Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.065672

ARTICLE

ADFEmu: Enhancing Firmware Fuzzing with Direct Memory Access (DMA) Input Emulation Using Concolic Execution and Large Language Models (LLMs)

Yixin Ding[1], **Xinjian Zhao**[1], **Zicheng Wu**[1], **Yichen Zhu**[2], **Longkun Bai**[2] **and Hao Han**[2,*]

[1]Information and Telecommunication Branch, State Grid Jiangsu Electric Power Co., Ltd., Nanjing, 210024, China
[2]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210024, China

*Corresponding Author: Hao Han. Email: hhan@nuaa.edu.cn

Received: 19 March 2025; Accepted: 26 June 2025; Published: 30 July 2025

ABSTRACT: Fuzz testing is a widely adopted technique for uncovering bugs and security vulnerabilities in embedded firmware. However, many embedded systems heavily rely on peripherals, rendering conventional fuzzing techniques ineffective. When peripheral responses are missing or incorrect, fuzzing a firmware may crash or exit prematurely, significantly limiting code coverage. While prior re-hosting approaches have made progress in simulating Memory-Mapped Input/Output (MMIO) and interrupt-based peripherals, they either ignore Direct Memory Access (DMA) or handle it oversimplified. In this work, we present ADFEmu, a novel automated firmware re-hosting framework that enables effective fuzzing of DMA-enabled firmware. ADFEmu integrates concolic execution with large language models (LLMs) to semantically emulate DMA operations and synthesize peripheral input sequences intelligently. Specifically, it learns DMA transfer patterns from the firmware's context and employs guided symbolic execution to explore deeper and more diverse execution paths. This approach allows firmware to operate stably without hardware dependencies while achieving higher fidelity in emulation. Evaluated on real-world embedded firmware samples, ADFEmu achieves a 100% re-hosting success rate, improves total execution path exploration by 5.31%, and triggers more crashes compared to the state-of-the-art. These results highlight ADFEmu's effectiveness in overcoming long-standing limitations of DMA emulation and its potential to advance automated vulnerability discovery in peripheral-rich embedded environments.

KEYWORDS: Fuzz testing; firmware rehosting; DMA; concolic execution; LLMs

1 Introduction

In recent years, the rapid advancement of the Internet of Things (IoT) has led to a significant increase in the number of embedded devices, resulting in substantial security vulnerabilities. The widespread use of these devices across various sectors means that their security issues can directly impact personal privacy and the safety of critical infrastructure. However, the firmware of embedded devices often features a simple design and limited resources, making them vulnerable to attacks [1]. As such, conducting security research on embedded firmware is crucial to ensuring the stability of the entire IoT ecosystem. In the security research of embedded devices, dynamic firmware analysis has emerged as a popular approach due to its high efficiency and low false-positive rate. To perform dynamic analysis on firmware, the firmware program needs to be executed in practice. However, due to the high dependency of firmware on the actual hardware peripherals of the device during execution, the current technology for firmware emulation in embedded devices still faces challenges such as requiring extensive manual analysis and a lack of general applicability.

Recent studies [2] have introduced firmware re-hosting methods that allow firmware to run in virtualized environments, thereby mitigating hardware dependencies. Nonetheless, existing firmware re-hosting techniques [3,4] lack adequate support for handling Direct Memory Access (DMA) inputs, significantly limiting the types of devices that can be emulated and the extent of firmware code coverage.

This lack of automated and generalized DMA handling greatly restricts the ability of re-hosted environments to faithfully execute firmware, often leading to low code coverage and incomplete behavioral analysis. Existing frameworks typically either ignore DMA interactions altogether or require labor-intensive manual modeling of DMA behavior, which hinders scalability and limits applicability across diverse firmware targets.

To overcome these challenges, we propose ADFEmu, a dynamic symbolic execution-based re-hosting framework for embedded firmware, which provides optimized support for handling DMA inputs beyond existing re-hosting techniques. ADFEmu enables the emulation of common embedded firmware without the need for actual hardware devices by dynamically determining DMA input calls and solving for the expected execution paths of firmware code using Dynamic Symbolic Execution (DSE).

To address the limitations of traditional DSE, ADFEmu leverages LLMs to guide symbolic path exploration with semantic awareness, allowing it to prioritize meaningful execution paths and synthesize plausible DMA inputs during fuzz testing. This hybrid design significantly improves the efficiency and effectiveness of firmware code coverage and vulnerability discovery, without requiring any manual effort. The core design of the ADFEmu framework allows for the provision of anticipated DMA inputs during firmware fuzz testing, thus facilitating the exploration of more firmware code logic via DMA inputs.

