehosting platforms suffer from key challenges. To this end, we set up the design goals for our rehosting platform.

- **G1 High Fidelity:** BOARDRUNNER must achieve a high level of fidelity, mapping a physical device to a high-fidelity virtual instance, ensuring users can create a modular rehosted setup that can scale up.
- **G2 Efficient Emulation:** BOARDRUNNER must minimize the execution overhead introduced by instrumentation and peripheral modeling, enabling efficient repeated dynamic analysis.
- **G3 Low Manual Effort:** BOARDRUNNER must rehost unknown firmware with minimal manual intervention.

In the following sections, we list the three main components of BOARDRUNNER. 1) Virtual I/O (VIO) Models, 2) FASTDYN, and 3) VIOMAPPER.

8 VIRTUAL I/O (VIO) MODELS

One of the primary challenges in firmware rehosting is the severe lack of standardization across different microcontrollers (MCUs). This lack of standardization manifests in two critical ways that impede scalable emulation: intra-peripheral diversity and inter-peripheral dependencies.

First, each Original Equipment Manufacturer (OEM) provides its own proprietary protocol to initialize and operate a given peripheral. For example, to use a Universal Synchronous/Asynchronous Receiver/Transmitter (USART), the protocol to manipulate the register file is completely different between STMicroelectronics' STM32F4Discovery and Analog Devices' MAX78000 MCUs. These differences arise from varying register layouts, clocking models,

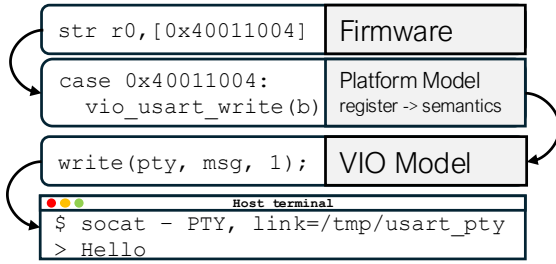


Figure 2: Virtual USART in action. Vendor-specific USART MMIO accesses are translated into high-level VIO operations and surfaced at the host as a pseudo-terminal.

and initialization sequences. Due to this diversity, writing a new model for each platform is not scalable.

Second, existing works typically superficially emulate several key system-level dependencies, such as the clock tree, because high-fidelity emulation is often not strictly required to merely achieve basic firmware code coverage. However, compositional rehosting fundamentally breaks under this assumption. Because each peripheral may independently reside in either software emulation or physical hardware (HITL), a low-fidelity clock module fails to synchronize state across these domains. For example, consider a hybrid configuration where a complex system control module, e.g., the Reset and Clock Control (RCC), executes on physical hardware via HITL, while a target GPIO peripheral is emulated in software. On a physical board, if the firmware clears the GPIO’s clock-enable bit within the RCC, the GPIO is disabled, and any subsequent reads to its data registers will be dropped or return zero. However, in a naive compositional setup, the software-emulated GPIO is completely isolated and unaware of the physical RCC’s state changes. Consequently, the emulated GPIO will incorrectly continue to process reads and writes, causing the emulator’s execution state to silently diverge from reality. Therefore, high fidelity in emulated peripherals is not merely a “nice-to-have” feature; it is a strict necessity for compositional modeling.

VIO models mitigate these challenges by providing a library of high-fidelity abstract emulation models of commodity peripherals. To address OEM diversity, we observe that while manually creating an emulation model for every distinct USART variant is unscalable, all implementations ultimately share the same high-level operational semantics. The firmware must initially set common parameters (e.g., baud rate, flow control), after which it simply reads from or writes to the peripheral. VIO models capture this abstract, high-level behavior, creating a high-fidelity emulation of a virtual device that is not tied to any particular hardware implementation. For GPIO, for example, a target-specific write to a GPIO output-data register is translated by the generated model into the corresponding VIO operation that updates the abstract pin state, while reads from the input-data register are mapped to the VIO operation that queries that state. To prevent the desynchronization issues seen in prior work, BOARDRUNNER standardizes inter-peripheral interactions by defining explicit abstract interfaces for probing and updating dependency states. Figure 6 shows how this capability is used to maintain consistency across the hardware-software boundary.

```

1 void usart3_write(void *opaque, hwaddr addr,
2                  uint64_t value, unsigned size) {
3     USART3State *s = opaque;
4     switch (addr - USART3_BASE) {
5     case USART_TDR_OFF:
6         api_pty_write_req(s->pty_fd, (uint8_t)value);
7         break;
8         /* ... other registers ... */
9     }
10 }
  
```

Figure 3: Excerpt from a VIOMAPPER-generated USART3 model on NUCLEO-H753ZI.

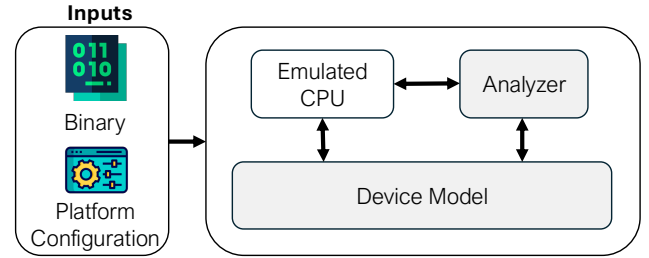


Figure 4: FASTDYN Overview. FASTDYN allows modifying firmware execution using instrumentation and emulates peripherals using its device models.

By leveraging these abstract virtual models, BOARDRUNNER successfully achieves its first design goal, *G1: High Fidelity*. Figure 2 illustrates this architecture in action using the USART VIO model. From the firmware’s perspective, it simply reads and writes to its native, vendor-specific MMIO register interface. However, the BOARDRUNNER device model transparently intercepts these accesses and translates them into standardized VIO calls routed to a host-facing pseudo-terminal. As shown in Figure 3, a write to USART3’s Transmit Data Register (TDR) becomes a single VIO call (`api_pty_write_req`) that surfaces the byte at the host PTY, and the read path from the Receive Data Register (RDR) is symmetric. Consequently, a user on the host can observe the output and directly drive the serial byte stream, interacting with the emulated peripheral exactly as if it were a physical UART.

8.1 FASTDYN

BOARDRUNNER uses FASTDYN as its emulation engine. FASTDYN is a QEMU-based [4] emulation engine. At its core, FASTDYN aims to provide **Fast Dynamic** firmware analysis. FASTDYN allows a very flexible rehosting configuration, allowing dynamic instrumentation and a rich device model that allows switching between different device models. Figure 4 shows the internal components of FASTDYN. To use FASTDYN, users must provide the firmware binary to rehost and the **Platform Configuration**, giving information about the target platform for the binary. FASTDYN allows modifying and observing the behavior of the firmware using **Analyzer**. Lastly, FASTDYN implements **Device Models** to facilitate compositional rehosting. We describe the design of each component in the following sections.

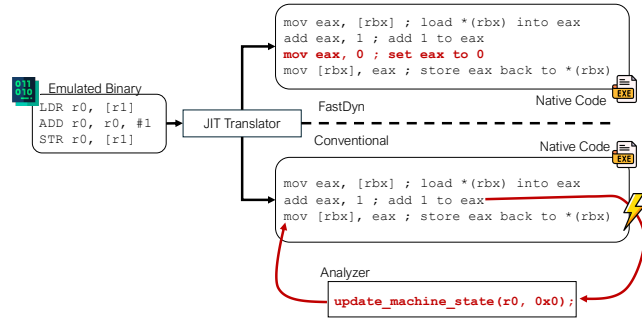


Figure 5: Difference in conventional firmware analysis techniques and FASTDYN’s Analyzer. FASTDYN extends the JIT translation layer to inject native code required for firmware analysis, resulting in a very high analysis throughput.

8.1.1 Analyzer. Analyzer is the main module responsible for manipulating and analyzing the state of the emulated firmware. In order to do so, we need to instrument the firmware to manipulate and dump the state of the machine. One possible approach is binary instrumentation, but unfortunately, binary instrumentation is usually brittle; therefore, prior work has relied on the emulator’s debugging interface to perform dynamic state modifications. Existing work [9, 33] implements their instrumentation logic by issuing debug commands to the underlying emulator. Unfortunately, this approach is slow for several reasons. Beyond the obvious overhead of halting execution, a subtler bottleneck arises from internal caching behavior. Modern emulators operate much like JIT compilers: they translate firmware instructions into host instructions and cache translated basic blocks, decoded state, and auxiliary metadata. However, every time the debugger halts execution or modifies architectural state, the emulator is forced to invalidate these caches to preserve consistency. As a result, each instrumentation action becomes extremely costly.

To address this, FASTDYN augments the JIT translation of the emulator to directly manipulate the translation of the code from target instruction to host instruction. Figure 5 highlights the difference from the existing approach. FASTDYN directly injects the required instrumentation during translation of code from target code to native code. Using this technique, FASTDYN allows two types of instrumentation. The injected code can be used to do inline modification (modifiers) or call the FASTDYN plugin infrastructure (virtual instructions).

1) Dynamic Binary Instrumentation (Modifiers) Depending on the use case, simple state updates can be executed inline without leaving the code buffer, while more complex transformations are supported via specialized FASTDYN plugins invoked from within the translated code. A common use case during rehosting is to instrument the firmware. For instance, users may want to skip a particular condition during execution. These are basically dynamic instrumentation.

2) FASTDYN Plugins (Virtual Instructions). While Modifiers provide an efficient mechanism for performing lightweight, inline state updates, more sophisticated analyses, such as complex semantic checks, multi-step inspections, or operations requiring

host-side services, for example, propagating taint information to track whether untrusted input flows into privileged MMIO registers, which requires updating host-side taint metadata, checking cross-register dataflow, and optionally blocking unsafe writes, demand a more expressive interface. To support these cases, FASTDYN introduces Virtual Instructions, i.e., instructions that are not part of the firmware ISA, but can be seen as powerful instructions that can interact with the host. Virtual instructions programmable callbacks that can be directly injected in the translated host code and allow the JIT engine to seamlessly transition from executing firmware logic to executing host-provided analysis routines without resorting to debugger intervention.

Through this mechanism, BOARDRUNNER exposes a rich API for inspecting and modifying system state, building introspection tools, implementing dynamic analyses, and performing higher-level instrumentation tasks. This design enables complex reasoning about firmware behavior while retaining the performance benefits of inlined, JIT-level instrumentation.

8.1.2 FASTDYN Device Model. FASTDYN intercepts all MMIO accesses performed by the firmware and forwards them to its device model infrastructure. Moreover, FASTDYN allows the device model to use the Analyzer module as well to achieve flexible emulation configuration using virtual instructions and modifiers, allowing inline device emulation. Figure 6 shows the overview of the device model. At the heart of the device emulation is the *Device Model*. The device model defines how the IO access is emulated. FASTDYN allows each peripheral to be independently handled by different device-model backends, enabling flexible compositional rehosting configurations. However, this flexibility introduces an inter-peripheral dependency problem, as real systems contain subtle dependencies that must be respected for correct execution. A representative example is the GPIO peripheral. While its register interface can be modeled in isolation, accurate execution requires that the underlying clock tree be properly configured and emulated. Since different peripherals may be handled by different models, state changes in one model must remain visible to dependent peripherals handled by another model. This creates a dependency relationship that must be explicitly captured and enforced. FASTDYN addresses this problem using Device Dependency Chains and MultiTarget Peripherals (MTP), which together allow MMIO accesses to be routed to multiple relevant models when required by an inter-peripheral dependency. We describe these mechanisms in the following sections.

Device Dependency Chains. Dependencies are explicitly declared in the Platform Configuration. In the GPIO example, a dependency entry is added from GPIO to the Clock module, ensuring that any model serving GPIO also handles the relevant clock registers.

MultiTarget Peripherals (MTP). Given the dependency chain, MTP enables routing an MMIO access to multiple models when required. This behavior contrasts with traditional hardware system agents, where each MMIO access is delivered to exactly one peripheral, by allowing FASTDYN to forward a single access to all models responsible for satisfying a dependency. This ensures consistency across interdependent device models while preserving the flexible model architecture of FASTDYN.

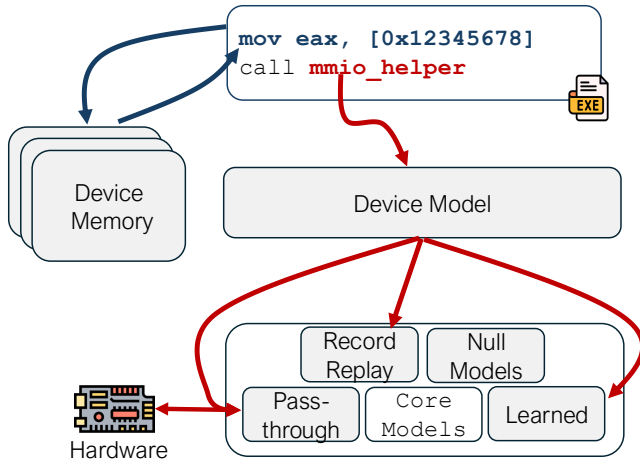


Figure 6: FASTDYN Device Model. Each device access can be routed to multiple models. The device model utilizes virtual instructions and modifiers to achieve inline device modeling.

Device models. FASTDYN provides multiple device-handling models to support diverse rehosting scenarios. We describe the design of different models as follows:

Core model: Handwritten, built-in models that are statically linked with FASTDYN. These models implement the essential behavior of common peripherals and serve as fast, stable references.

Passthrough model: This model forwards MMIO operations directly to physical hardware. BOARDRUNNER uses Virtual Instructions to invoke host-side passthrough operations directly from JIT-translated code. To avoid routing these operations through external programs such as OpenOCD and protocols such as TCL, we introduce LibHW. LibHW provides native access to debug interfaces such as ST-Link, J-Link, and CMSIS-DAP, allowing BOARDRUNNER to communicate directly with physical hardware.

Learned model: Learned models are external, dynamically loaded models plugged into FASTDYN as shared libraries. These models may be produced by automated rehosting techniques (such as VIOMAPPER), and the decoupled design allows FASTDYN to integrate any modeling approach that adheres to its device-model API.

Record/Replay (RR) model: During an HITL run, this model records every MMIO access and IRQ alongside the emulator instruction count at which it occurred. In a subsequent emulated run, it replays those events deterministically, synchronized to the instruction count rather than wall-clock time. Record/replay provides exact reproducibility for a captured execution and, as we show in our evaluation, serves as a reference oracle when assessing how well a learned model matches the recorded behavior.

Null model: A placeholder model whose reads return zero and whose writes are discarded. It is useful for addressing the ranges the user wants to deliberately ignore.

8.1.3 Platform Configuration. In order to initialize a target platform, FASTDYN requires three configuration components:

CPU: Specifies the target CPU architecture and core configuration. This also governs any Modifiers and Virtual Instructions that the firmware relies on.

Memory: Describes the memory layout of the platform, including RAM, flash, and any special-purpose regions used by the firmware, such as DMA regions, MMIO burst regions, alias regions, etc.

Device: Defines how I/O address ranges are routed inside FASTDYN, indicating which peripheral is handled by which device model.

8.2 VIOMAPPER

By integrating instrumentation and device-model execution directly into the JIT-translated execution path, FASTDYN minimizes the overhead associated with firmware analysis and peripheral handling, thereby achieving our second design goal, **G2: Efficient Emulation**.

VIO models and FASTDYN provide a very flexible way to rehost firmware images. However, adapting them to a new firmware image would still require substantial manual effort. In order to mitigate this, we designed VIOMAPPER. VIOMAPPER is a dynamic analysis plugin for FASTDYN, that generates emulation models for devices using VIO models and firmware. Figure 7 shows the workflow of VIOMAPPER. VIOMAPPER first runs the firmware in the passthrough mode while enabling the tracing capabilities of the firmware. Next, VIOMAPPER uses the **Encoder** to encode the trace in an annotated time automata. For each device, VIOMAPPER uses **CodeGen**, an LLM-powered module to generate a device model based on the interaction observed between the firmware and hardware. Lastly, we verify the newly generated model using the **Verifier** module to ensure the generated model can faithfully replay the I/O trace. In case we find a discrepancy, we use the **Root Cause Analysis (RCA)** module to refine the model. We describe the design of each component in the following sub-section.