We evaluated our method on real firmware targets and fuzz-tested 10 open source firmware using ADFEmu. The experimental results indicate that, compared to existing methods, our approach achieves higher code coverage, uncovers more execution paths, and triggers more crashes.

To clearly state the objectives of this work, our research aims to:

(1) develop an automated and general approach to enable accurate DMA input emulation in firmware re-hosting;
(2) combine DSE and LLM techniques to improve the efficiency and scalability of firmware path exploration and fuzzing;
(3) demonstrate the effectiveness of the proposed ADFEmu framework through experiments on real-world DMA-enabled embedded firmware.

In summary, our research offers the following contributions:

(1) We propose a new approach for optimizing the emulation of firmware peripherals, specifically focusing on DMA input. The method integrates dynamic symbolic execution technology with fuzzing techniques, and incorporates LLM technology to further enhance path exploration efficiency and code coverage, with the goal of dynamic analysis for embedded firmware.
(2) We designed and implemented ADFEmu, a firmware re-hosting framework for fuzzing DMA-enabled embedded firmware.
(3) Through experiments on real firmware, we show that ADFEmu outperforms existing works in fuzzing DMA-enabled firmware. ADFEmu achieved higher code coverage and uncovered more execution paths.

The subsequent sections of this paper are structured as follows:

Section 2 introduces foundational knowledge essential for comprehending ADFEmu. Sections 3 delve into the design and implementation of ADFEmu. Section 4 outlines the evaluation results. Section 5 discusses the challenges faced in the paper and outlines future work directions. Section 6 summarizes the entire article.

2 Background and Related Work

To support the proposed approach, this section provides essential background on characteristics of embedded firmware, firmware re-hosting techniques, and the importance of DMA interactions in firmware execution. As shown in Table 1, existing peripheral emulation techniques vary significantly in their objectives, supported peripheral types, and scalability. Notably, most prior works are limited to Memory-Mapped Input/Output(MMIO) peripherals and fail to model DMA interactions effectively. The specific introduction content is as follows:

Table 1: Comparison of peripheral emulation techniques

Scheme	Objective	Peripheral types	Scope	Limitations
HALucinator [5]	Replaces HAL functions with manual handling.	MMIO	Specific HALs	Relies on source code.
IEmu [6]	Enhances fuzzing via invalidity-guided knowledge reasoning.	MMIO	ARM Cortex-M	Cannot handle complex non-knowledge-guided scenarios.
Fuzzware [7]	Improves fuzzing through precise MMIO modeling.	MMIO	ARM Cortex-M/RISC-V	Only supports MMIO peripherals.
P2IM [8]	Models MMIO interactions to simulate hardware peripherals.	MMIO	ARM Cortex-M	Cannot handle DMA inputs.
DICE [9]	Simulates DMA input channels to enable DMA interaction.	MMIO/DMA	ARM Cortex-M	Random input generation lowers DMA simulation accuracy.
ADFEmu (Ours)	Enhances DMA simulation via symbolic execution.	MMIO/DMA	ARM Cortex-M	Symbolic execution causes higher overhead and path explosion.

2.1 Embedded Firmware and Firmware Re-Hosting

Embedded devices perform specific tasks through driver software, commonly referred to as firmware. Unlike desktop software, firmware can directly interact with underlying hardware. According to Muench's description [10], embedded firmware can be categorized into three types based on the type of operating system used: devices based on general-purpose operating systems (such as real-time Linux), devices based on embedded operating systems (like VxWorks and ZephyrOS), and devices without an operating system, which use monolithic firmware. Monolithic firmware devices operate through control loops, usually processing external events triggered by interrupts from peripherals. While security research has predominantly focused on firmware for devices based on general-purpose operating systems [11,12,13], there is also growing attention on monolithic firmware to analyze the behavior of various hardware devices.

Firmware re-hosting refers to the process of extracting firmware from a physical device and constructing a virtual environment that accurately emulates the execution of the firmware, including its interaction with hardware peripherals. This process aims to replicate the behavior of the original device in a software environment, thereby facilitating vulnerability analysis and testing. Peripheral interaction in firmware can be generally classified into three categories: MMIO, interrupts, and direct memory access (DMA). MMIO is the most common mechanism and is widely supported by existing emulation frameworks.