The raw I/O traces generated by FASTDYN can easily reach gigabytes within minutes. Due to the nature of CPS and embedded systems, a substantial portion of these traces contains repeated behaviors. Firmware often performs the same sequences continuously during normal operation. Presenting such raw traces to any analysis framework is both inefficient and overwhelming. Moreover, raw traces lack semantic information: they record MMIO accesses but do not convey device roles, register semantics, or platform structure. Since the platform configuration is already known, we can generate a far more expressive and concise trace. To this end, Encoder conducts a number of analyses as described below:

Encoder. Encoder converts a raw MMIO execution trace into a symbolic peripheral automaton. It first performs *Peripheral Metadata Annotation* to map each read, write, and interrupt to its corresponding peripheral and register. Encoder then detects the onset of the dominant recurring operation to separate initialization from steady-state behavior using *Pattern Mining*. These recurring subsequences are transformed into a directed transition graph, which is normalized and merged to yield a compact automaton capturing the peripheral’s observed behavior. We describe the major components in detail below.

Peripheral Metadata Annotation: Encoder further enriches the automaton with architectural metadata by using System View Descriptor (SVD) files [3] to annotate each MMIO address with its symbolic name and device-level semantics, enabling direct mapping to Virtual I/O (VIO) models.

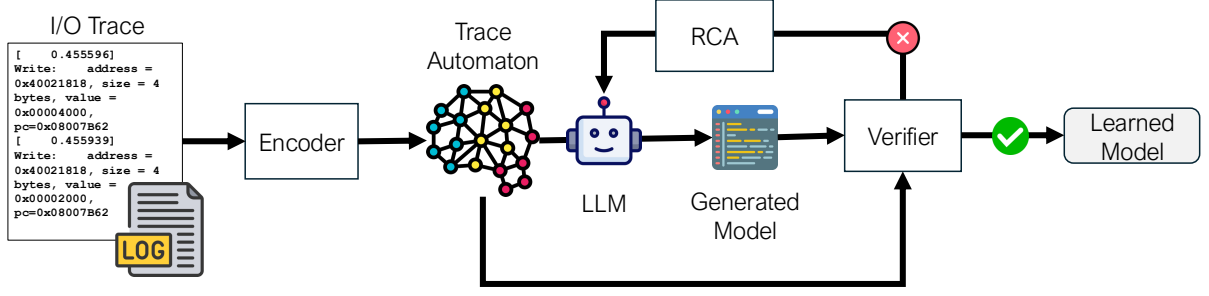


Figure 7: VIOMAPPER pipeline. The I/O traces from FASTDYN HITL mode are encoded into an automaton, which is used by CodeGen to generate a device model. The model is verified against the original I/O trace to ensure consistency.

Interrupt Handler Identification: Encoder also correlates interrupt events with the corresponding interrupt handlers, allowing it to identify the firmware’s interrupt-handling structure and distinguish asynchronous device-driven behavior from normal execution.

Pattern Mining: Encoder processes the raw I/O logs and produces a compact automaton that preserves the behavioral semantics of the original trace. It decomposes the trace into two categories: (1) repeating patterns and (2) linear, non-repeating sequences. This design is motivated by the observation that peripherals are typically exercised in two phases: an initial configuration phase followed by cyclic operational behavior. To detect repeated behavior, Encoder applies statistical and structural pattern-mining techniques, including N-gram analysis [6] and minimal cycle detection to extract the shortest repeating MMIO cycles. After extracting these patterns, the remaining trace consists of a mix of one-off initialization sequences and stable periodic cycles.

Entropy Analysis: Next, the encoder performs per-register Shannon entropy analysis to characterize the variability of register values. Registers with high entropy are typically data-bearing (e.g., sensor readings, FIFOs, DMA buffers), and our key insight is that such data registers depend heavily on specific application logic and thus cannot be modeled by a generic, one-size-fits-all rule. Identifying them is therefore essential to guide higher-level model synthesis.

Encoder uses these analyses to produce a concise timed automaton that captures the essential behavior of the firmware. As shown in section 10, Encoder can reduce the size of raw I/O traces by several orders of magnitude while preserving their semantic content. This reduction is essential for the CodeGen as LLMs have a limited context size; Encoder transforms an intractable context to a more tractable automaton that can be used by CodeGen.

CodeGen. CodeGen is the LLM-powered device-model generator of VIOMAPPER. It consumes the encoded automaton produced by Encoder and, using the device category and register annotations, selects the appropriate VIO APIs required to model the device’s behavior. Based on this information, CodeGen constructs a detailed contextual specification that is supplied to an LLM, which then synthesizes the corresponding device model. Appendix C details the prompt structure, refinement loop, and failure handling. The generated model is automatically compiled and subsequently evaluated by the Verifier to ensure consistency with the observed firmware behavior.

Component	SLOC
FASTDYN	7,996
VIOMAPPER	4,449

Table 3: Lines of Code (SLOC) Summary

Component	SLOC modified
Plugin	411
Core	615
Machine	964

Table 4: SLOC modified in QEMU by component.

Verifier. Because CodeGen relies on an LLM to synthesize device models, the output is susceptible to hallucinations or incorrect behaviors. To address this, VIOMAPPER employs the Verifier module, which validates each generated model before it is accepted. The Verifier constructs a dedicated test harness that replays the extracted I/O trace against the newly synthesized model. During this replay, we do not expect the same match with high entropy registers, as discussed earlier, these registers are domain-specific and are not generally produced by the CodeGen. By comparing the model’s responses with the original trace, the Verifier ensures that the generated implementation faithfully reproduces the observed device behavior and adheres to the semantics encoded in the automaton.

Root Cause Analysis (RCA). If the Verifier detects inconsistencies during trace replay, the RCA module examines the divergence between the generated I/O model and the original I/O trace to identify the underlying issue. Specifically, RCA analyzes the mismatched segment, locates the broken pattern or misconfigured transition, and classifies the type of error (e.g., incorrect state update, missing dependency, malformed timing). RCA then drives CodeGen to produce a targeted search-and-replace patch over the existing model rather than regenerate the entire C source. This saves tokens across iterations and, as the ablation in §10.4 confirms, is essential for complex peripherals.

VIOMAPPER is able to automatically generate the peripheral emulation, achieving our third goal, *G3 Low Manual Effort*.

9 IMPLEMENTATION

We describe the implementation of BOARDRUNNER per component. **FASTDYN.** FASTDYN is built as a QEMU TCG plugin against a QEMU 10.0.50-dev fork. We deliberately keep our QEMU-side changes

non-invasive: modifications are confined to the Cortex-M machine model, the NVIC, and a small set of plugin-API extensions that expose IRQ entry/exit and memory-modifier hooks. The SLOC breakdown is in Table 4. The bulk of FASTDYN—MMIO dispatch, the device-model registry, Modifiers, Virtual Instructions, and the record/replay backend—lives entirely in the plugin (Table 3), so FASTDYN tracks upstream QEMU with minimal friction. This contrasts with prior rehosting forks [33, 34, 50], which have diverged far enough from upstream that they lose ongoing architecture and performance improvements.

VIO Models. VIO Models are implemented as a standalone C library linked by every generated device model. The library currently provides high-level abstractions for USART, I²C, SPI, DMA, inter-peripheral Signals, PTY, FIFO, Ethernet (TAP), and block-device storage.

VIOMAPPER. VIOMAPPER is implemented in Python and uses the OpenAI API for LLM interactions.

As BOARDRUNNER accumulates support for more device classes, we expect the SLOC count to grow, primarily in the VIO library, which we view as the main surface of long-term engineering effort.