Several prior works have proposed techniques to improve the accuracy and scalability of firmware re-hosting. Clements et al. [5] proposed HALucinator, a system that enables scalable emulation and analysis of embedded firmware by leveraging hardware abstraction layers (HAL) and reusable high-level emulation functions. However, HALucinator relies on the availability of HAL, which limits its applicability to firmware samples that lack or do not expose HAL components. To address interrupt modeling, Wei et al. [6] introduced IEmu, a system that automatically extracts interrupt trigger rules by analyzing redundant check mechanisms in firmware binaries. While IEmu improves dynamic analysis effectiveness, it still performs worse than manual-specification-based tools such as SEmu, and its coarse-grained handling of MMIO data registers may hinder path diversity and reduce overall code coverage. Scharnowski et al. [7] proposed a novel firmware fuzz testing method called FUZZWARE, which improves the effectiveness of fuzz testing by accurately modeling MMIO and automatically eliminating input redundancy. However, FUZZWARE does not automatically handle DMA transfers, which limits its ability to analyze certain complex firmware. Feng et al. [8] developed P2IM, a technique that constructs processor-peripheral interface models to enable hardware-independent and scalable fuzz testing of firmware. Although P2IM achieves high effectiveness in MMIO-based emulation, it lacks support for DMA, limiting its applicability to firmware that depends on direct memory access.

These representative approaches have significantly advanced firmware re-hosting, especially in modeling MMIO and interrupts. However, most fail to address DMA, which is critical in many real-world embedded systems. Ignoring DMA may lead to incomplete emulation or missed execution paths. Our work seeks to bridge this gap by automatically modeling DMA behavior in firmware re-hosting, enabling more complete and accurate emulation of embedded systems.

2.2 DSE and Large Language Models

In recent years, the rapid development of large language models has introduced new technical solutions to research in IoT security and embedded system security. For instance, researchers have already applied these models in fields like symbolic execution and fuzzing. For example, Chen et al. [14] proposed an automated modeling method based on LLMs to resolve unknown functions in symbolic execution of WebAssembly programs, improving the security and reliability of software used in medical consumer electronics. Wang et al. [15] introduced LLM-Sym, a prototype symbolic execution engine that integrates LLMs with the Z3 SMT solver to solve path constraints more efficiently. In the domain of fuzzing, Jiang et al. [16] discussed five major challenges in applying LLMs to fuzzing and proposed effective solutions, validated through their application to fuzz testing of database management systems. Xia et al. [17] presented Fuzz4All, a general-purpose fuzzing tool leveraging LLMs to support multiple programming languages and generate diverse inputs in an LLM-driven fuzzing loop, which helped discover vulnerabilities across different systems.

Although these works highlight the promise of LLMs in static and dynamic program analysis, the integration of LLMs with DSE particularly in the context of embedded firmware remains underexplored. DSE is a program analysis technique that combines concrete execution with symbolic execution. Dynamically analyzes and generates symbolic constraints during the actual execution of a program. Unlike traditional symbolic execution, DSE makes the analysis process more flexible and efficient by incorporating concrete inputs while the program runs, particularly when dealing with complex control flows and non-static data structures. Cha et al. [18] proposed a method called ParaDySE, which enhances dynamic symbolic execution by automatically learning search heuristics to improve code coverage and vulnerability detection capabilities. However, the benchmark programs used in their experiments are limited in number, which may not fully represent the performance of search heuristics across all types of program. Jaffar et al. [19] introduced a new interpolation algorithm and integrated it into the KLEE system, aiming to address the path explosion problem in dynamic symbolic execution and improve code reachability analysis, i.e., proving whether a target program point is reachable or not.

This paper proposes a method that leverages LLMs during the (DSE) process to extract program path information, understand control structures, and generate constraints for execution paths. In addition, LLMs can use the path information to generate the corresponding code to solve path constraints. This approach effectively improves the accuracy of symbolic execution.

2.3 Direct Memory Access of Embedded Firmware

DMA (Direct Memory Access) is a commonly used asynchronous communication technique in embedded devices [20]. Through DMA, peripherals can directly transfer data to memory or read data from memory without the involvement of the CPU (Central Processing Unit), thereby improving the system's efficiency and performance. In embedded devices, the DMA controller is responsible for managing the data transfer process. The DMA controller establishes channels between various peripherals and memory, enabling direct data transfer between them. By configuring the DMA controller's registers, parameters such as the source address, destination address, and transfer length of the data transfer can be specified. The DMA transfer process is initiated when a peripheral enabled for DMA makes a request to use DMA. The CPU then configures the DMA stream and delegates the memory copy function to the DMA controller. Subsequently, the DMA controller transfers data to the RAM (Random Access Memory) area or to a DMA-capable peripheral without further CPU involvement. Finally, when the DMA operation is complete, an interrupt is generated to notify the CPU of the completion.

Mera et al. [9] proposed DICE, a tool for automatic simulation of DMA input channels in dynamic firmware analysis. DICE identifies the source and destination pointers written by the firmware during DMA transfer configuration and uses this information to emulate DMA input channels, enabling firmware analyzers to recognize and manipulate DMA inputs. DICE achieved high accuracy on sample firmware (89% true positive rate, 0% false positive rate) and improved analysis coverage on real-world firmware. However, DICE adopts a fully randomized input generation strategy, which inevitably reduces the accuracy of DMA input channel simulation and DMA interaction. The same research group also proposed D-Box [21], a method to address the lack of DMA support in memory protection unit (MPU)-based resource isolation schemes in embedded applications. Experimental results show that compared to standard FreeRTOS-MPU, D-Box significantly reduces the attack surface and power consumption while maintaining low overhead. Gross et al. [22] manually modified XMPU and XPPU register configurations and combined them with cache maintenance operations to resolve memory isolation issues caused by the accelerator coherency Port interface in FPGA-SoC. Experimental results demonstrated that this method effectively prevents DMA-based attacks.

Current DMA-based modeling approaches for embedded firmware analysis still suffer from several issues, such as reliance on randomized input generation, manual hardware configuration, and a lack of semantic understanding of firmware behavior. These limitations reduce the accuracy, scalability, and depth of the analysis. To address these challenges, we propose ADFEmu, an LLM-assisted dynamic emulation approach. This approach leverages large language models to intelligently infer DMA input structures,guides symbolic execution with path-sensitive constraints,and automatically configures DMA behavior. As a result, ADFEmu significantly improves emulation accuracy and code coverage.

3 Design and Implementation

The design of ADFEmu is centered around using dynamic symbolic execution technology to support DMA input during the firmware rehosting process. Its emulation scope applies to the aforementioned monolithic firmware, with the goal of expanding the applicability of firmware that can be re-hosted.

Fig. 1 presents the overall architecture of ADFEmu framework. ADFEmu targets firmware binaries in ELF format for emulation. Initially, through automated static analysis, information such as the required RAM size, memory address mapping range, and starting address of the target firmware is determined. ADFEmu then loads and begins executing the firmware using QEMU.