10 EVALUATION

We evaluate BOARDRUNNER on the five development boards listed in Table 5, spanning Cortex-M3, Cortex-M4F, and Cortex-M7 cores from two MCU vendors [2, 42–44, 49]. The boards expose a diverse set of MCU peripherals, including UART, SPI, I²C, GPIO, timers, DMA, ADC, and PWM. The specific firmware samples used in each evaluation are introduced with the corresponding research question.

For peripheral-model synthesis, we use OpenAI GPT-5.4 with medium reasoning effort and default settings (no fine-tuning, default temperature) [35]. The synthesized C models for all evaluated firmware are included in our released artifact, and the prompts are reproducible by re-running VIOMAPPER on each firmware’s HITL trace. All experiments run on Ubuntu 22.04.5 LTS on an AMD Ryzen 7 7800X3D (8 cores / 16 threads). We evaluate whether BOARDRUNNER provides a practical execution substrate for dynamic firmware security analysis. We first evaluate the performance and rehosting fidelity required by downstream analyses (RQ1–RQ2), then evaluate its use on real-world firmware and fuzzing (RQ3), and finally study the contribution of VIOMapper’s synthesis components (RQ4).

RQ1. How does FASTDYN’s emulation performance compare to existing rehosting strategies?

RQ2. Can BOARDRUNNER rehost SDK firmware end-to-end?

RQ3. Can BOARDRUNNER faithfully rehost real-world firmware and expose it to downstream fuzzing?

RQ4. How does each VIOMAPPER component contribute to model synthesis?

10.1 RQ1: FASTDYN rehosting performance

We evaluate FASTDYN’s execution overhead using both operation-level microbenchmarks and end-to-end macrobenchmarks. The microbenchmarks measure three operations commonly used during rehosting: firmware function bypass, host-side instrumentation, and HITL hardware access, while the macrobenchmarks measure end-to-end performance on a public firmware corpus.

Board	MCU	SDK / Example Source
NUCLEO-F103RB	STM32F103RB	STM32CubeF1 [45]
STM32F429I-DISC1	STM32F429ZI	STM32CubeF4 [46]
STM32F769I-EVAL	STM32F769NI	STM32CubeF7 [47]
NUCLEO-H753ZI	STM32H753ZI	STM32CubeH7 [48]
MAX78000FTHR	MAX78000	MSDK [1]

Table 5: Evaluation targets and the vendor SDK sources used to obtain SDK firmware workloads.

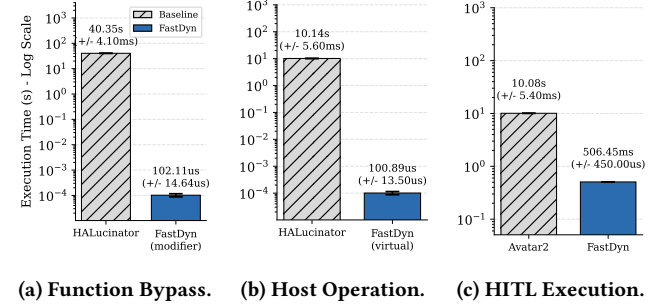


Figure 8: Isolation microbenchmarks on STM32F429I-DISC1 (log scale, mean $\pm$ SD, $n=100$ runs, $N=1000$ ops per run). (a) HALucinator’s function bypass versus FASTDYN’s inline modifier (b) HALucinator’s host operation versus FASTDYN’s plugin virtual (c) Avatar2’s OpenOCD bridge versus FASTDYN’s direct probe interface

Microbenchmarks. For the microbenchmarks, we evaluate the overhead of three operations commonly used during firmware rehosting: bypassing a firmware function, performing a host-side operation in response to firmware execution, and forwarding an MMIO access to physical hardware. Each microbenchmark executes a tight loop containing 1,000 instances of the operation under study. For *function bypass*, we skip the execution of a target firmware function. For *host operation*, we intercept a firmware function and record each invocation in a host-side log, representing analyses that require communication with or computation on the host. For *hardware access*, the firmware performs MMIO accesses that are forwarded to the physical device. We use HALucinator [9] as the baseline for function bypass and host-side operations. Since HALucinator does not support HITL execution, we use Avatar [33] as the baseline for hardware access.

Figure 8 shows the results. Thanks to our optimizations, FASTDYN bypasses 1000 functions in $102.11 \pm 14.64 \mu\text{s}$, compared with $40.35 \pm 0.004 \text{ s}$ for HALucinator, corresponding to a $3.95 \times 10^5 \times$ improvement. Similarly for host operations, FASTDYN completes the same sequence in $100.89 \pm 13.50 \mu\text{s}$, compared with $10.14 \pm 0.006 \text{ s}$ for HALucinator, corresponding to a $1.01 \times 10^5 \times$ improvement. For hardware access, FASTDYN completes the 1,000-operation sequence in $506.45 \pm 0.45 \text{ ms}$, compared with $10.08 \pm 0.005 \text{ s}$ for Avatar2, corresponding to a $19.90 \times$ improvement. These microbenchmarks intentionally minimize unrelated firmware execution so that the operation under study dominates runtime. The reported speedups therefore characterize favorable per-operation improvements rather than expected end-to-end application speedups.

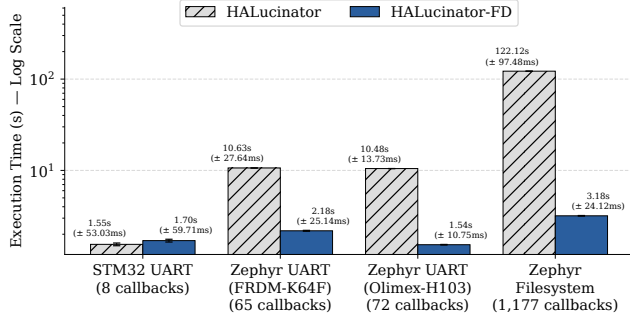


Figure 9: Execution time of HALucinator versus HALucinator-FD on the HALucinator corpus (log scale, mean $\pm$ SD, $N = 10$).

End-to-end macrobenchmarks. To evaluate how the microbenchmarks’ per-operation savings translate to real firmware, we run an end-to-end comparison on four workloads from HALucinator’s public corpus. These workloads span bare-metal and RTOS code on two vendors, including a short STM32 UART Hyperterminal demo, Zephyr UART echoes on NXP FRDM-K64F and Olimex STM32-H103, and a Zephyr LittleFS filesystem test on FRDM-K64F. HALucinator’s peripheral models are written in Python, so for a fair comparison we extend the FASTDYN plugin with an embedded CPython interpreter that hosts these models. We refer to this configuration as HALucinator-FD (FastDyn backend), and compare its execution time against unmodified HALucinator. Both configurations run byte-identical firmware images. Each workload runs 10 iterations, and each iteration launches a fresh QEMU process, so no JIT, interpreter, or plugin state persists across runs. We report mean $\pm$ standard deviation (SD).

Figure 9 reports the results alongside the runtime handler-callback count each workload triggers, recorded via HALucinator’s per-breakpoint counters. Speedup grows monotonically with callback count, consistent with the per-callback savings measured in isolation. The STM32 UART workload fires only 8 callbacks over a ~ 1.5 s bare-metal initialization sequence. Although FASTDYN’s per-callback cost is far lower than HALucinator’s (Figure 8), 8 callbacks are too few to recover HALucinator-FD’s fixed plugin-load and CPython initialization overhead, yielding a $0.91\times$ ratio. On the three multi-second Zephyr workloads, the same per-callback savings accumulate over 65 to 1,177 callbacks and dominate the fixed cost. This produces $4.87\times$, $6.82\times$, and $38.37\times$ speedups on the FRDM-K64F, Olimex-H103, and Filesystem workloads respectively. The Filesystem case shows the largest gain because each top-level filesystem call issues many handler callbacks, amplifying the per-callback savings measured in isolation. The average speedup across the four workloads is $12.74\times$.

10.2 RQ2: SDK firmware rehosting

To evaluate whether BOARDRUNNER can rehost SDK firmware, we run the 70-workload suite across the five boards in Table 5. Each workload is derived from a vendor SDK example and is paired with a functional check (Table 7). For each workload we (1) collect a hardware trace by running the firmware in passthrough mode through FASTDYN, (2) synthesize a peripheral model from the trace

Table 6: Comparison of BOARDRUNNER against naive baselines on NUCLEO-F103RB. Each cell reports the number of unique basic blocks executed by the firmware. FC marks functional correctness, TV marks trace verification.

Firmware	Passthrough	Null	Replay	BOARDRUNNER
GPIO (cbk)	191	101	191	191
UART (RX/TX)	415	352	415	415
I ² C (RX/TX)	438	304	438	438
SPI (RX/TX)	291	262	291	291
ADC+DMA	378	265	378	378
FC	✓	✗	✗	✓
TV	✓	✗	✓	✓

with VIOMAPPER, and (3) re-run the firmware in emulation using the generated model. A workload is successfully rehosted if Verifier accepts the model and the firmware passes its functional check. BOARDRUNNER rehosts all 70 workloads.

Comparison against naive baselines. While an end-to-end execution is necessary for successful rehosting, it is not sufficient to validate model correctness, since a deterministic replay of the recorded MMIO trace would produce the same outcome. To distinguish such trace memorization from genuine semantic modeling, we evaluate BOARDRUNNER against two naive baselines on NUCLEO-F103RB across five representative peripherals (GPIO, UART, I²C, SPI, and ADC+DMA): the Null model (§8.1.2) and the Replay model (§8.1.2). Table 6 reports the unique basic blocks reached under each mode, together with whether the firmware passes its functional check and whether the model’s MMIO output matches the recorded hardware trace.

While both naive baselines reach a non-trivial fraction of the basic-block count, neither satisfies the firmware’s functional check. The Null model allows the firmware to complete initialization but cannot drive any further activity, since its zero responses provide no peripheral state. The Replay model, by construction, reproduces the recorded basic-block count and trace exactly, yet still fails the functional check, since it cannot respond to inputs outside the recording. Only the BOARDRUNNER-generated model satisfies both functional correctness and trace verification. This observation also motivates our ablation study (§10.4), where functional correctness, rather than coverage, is treated as the primary metric of model quality.

Cost Evaluation. For cost evaluation, we ran VIOMAPPER on the NUCLEO-H753ZI. VIOMAPPER synthesized twelve peripheral models across eight firmware samples at a total LLM cost of \$10.05, averaging \$0.84 per model. Synthesis cost is primarily driven by two factors. First, more complex peripherals expose a larger configuration and operation space through their VIO interfaces, requiring VIOMAPPER to infer and map a larger set of target-specific MMIO behaviors. For example, a UART model exposes a relatively small set of operations such as configuring the baud rate and transmitting or receiving bytes. In contrast, a network interface may expose substantially more operations, such as configuring the MAC address and managing transmit/receive descriptor rings. Second, models

Table 7: Functional operations exercised by SDK firmware workloads (as modeled).

Peripheral	Functional Operations
UART	Receive or transmit a byte
SPI	Receive or transmit a byte
I ² C	Read or write a byte to a slave
GPIO	Read, set, or clear a pin; Execute callback after interrupt
ADC	Read an analog-to-digital conversion
DMA	Transfer from Peripheral MMIO register to RAM
Timer	Read counter value; Execute callback after interrupt
PWM	Configure PWM as an autonomous peripheral

that do not converge in the initial synthesis step require additional RCA-driven refinement iterations, further increasing token usage.

Table 9 illustrates the combined effect of these factors. USART converges in one iteration using 26.2 k tokens, SPI requires two iterations and 40.7 k tokens, while the compositional ADC+DMA workload requires seven iterations and 221.1 k tokens. Across these workloads, token usage therefore varies by roughly 8× between the simplest first-iteration model and the most refinement-intensive compositional model.

10.3 RQ3: Real-world firmware and fuzzing

We extend the evaluation to four real-world firmware targets (Table 8): an Ethernet stack (LwIP), a storage stack (FatFs over SDMMC), an IoT client (MQTT, built on Eclipse Paho Embedded C [15]), and an industrial fieldbus slave (Modbus RTU, built on FreeMODBUS). LwIP and FatFs run on STM32F769I-EVAL, MQTT runs on NUCLEO-H753ZI, and Modbus runs on NUCLEO-F103RB. Each firmware was first validated on the physical board before rehosting. Since BOARDRUNNER learns peripheral models from a hardware-in-the-loop trace, the corpus is bounded by board availability rather than software effort. The same constraint applies to prior trace-driven rehosters such as Pretender [21] and Conware [41].

Methodology. We measure each firmware under three configurations. For context, we include *No peripheral* mode as an execution floor rather than a competing rehosting baseline. This mode runs the firmware on unmodified QEMU with no FASTDYN plugin and no peripheral model. BOARDRUNNER (*Normal*) mode runs the firmware with the VIOMAPPER-generated model serving the protocol-coupled peripheral, while FASTDYN’s compositional passthrough handles the rest of the address space. BOARDRUNNER (*Fuzzing*) mode reuses the same model as a fuzzing harness, with the fuzzer mutating the input bytes the firmware reads through the model.

For each configuration, we report two metrics. *Basic Blocks (BB)* is the number of unique basic blocks the firmware executes during the run. *Functional Correctness (FC)* indicates whether the firmware passes its functional check: serving an HTTP page (LwIP), reading back a written file (FatFs), completing a publish-subscribe round-trip (MQTT), and completing Modbus request-response transactions (Modbus). Under fuzzing, FC additionally requires the model to remain stable for the duration of the campaign, so that benign traffic interleaved with mutated inputs continues to be served.

Results. Across all four firmware targets, BOARDRUNNER reproduces the functional behavior of the real device. *No peripheral* mode stalls at the first unsupported peripheral access in every case and

never reaches the application code. Three of the four models (FatFs, MQTT, Modbus) cover a single peripheral and converge from the MMIO trace alone. LwIP is the only target where the trace alone is not enough, since its DMA descriptors live in RAM and never appear as MMIO traffic. For this case, the user supplies a snapshot of the relevant memory region together with the descriptor element size, which Encoder forwards to CodeGen alongside the trace. All four models converge in six or fewer iterations.

Fuzzing follow-on. Once a model is verified, it can be reused as a fuzzing harness without any additional work: the same byte boundary VIOMAPPER learned is also where the fuzzer injects mutated input. Mutated inputs are produced through a generic backend interface, which currently supports LibAFL [19] and a modified version of AFLNET [36] that delivers stateful sequences of inputs directly to the model rather than through network sockets. For each firmware we run three independent 24-hour campaigns. The fuzzer mutates the protocol bytes the parser consumes: Ethernet frames for LwIP, MQTT frames for the Paho client, Modbus PDUs for FreeMODBUS, and SD command frames for FatFs. Because the HAL, driver, and middleware run unmodified, the parsers themselves are exercised by the fuzzer, including the regions historically associated with vulnerabilities in these stacks (e.g., length-field decoding, CRC validation, and PDU dispatch).

BB under fuzzing exceeds BB under normal execution on LwIP, MQTT, and Modbus, where the fuzzer drives each parser through paths the recorded run does not enter. FatFs does not show such gains, since its firmware has limited control flow. Across all four campaigns, the model itself remains stable, with no crashes, deadlocks, or desynchronizations on benign traffic interleaved with mutated input. Because the fuzzer drives protocol bytes never present in the recorded trace, this stability provides evidence that the synthesized models can support executions beyond those observed during trace collection. This experiment is not intended to evaluate a new fuzzing algorithm; rather, it tests whether BOARDRUNNER preserves sufficient peripheral and protocol semantics for existing fuzzers to exercise security-relevant firmware functionality under rehosting.

We did not discover any new vulnerabilities in the four firmware targets within the campaign budget. Our current fuzzing setup uses a basic AFL-style integration and treats the generated device model primarily as an input interface, rather than exploiting the richer semantics exposed by BOARDRUNNER. We discuss model-aware fuzzing opportunities enabled by these semantics in Section 11.