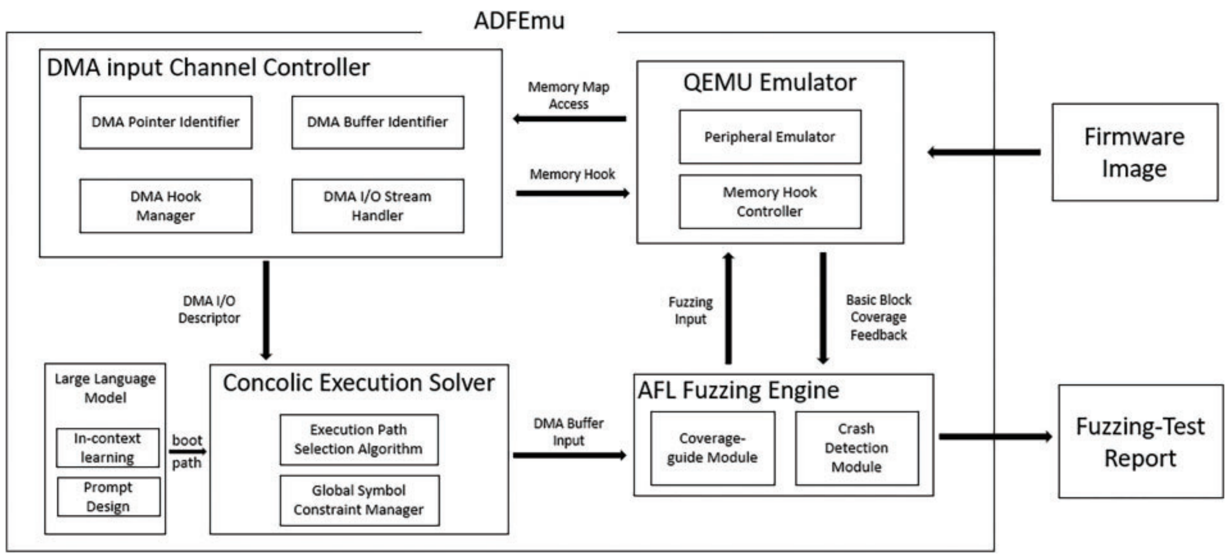

Figure 1: Architecture of ADFEmu

However, since there is no actual hardware peripheral present, theoretically, the firmware would encounter an error and exit when attempting to execute code blocks that access peripherals. To handle this, ADFEmu is designed to place hooks in the MMIO-mapped memory regions. When it detects that the firmware is attempting to read from or write to these MMIO-mapped memory regions, i.e., when it tries to invoke peripheral logic, ADFEmu intercepts this process via the hook, generates and returns an input using the fuzzing tool to ensure the firmware continues to run smoothly.

Concurrently, ADFEmu continuously monitors the state of the DMA controller. If it detects an attempt by the firmware to activate the DMA controller, ADFEmu again uses hooks to manage the configuration of the DMA transfer descriptors. It dynamically creates DMA input channels by reading parameters such as the source address, destination address, and transfer length during the firmware's DMA operations. In addition, the core innovation of ADFEmu lies in the fact that the input provided by the emulated peripherals can influence the subsequent state of the program. Our framework uses DSE to deduce and solve for the expected input values the program anticipates. By following the paths selected through DSE, ADFEmu can access as much DMA-related logic in the firmware as possible, uncovering potential vulnerabilities in the firmware while ensuring the emulation process does not crash.

During this process, we integrate LLM technology, where the model assists in path selection for dynamic symbolic execution by designing context information and relevant prompts based on program path data and control structures. Finally, after exploring as many potential program paths as possible, the fuzzing engine identifies firmware vulnerabilities based on crash outcomes.

3.1 Fuzzing-Based Peripheral Emulator

ADFEmu uses QEMU [23] to emulate the ARM Cortex-M architecture instruction set of embedded firmware. Before the emulation process begins, ADFEmu utilizes a static analysis tool to obtain the address range of the target firmware's MMIO regions and configures these in QEMU. For example, in the commonly used STM32F103 Microcontroller Unit(MCU), the memory range mapped to hardware peripherals spans from 0x40000000 to 0x5fffffff. During the entire emulation process, ADFEmu hooks all MMIO accesses within this specified memory range. When the firmware accesses the MMIO memory region, the program's inputs are the inputs required by the peripheral hardware. ADFEmu then invokes a fuzzer to provide specific inputs, ensuring the continuous operation of the emulation process. When the firmware accesses logic related to DMA inputs, ADFEmu transfers control of the emulation process to the DMA input channel controller for handling.

In our implementation, we primarily modified the memory.c file in the QEMU source code, adding hook-related logic to the memory access functions. During the QEMU emulation process, each memory access triggers calls to the unassigned_mem_write or unassigned_mem_read functions. We added specific conditional checks within these functions for accesses to DMA pointers and MMIO memory regions. When these conditions are met, control is taken over by the hook and handed off to other modules for processing.

3.2 DMA Input Channel Controller

To simulate DMA inputs, it is necessary to monitor and process read or write commands to the DMA memory buffer and confirm whether the entire data transfer process is completed. ADFEmu draws on concepts from DICE related to the definition of a virtual DMA controller, including DMA pointer, DMA buffer, and DMA descriptor. In real hardware, a DMA transfer requires writing parameters such as the source address, destination address, and transfer length into the DMA controller. Similarly, ADFEmu simulates DMA transfers in three distinct steps: (1) identification and activation of DMA pointers, (2) data transfer simulation, and (3) transfer completion detection.

According to the definition, ADFEmu identifies pointers that reference the DMA memory area and classifies them as DMA pointers. Once identified, the virtual DMA controller in ADFEmu combines a source address pointer and a destination address pointer into a DMA buffer structure. It writes the DMA buffer configuration information into the DMA descriptor and then creates a new DMA input channel to simulate the execution of the DMA transfer. DMA transfer is essentially a continuous memory read/write operation. However, due to the absence of physical hardware support, ADFEmu cannot directly obtain all required configuration parameters. Specifically, the starting address of the DMA buffer can be recognized through symbolic memory access, but the buffer length and ending address are generally unknown. To address this issue, we designed a heuristic DMA input channel algorithm that operates in a dynamic environment.