10.4 RQ4: Ablation study

We ablate VIOMAPPER’s four components in turn: the Encoder (A1), Root Cause Analysis (A2), VIO API definitions (A3), and Verifier feedback loop (A4). Each ablation runs on three peripherals from NUCLEO-H753ZI: USART, SPI with an external slave, and a compositional ADC+DMA workload. The compositional case is the stress test, since the ADC writes the DMA destination pointer through ADC MMIO and either model failing fails the entire run. We cap each ablation at 10 LLM iterations and consider a run converged only if the synthesized model passes both VIOMAPPER’s verification and the firmware’s functional check. Table 9 reports the results.

Table 8: Real-world firmware rehosting and fuzzing. BB: unique basic blocks executed. FC: passes functional check.

Firmware	MCU	Middleware	Modeled Interfaces	No Peripheral Mode		BOARDRUNNER (Normal)		BOARDRUNNER (Fuzzing)	
				BB	FC	BB	FC	BB	FC
LwIP HTTP server	STM32F769NI	lwIP (raw API)	ETH MAC + DMA	170	✗	1838	✓	2651	✓
FatFs / uSD	STM32F769NI	FatFs over SDMMC	SDMMC1	141	✗	1416	✓	1416	✓
MQTT (Paho)	STM32H753ZI	Eclipse Paho	USART3	29	✗	866	✓	892	✓
Modbus RTU slave	STM32F103RB	FreeMODBUS	USART2	157	✗	518	✓	623	✓

Table 9: Component ablation on NUCLEO-H753ZI. Each cell reports iterations and total LLM tokens; ✓/✗ indicate pass/fail within the 10-iteration cap. A4 reports a binary success only, since it disables the feedback loop.

Peripheral	Full VIOMAPPER		A1: No Encoder		A2: No RCA		A3: No VIO		A4: No Verifier	
	Iters	Tokens	Iters	Tokens	Iters	Tokens	Iters	Tokens	Success	Tokens
USART	✓ 1	26.2k	✓ 1	96.4k	–	–	✗ 3	90.6k	✓	26.2k
SPI	✓ 2	40.7k	✓ 1	145.2k	✓ 1	19.2k	✗ 2	33.0k	✗	16.3k
ADC+DMA	✓ 7	221.1k	✗ 2	1.066M	✗ 10	257.7k	✗ 10	282.9k	✗	55.7k

A1: Encoder. We bypass the Encoder and pass the raw MMIO trace directly to CodeGen. A1 still converges on USART and SPI, but at 3–4× the token cost of the full pipeline: USART takes 96.4 k tokens against 26.2 k baseline, and SPI takes 145.2 k against 40.7 k. The cause is that without entropy classification, the LLM models polling status bits as constants, the firmware loops indefinitely waiting for a never-changing flag, and the resulting emulation trace grows beyond what the prompt can carry. On the compositional ADC+DMA workload, this same pathology pushes the joint correction prompt past the API’s 922 k input-token limit and the pipeline halts after initial synthesis.

A2: RCA. We disable Root Cause Analysis, so the LLM must regenerate the model on each iteration without targeted patches. On USART, baseline converges at iteration 1, leaving A2 with no broken iteration to correct. On SPI, regeneration is more efficient than patching, converging in one iteration at 19.2 k tokens against baseline’s two iterations and 40.7 k. The pattern reverses on the compositional ADC+DMA workload, where regenerating two coupled models from a generic correction prompt fails to converge within the cap. RCA matters when the model space is large enough that regeneration loses information across iterations.

A3: VIO. We strip the VIO API definitions, leaving the LLM with only bare MMIO callbacks. Without protocol-level abstractions, the LLM falls back to trace memorization, exhibiting the same failure mode we observed for the Replay model in §10.2. USART and SPI both pass VIOMAPPER’s verification but fail the firmware’s functional check, since the synthesized model can only replay recorded values and cannot respond to fresh inputs the firmware generates at runtime. ADC+DMA fails outright within the cap. Without VIO, the LLM has no path to a semantically faithful model.

A4: Verifier. We disable the Verifier’s feedback loop, leaving each run with a single one-shot synthesis. USART is simple enough that the LLM converges on the first attempt, so A4 succeeds. SPI and ADC+DMA do not single-shot converge, and without the feedback

loop to correct their broken first attempts, both fail. The feedback loop is essential as soon as peripheral complexity exceeds what the LLM can model in one attempt.

11 DISCUSSION

Trace Coverage. VIOMAPPER learns from HITL traces, so model fidelity is bounded by the behaviors they capture. Although the LLM may infer behavior beyond the observed trace, trace-replay verification cannot validate such behavior. Our fuzzing experiments provide evidence that the models support executions beyond the recorded traces, but do not establish correctness for all unseen peripheral states or interactions.

Semi-Automatic Rehosting. BOARDRUNNER automates target-specific MMIO-to-VIO mapping for peripheral classes already supported by the VIO library. The same reusable VIO model can be mapped to different MCU register interfaces by VIOMAPPER, while supporting a previously unsupported peripheral class requires implementing a new VIO model once. Incomplete traces or platform metadata may also require manual refinement. Thus, BOARDRUNNER is currently semi-automatic, with manual effort decreasing as more peripherals are added to the VIO library.

Portability. Our current implementation targets ARM Cortex-M MCUs and uses CMSIS SVD files as the default source of register metadata, but neither VIO nor VIOMAPPER fundamentally depends on SVD or ARM. Equivalent structured metadata may come from Device Tree, vendor documentation, or other platform descriptions; when such metadata is incomplete, additional analyst effort is required. Extending BOARDRUNNER to RISC-V or other architectures additionally requires FASTDYN support for the corresponding CPU backend. These are primarily engineering limitations rather than restrictions of the VIO abstraction.

Model-Aware Fuzzing. Our current fuzzing evaluation uses a basic AFL-style integration that treats the generated device model primarily as an input interface. However, BOARDRUNNER exposes

richer semantics, including peripheral operations, configuration state, data paths, and inter-peripheral dependencies, which could enable domain-aware mutation strategies and restrict fuzzing to hardware-feasible behaviors. For example, configuration and data registers could use different mutation strategies, while structured peripheral inputs could be paired with grammar-aware generators. We leave such model-aware fuzzing to future work.

12 CONCLUSION

We presented BOARDRUNNER, a compositional firmware-rehosting framework uniting reusable high-fidelity device models with FAST-DYN's efficient inline execution. Across five MCUs, BOARDRUNNER minimizes manual effort while preserving the exact peripheral semantics required for faithful execution, achieving a 12.74× average speedup over HALucinator on macrobenchmarks. Crucially, real-world evaluations confirm that BOARDRUNNER-generated models seamlessly expose complex firmware interfaces to downstream fuzzing. BOARDRUNNER delivers a practical, high-performance execution substrate for dynamic security analysis of hardware-coupled firmware.

13 ACKNOWLEDGMENT

This work is based on research sponsored by DARPA under agreement number FA8750-24-2-0500. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work was also supported in part by startup funds from the Department of Computer Science and Engineering at The Pennsylvania State University.