This algorithm dynamically creates or terminates DMA input channels based on the detection of DMA pointers. At the same time, it proactively provides corresponding data input to the destination address in memory to emulate hardware DMA behavior. The detailed process is shown in Algorithm 1.

Algorithm 1: DMA input channel algorithm

```
Require: DMA Pointers
Ensure: Dynamic DMA Input Channel
1: NewChannel ← CreateChannel(DMA Pointers)
2: CurrentPtr ← MemorySeek(NewChannel)
3: while TransferNotEnd(NewChannel) do
4:     temp ← GetDataValue(NewChannel)
5:     Transfer(CurrentPtr, temp)
6:     CurrentPtr ← CurrentPtr + 4                    ▷ Move to next memory address
7: end while
8: CloseChannel(NewChannel)
9: return NewChannel
```

The DMA input channel algorithm takes the identified DMA pointers as input. It then establishes a DMA input channel based on the addresses of these DMA pointers. During the dynamic memory read/write process, the status of the data transfer is continuously monitored. When the DMA stream transfer loop in the firmware is detected to have ended, the DMA input channel is also closed accordingly.

In obtaining the specific values for the DMA input provided to the firmware, a process similar to the one described earlier is used, where the fuzzing engine generates values for the firmware. However, because the data content in the input stream often relates to a series of firmware program codes with specific semantic information, relying solely on the fuzzing engine to provide all hardware-generated values for DMA requests is not feasible. Therefore, ADFEmu enhances the existing method by incorporating a Concolic Execution Solver. By using a path selection algorithm, the solver identifies the most promising program paths and adds constraint solving to obtain valuable concrete inputs, which are then provided to the DMA input channel for read and write operations.

After the data transfer is complete, the DMA controller in a real device issues an interrupt to the CPU to notify the user. The controller then closes the previously configured DMA stream and waits for subsequent transfer operations. In ADFEmu, the virtual DMA input channel controller, lacking hardware support, cannot rely on interrupts to determine when the data transfer has ended. We have adopted two methods to determine the appropriate timing to close the DMA input channel:

(1) When a new DMA stream is detected and a new input channel is created, the previous one is terminated, and
(2) If the firmware starts writing to the memory buffer associated with the source address, this is treated as the end of the current DMA transfer.

3.3 Concolic Execution Solver of DMA Input

Ideally, if relying on a fuzzing engine to randomly provide DMA inputs for the firmware code and having unlimited time would ensure the success of the emulation. However, in actual testing, it has been observed that the random inputs provided by the fuzzing engine often cause the firmware program to enter error-handling logic or crash outright. Therefore, our method innovatively incorporates a Concolic Execution Solver to explore more meaningful code segments.

The resulting higher code coverage increases the likelihood of traversing vulnerable code within the firmware, thereby aiding in the identification of potential vulnerabilities.

Each time the firmware program invokes a DMA input, the DMA input channel controller sets the target address accessed by the program as a symbolic variable to be solved by the concolic execution engine. Subsequently, after the symbolic variable is resolved by the solver, the result is returned for further processing.

Fig. 2 illustrates the objective of our path selection algorithm, which is to solve for inputs that guide the program's execution flow toward a selected path. This path must satisfy the branching conditions of the program and avoid triggering the firmware's error-handling logic.