REFERENCES

- [1] Analog Devices. 2025. MSDK: Maxim/Analog Devices Microcontroller Software Development Kit. <https://github.com/analogdevicesinc/msdk>. Accessed: 2025-11-09.
- [2] Analog Devices. 2026. MAX78000EVKIT Evaluation Kit for the MAX78000. <https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/max78000evkit.html>. Accessed: 2026-01-20.
- [3] Arm Limited. 2023. SVD Description (*.svd) Format. https://arm-software.github.io/CMSIS_5/SVD/html/svd_Format_pg.html [Online; accessed 2025-11-09].
- [4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. <https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator>
- [5] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 746–759.
- [6] William B. Cavnar and John M. Trenkle. 1994. N-Gram-Based Text Categorization. In *Proceedings of the 3rd Annual Symposium on Document Analysis and Information Retrieval (SDAIR-94)*. Las Vegas, NV, 161–175.
- [7] Check Point Research Team. 2020. The Dark Side of Smart Lighting: Check Point Research Shows How Business and Home Networks Can Be Hacked from a Lightbulb. Check Point Blog. <https://blog.checkpoint.com/security/the-dark-side-of-smart-lighting-check-point-research-shows-how-business-and-home-networks-can-be-hacked-from-a-lightbulb/>
- [8] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. 2024. MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5359–5376.
- [9] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1201–1218. <https://www.usenix.org/conference/usenixsecurity20/presentation/clements>
- [10] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide security testing of Real-World embedded systems software. In *27th USENIX security symposium (USENIX security 18)*. 309–326.
- [11] Ang Cui, Michael Costello, and Salvatore J. Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings of NDSS 2013*. <https://expertknobtwiddlers.com/publications/ndss-2013.pdf>
- [12] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. 463–478.
- [13] Jyoti Deogirikar and Amarsinh Vidhate. 2017. Security attacks in IoT: A survey. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 32–37.
- [14] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (Los Angeles, CA, USA) (PPREW-5)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. <https://doi.org/10.1145/2843859.2843867>
- [15] Eclipse Foundation. 2026. Eclipse Paho MQTT Embedded C Client. <https://github.com/eclipse-paho/paho.mqtt.embedded-c>. Accessed: 2026-04-18.
- [16] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing embedded systems using debug interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1031–1042.
- [17] Guy Farrelly, Michael Chesser, and Damith C Ranasinghe. 2023. Ember-IO: Effective firmware fuzzing with model-free memory mapped IO. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 401–414.
- [18] Zhongwen Feng and Junyan Ma. 2024. TWFuzz: Fuzzing embedded systems with three wires. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 107–118.
- [19] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (Los Angeles, CA, USA) (CCS '22)*. ACM, New York, NY, USA, 1051–1065.
- [20] Samuel Gibbs. 2015. Jeep owners urged to update their cars after hackers take remote control. *The Guardian*. <https://www.theguardian.com/technology/2015/jul/21/jeep-owners-urged-update-car-software-hackers-remote-control> Accessed: 2026-08-18.
- [21] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 135–150. <https://www.usenix.org/conference/raid2019/presentation/gustafson>
- [22] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2245–2262.
- [23] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 321–338. <https://www.usenix.org/conference/usenixsecurity21/presentation/johnson>
- [24] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *International Conference on Emerging Security Information, Systems and Technologies*. Nice, France, 21–26.
- [25] Markus Kammerstetter, Christian Platzter, and Wolfgang Kastner. 2014. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 329–340.
- [26] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*.
- [27] Chongqing Lei, Zhen Ling, Yue Zhang, Yan Yang, Junzhou Luo, and Xinwen Fu. 2024. A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers. In *33rd USENIX Security Symposium (USENIX Security 24)*. 7085–7102.
- [28] Wengqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. 2021. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. In *Proceedings 2021 Network and Distributed System Security Symposium (Virtual)*. Internet Society, Reston, VA.
- [29] Wengqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μ AFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*. 1–12.

- [30] Lulu Liang, Kai Zheng, Qiankun Sheng, and Xin Huang. 2016. A denial of service attack method for an iot system. In *2016 8th international conference on Information Technology in Medicine and Education (ITME)*. IEEE, 360–364.
- [31] Zheyu Ma, Qiang Liu, Zheming Li, Tingting Yin, Wende Tan, Chao Zhang, and Mathias Payer. 2025. Truman: Constructing Device Behavior Models from OS Drivers to Fuzz Virtual Devices. In *32nd Annual Network and Distributed System Security Symposium, NDSS*.
- [32] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. 2024. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5323–5340.
- [33] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar²: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*.
- [34] Anh Quynh Nguyen and Hoang Vu Dang. 2015. Unicorn: Next Generation CPU Emulator Framework. Briefing, Black Hat USA 2015. <https://blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf>
- [35] OpenAI. 2026. Introducing GPT-5. <https://openai.com/index/introducing-gpt-5/>. Accessed: 2026-01-20.
- [36] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465. <https://doi.org/10.1109/ICST46399.2020.00062>
- [37] Ruben Santamarta. 2019. Arm IDA and Cross Check: Reversing the Boeing 787’s Core Network. Black Hat USA. <https://www.ioactive.com/event/black-hat-usa-2019-ruben-santamarta/>
- [38] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1239–1256.
- [39] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2023. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2885–2902.
- [40] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Fimalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society. <https://doi.org/10.14722/ndss.2015.23294>
- [41] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. 2021. Conware: Automated modeling of hardware peripherals. In *Proceedings of the 2021 ACM Asia conference on computer and communications security*. 95–109.
- [42] STMicroelectronics. 2026. 32F429IDISCOVERY | Product. <https://www.st.com/en/evaluation-tools/32f429idiscovery.html>. Accessed: 2026-01-20.
- [43] STMicroelectronics. 2026. NUCLEO-F103RB | Product. <https://www.st.com/en/evaluation-tools/nucleo-f103rb.html>. Accessed: 2026-01-20.
- [44] STMicroelectronics. 2026. NUCLEO-H753ZI | Product. <https://www.st.com/en/evaluation-tools/nucleo-h753zi.html>. Accessed: 2026-01-20.
- [45] STMicroelectronics. 2026. STM32CubeF1. <https://www.st.com/en/embedded-software/stm32cubef1.html>. Accessed: 2026-01-20.
- [46] STMicroelectronics. 2026. STM32CubeF4. <https://www.st.com/en/embedded-software/stm32cubef4.html>. Accessed: 2026-01-20.
- [47] STMicroelectronics. 2026. STM32CubeF7. <https://www.st.com/en/embedded-software/stm32cubef7.html>. Accessed: 2026-01-20.
- [48] STMicroelectronics. 2026. STM32CubeH7. <https://www.st.com/en/embedded-software/stm32cubeh7.html>. Accessed: 2026-01-20.
- [49] STMicroelectronics. 2026. STM32F769I-EVAL | Product. <https://www.st.com/en/evaluation-tools/stm32f769i-eval.html>. Accessed: 2026-01-20.
- [50] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares. In *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*.
- [51] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2007–2024. <https://www.usenix.org/conference/usenixsecurity21/presentation/zhou>
- [52] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. 2022. 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*. 3269–3283.

A OPEN SCIENCE

All artifacts needed to reproduce this paper’s results are available anonymously. BOARDRUNNER and FASTDYN are available at [//anonymous.4open.science/r/Boardrunner-Fastdyn, LibHW, the hardware-backend library, is available at <https://anonymous.4open.science/r/libhw>, and the forked QEMU that FASTDYN builds against is available at <https://anonymous.4open.science/r/qemu-fork>. The BOARDRUNNER/FASTDYN repository contains the rehosting framework, the VIOMAPPER synthesis pipeline, the generated peripheral models, the firmware binaries and HITL traces for every evaluated workload, and the evaluation scripts. Reproduction instructions are under docs/. Model synthesis used OpenAI GPT-5.4 at medium reasoning effort. Because the LLM is non-deterministic, we archive every generated peripheral model so that downstream evaluation can be reproduced without API access. Third-party middleware \(LwIP, FatFs, Eclipse Paho MQTT, FreeMODBUS\) and fuzzing engines \(LibAFL, AFLNET\) are cited to their upstream sources rather than redistributed. Upon acceptance, we will de-anonymize the repositories and release them under a permissive open-source license.](https://anonymous.4open.science/r/Boardrunner-Fastdyn, LibHW, the hardware-backend library, is available at https://anonymous.4open.science/r/libhw, and the forked QEMU that FASTDYN builds against is available at https://anonymous.4open.science/r/qemu-fork. The BOARDRUNNER/FASTDYN repository contains the rehosting framework, the VIOMAPPER synthesis pipeline, the generated peripheral models, the firmware binaries and HITL traces for every evaluated workload, and the evaluation scripts. Reproduction instructions are under docs/. Model synthesis used OpenAI GPT-5.4 at medium reasoning effort. Because the LLM is non-deterministic, we archive every generated peripheral model so that downstream evaluation can be reproduced without API access. Third-party middleware (LwIP, FatFs, Eclipse Paho MQTT, FreeMODBUS) and fuzzing engines (LibAFL, AFLNET) are cited to their upstream sources rather than redistributed. Upon acceptance, we will de-anonymize the repositories and release them under a permissive open-source license.</p>
</div>
<div data-bbox=)

B ETHICAL CONSIDERATIONS

Dual-use. BOARDRUNNER is a defensive firmware rehosting framework intended for security analysis through emulation and fuzzing. Any rehosting tool can in principle be used to locate vulnerabilities in deployed firmware, but the underlying techniques are already public, and BOARDRUNNER lowers the cost of legitimate analysis for vendors, auditors, and researchers.

Subjects and targets. This work involves no human subjects or personal data, so institutional review is not applicable. All evaluation runs on vendor evaluation boards owned by the authors (Table 5) and on publicly available firmware shipped with those boards’ SDKs. Our real-world targets (LwIP, FatFs, MQTT, and Modbus RTU) are widely-deployed open-source middleware. We do not target any specific product or deployment, and the RQ3 fuzzing campaign runs against reference implementations rather than live systems.

Responsible disclosure. Any vulnerability identified in upstream components through continued use of BOARDRUNNER will be reported to the affected maintainers under coordinated disclosure before public release.

C LLM PROMPTS AND REFINEMENT.

This appendix documents VIOMAPPER’s CodeGen prompts and the RCA-driven revision loop.

C.1 Initial Prompt Composition

The initial CodeGen prompt is assembled from five blocks; the boxes below use a running USART3 example on NUCLEO-H753ZI. **Role and task framing.** Identifies the LLM as an expert on the target MCU family and peripheral, and states the synthesis task.

Role & task framing

Template.

You are an expert reverse engineer specializing in embedded systems and writing C emulation for peripherals. You have read the reference manual for {mcu_family} with special familiarity with the {peripheral_name} peripheral. Your task is to analyze the following summary of MMIO trace data and generate a complete C device model.

Example.
... reference manual for STM32H753x with special familiarity with usart3 peripheral.

Available APIs. The VIO and framework APIs the LLM may invoke, filtered by device category. A hard-blocker policy requires the LLM to stop and propose a signature.

Available APIs & missing-API policy

Template.

```
## Available APIs
You must use the following APIs to construct the device model.
- `(return_type) {api_name}({parameters})`:
  {one-line description}
...

### Missing API Policy (HARD BLOCKER)
If the model requires an API that is not listed above, you
MUST stop and propose a precise signature
(api_<domain>_<action>, inputs, output, description) rather
than generate a placeholder implementation.
```

Example.

```
- `void api_pty_write_req(int fd, uint8_t value)`:
  Writes the byte to a host-side pseudo-terminal.
- `int api_pty_read_nonblock(int fd, uint8_t *buff)`:
  Reads a single byte from the PTY in non-blocking mode.
```

Framework rules. Three hardened rules addressing recurring LLM mistakes: absolute addressing (subtract peripheral base from the callback address), callback signatures (`_init` returns a state-struct pointer threaded back as `opaque`), and stateful emulation (no trace memorization).

Critical framework rules

Template.

```
## Critical Framework Rules
### 1. Absolute Addressing
The framework passes absolute guest physical addresses to
the model's read/write callbacks. You MUST subtract the
peripheral's base address before comparing against register
offsets.

### 2. Callback Function Signatures
Your _init function MUST return a pointer to your state
struct; the framework passes this pointer as opaque to
every subsequent _read and _write call.

### 3. Stateful Emulation
Track peripheral state explicitly. Do not memorize trace
values or return canned bytes.
```

Example (rule 1 illustration in prompt).

```
hwaddr offset = addr - MY_PERIPH_BASE;
switch (offset) {
  case 0x00: /* first register */ ...
}
```

Encoded trace. Encoder's timed automaton with SVD-annotated register names (§8.2): initialization phase, dominant recurring cycle, rare transitions, RMW and write-poll patterns, per-register entropy, and per-vector ISR traces.

Encoded trace

Template.

```
## Detected Runtime Loops (loop_pattern*.txt)
# Pattern #{n} (found {count} times)
[{t}] {op} to {periph}->{reg} ({addr}) value={val}, pc={pc}
...
```

```
## Register Entropy Analysis (entropy.txt)
- Register {name}: {LOW|MEDIUM|HIGH} - Entropy = {bits}

## ISR Analysis (isr_analysis.txt)
## Repeating ISR -- NVIC IRQn={n} ##
# Total Interrupts on this Vector: {count}
[{t}] INTERRUPT on {periph}, NVIC IRQn={n}
  - Inferred ISR Trace:
    [{t}] {op} to {periph}->{reg} ({addr}) value={val}
    ...

Example.
# Pattern #2 (found 182 times)
[11.503746] INTERRUPT on USART3, NVIC IRQn=39
[11.504009] READ to USART3->ISR (0x4000481C) value=0x6000D0

- Register CR1: LOW (status/control) - Entropy = 1.88 bits
```

Output-format contract. Requires three output sections — a summary, a register analysis, and the complete C source in a single fenced ````c ... ```` block — matching a skeleton that names the required `_init/_read/_write` symbols.

Output-format contract

Template.

```
## Required Output Format:
### 1. High-Level Summary
{one-paragraph summary of the peripheral's purpose}
### 2. Register Analysis
{bulleted list of important registers and their functions}
### 3. C Device Model Source Code
```c
static {periph}State g_{periph};
uint64_t {periph}_read(void *opaque, hwaddr addr,
 unsigned size);

void {periph}_write(void *opaque, hwaddr addr,
 uint64_t value, unsigned size);

void* {periph}_init(ConfigSection* model_info);
```

Example.
static USART3State g_usart3;
uint64_t usart3_read(void *opaque, hwaddr addr,
                    unsigned size) { ... }
```

C.2 Revision Prompt

When the Verifier rejects a model on trace replay, RCA constructs a fresh revision prompt and each iteration is user-approved. The revision prompt reuses the initial prompt's static blocks (APIs, framework rules) and replaces the rest with three revision-specific ones (subsection C.2): a task framing that says *correct* instead of *generate*, a *Model Under Correction* block that embeds the current C source, and an analysis section carrying RCA's structured analysis of the divergence. The output-format contract switches to search-and-replace patches, which (as noted for RCA in §8.2) cuts token cost every iteration after the first.

Revision prompt (blocks that differ from initial)

Template.

| | |
|-------------------------|--|
| Task framing: | Your task is to analyze the following summary of MMIO trace data and correct the existing C device model. |
| Model Under Correction: | // FILE: {model_filename}
{current C source of the model being revised, embedded in a fenced c block} |
| Analysis data: | {RCA's structured analysis of the divergence: located broken pattern, classified error type (e.g., incorrect |

```

state update, missing dependency,
malformed timing})

Output format: 1. Previous Model Failure Analysis
                2. Register Analysis
                3. SEARCH/REPLACE blocks:
                  // FILE: {model_filename}
                  <<<<<<< SEARCH
                  [exact old code]
                  =====
                  [new code]
                  >>>>>>> REPLACE

```

Example (SEARCH/REPLACE rules stated in the output-format contract).

1. The SEARCH text must exactly match the current file, character for character, including whitespace and indentation.

Output-side failures (unparseable output, patch-application errors, or compile errors) reuse the same structure with the error text substituted for the RCA analysis.

Failure re-prompt (compile-error example)

Template.

```

Task framing:    Correct the existing C device model.

Model Under     // FILE: {model_filename}
Correction:      {current C source that failed}

Error message:  {parse, patch-application, or compile
                  error captured by the pipeline}

Output format:  SEARCH/REPLACE blocks (as above)

```

Example (compile error).

```

usart3_model.c:42:5: error: 'USART3State' has no
  member named 'rx_ready'
42 |         s->rx_ready = 1;
   |           ^~

```