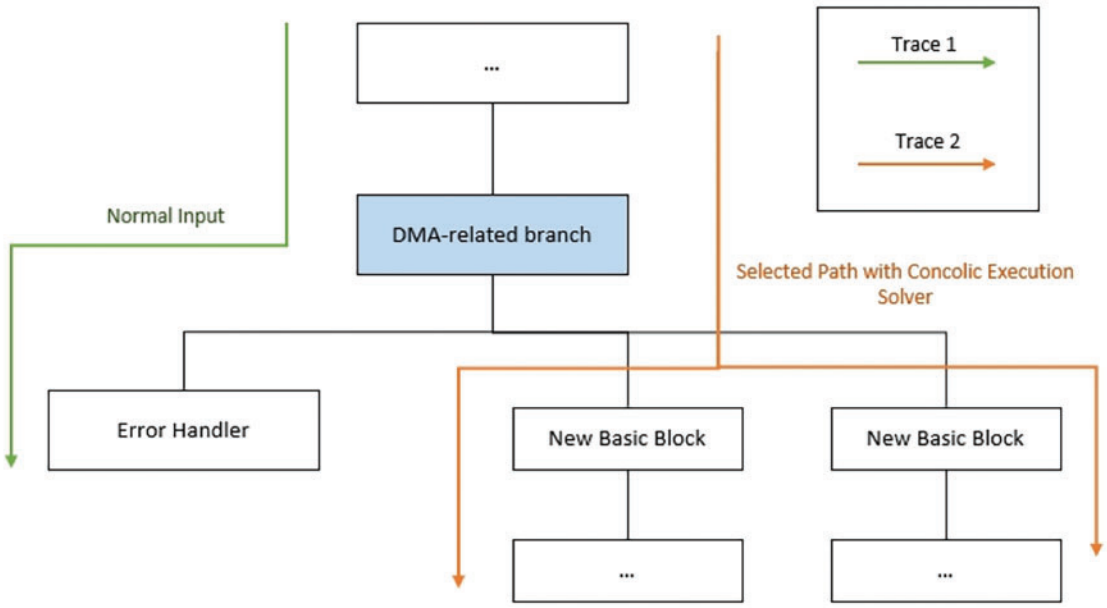

Figure 2: The illustration of path selection algorithm

The path selection algorithm uses the dynamic symbolic execution engine to explore a certain range of basic blocks ahead from the current code block. When it encounters branch conditions, all active states are recorded and saved. To prevent potential state explosion during symbolic execution, the exploration range is strictly limited to within five branch jumps. Upon reaching this range limit or encountering an abnormal return, the algorithm selects a branch from all possible paths based on a certain priority and constrains the associated symbolic variables for solving. For multiple possible path branches, the algorithm hands the path selection to LLMs, which chooses the optimal path. It should be noted that our input data here is in assembly language. The process of optimal path selection involves first providing all paths (basic code blocks) to the LLM and asking it to analyze each basic block. When multiple path candidates exist, the algorithm consults a Large Language Model (LLM) to assist in prioritization. The large language model used in this study is GPT-4 [24]. The prompt words we designed are as follows:

LLM Prompt

Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

Instruction:
Please analyze the content of each program basic block below, generate different program paths by combining each basic block without repetition, and then rank these paths according to how much program information they can capture in the dynamic symbolic execution process.

Input:
A list of basic blocks extracted from the current firmware execution trace.

Response:
A ranked list of synthesized program paths, ordered by their potential to reveal semantic information during dynamic symbolic execution.

After the LLM selects the most promising paths, the algorithm further filters and selects a branch from the filtered paths according to a priority, and the symbolic variables are constrained for solving again. This aligns with our goal of maximizing code coverage. The algorithm excludes paths that would lead to a dead-end state. Additionally, by leveraging a pre-trained large model, the algorithm evaluates the potential value of the paths, predicting which paths are more likely to uncover previously uncovered logic or potential vulnerabilities. The algorithm selects the path with the highest address among the candidates, as a higher address indicates deeper progression within the program. The full process is described in Algorithm 2.

Algorithm 2: Path selection algorithm

```
Require: Current Basic Block
Ensure: Selected Path
 1: States ← CURRENTSTATE(Current Basic Block)
 2: StepDepth ← 0
 3: while StepDepth < 5 do
 4:     tempStates ← STEPONCE(States)
 5:     if ISACTIVE(tempStates) then
 6:         UPDATESTATES(States, tempStates)
 7:     else if ISDEADEND(tempStates) then
 8:         DROPSTATE(tempStates)
 9:     end if
10:     StepDepth ← StepDepth + 1
11: end while
12: if HAVENEWPATH(States) then
13:     SelectedPath ← GETNEWPATHSTATE(States)
14: else
15:     SelectedPath ← GETHIGHESTPATH(States)
16: end if
17: return SelectedPath
```

In our implementation, we utilized Angr [25] as the concolic execution engine. During runtime, ADFEmu maintains a shared memory region between Angr and QEMU. When QEMU requires Angr to perform path selection and input solving, it synchronizes the current code block information and context with Angr. Angr then creates symbolic variables and executes the path selection algorithm, solving for the expected state. Once the results are obtained, they are returned to QEMU, allowing QEMU to continue the DMA transfer process.

4 Evaluation

We evaluated ADFEmu from three perspectives: (1) whether it can maintain stable operation without crashing for an extended period while running various firmware on different MCUs; (2) whether it can cover more execution paths during firmware re-hosting; and (3) whether it can discover more unknown bugs in the firmware.

To evaluate the first aspect, we collected ten open source real-world firmware samples from GitHub, each running on four different types of MCUs. We used ADFEmu to run these firmware samples for an extended period to assess the stability of re-hosting.

For the second and third aspects, we conducted a comparative evaluation with DICE. To ensure consistency in the environment, we strictly followed the environment setup instructions provided in DICE's GitHub repository and performed the comparison on the same server.

All our experiments were conducted on an Intel(R) Xeon(R) Silver 4210R CPU @ 2.40 GHz with 8 GB of RAM, running on a 64-bit Ubuntu 18.04 LTS system.

4.1 Re-Hosting Stability of ADFEmu

Table 2 lists the information on the open source firmware used in our experiments. To ensure successful execution of these firmware samples, we integrated the AFL Fork Server-related code logic into the firmware source code. We conducted experiments on these open source firmware samples by running each experiment continuously for 24 h, repeating the experiment three times for each firmware.

Table 2: Firmware information used for evaluation

Firmware	Year	MCU	OS
Modbus [26]	2019	STM32F303	FreeRTOS
Guitar Pedal [27]	2015	STM32F303	Mbed OS
Soldering Station [28]	2020	STM32F103	Baremetel
Stepper Motor [29]	2016	STM32F466	Baremetel
GPS Receiver [30]	2024	STM32F103	Baremetel
MIDI Synthesizer [31]	2014	STM32F429	Baremetel
Oscilloscope [32]	2018	STM32F103	Arduino
DDS-WaveGen [33]	2022	STM32F103	Baremetel
GPS-Logger [34]	2020	STM32F429	Baremetel
PatternDriver [35]	2021	STM32F103	Baremetel

If the target firmware can maintain normal operation without crashing during the experiment and produces correct output at the end of the experiment, we consider ADFEmu to be capable of successfully emulating the target firmware.

Our experimental results show that DICE successfully emulated nine out of the ten open source firmware samples.

The only exception was DDS-WaveGen, which encountered a timeout issue with all input cases during the experiment and thus failed to emulate successfully. The overall emulation success rate for DICE was 90%. In contrast, ADFEmu achieved a 100% success rate in emulating all ten open source firmware samples without requiring any manual adaptation.

4.2 Code Coverage and Crash Detection of ADFEmu

In our experiment, the coverage data and crash trigger counts for the firmware were obtained from the result reports generated by AFL [36] at the end of the fuzzing test. To minimize random factors, each experiment was run continuously for 24 h and repeated three times for each firmware sample. To ensure consistency in the initial conditions of the fuzz testing, we provided the same initial seed, with a length of 1024 bytes, for all firmware samples.

Table 3 presents our experimental results. The results indicate that compared to DICE, ADFEmu achieves a higher basic block coverage in most firmware. For example, in the case of Modbus, ADFEmu has a coverage of 59.1%, whereas DICE only achieves 56.8%, representing an improvement of 4.05%. Similar improvements are evident in the Soldering Station (+14.24%) and GPS Receiver (+13.47%). DDS-WaveGen could not run successfully because all input use cases timed out, so the results could not be compared. This demonstrates that ADFEmu is more effective than DICE in enhancing firmware code coverage.

Table 3: Fuzzing test evaluation results

Firmware	**BBL Cov. [%]**			**Total paths**			**Max depth**		**Crash counts**	
	ADFEmu	**DICE**	**Improve**	**ADFEmu**	**DICE**	**Improve**	**ADFEmu**	**DICE**	**ADFEmu**	**DICE**
Modbus	59.1	56.8	+4.05%	745	756	-1.45%	11	8	33	30
Guitar pedal	17.5	16.0	+9.37%	2803	2773	+1.08%	3	5	1	0
Soldering Station	33.7	29.5	+14.24%	181	166	+9.03%	9	3	1	2
Stepper motor	21.6	22.4	-3.57%	4070	3844	+5.88%	5	3	0	0
GPS receiver	16.0	14.1	+13.47%	1918	1788	+7.27%	6	6	1	1
MIDI Synthesizer	43.0	40.7	+5.65%	866	836	+3.59%	9	7	1	1
Oscilloscope	28.6	27.3	+4.76%	256	249	+2.81%	4	4	0	0
DDS-WaveGen	21.3	–	–	248	–	–	7	–	1	–
GPS-Logger	11.2	11.3	-0.88%	915	894	+2.34%	8	7	0	0
PatternDriver	5.9	5.8	+1.72%	736	790	-6.83%	5	7	1	1
Total	–	–	–	12738	12096	+5.31%	67	50	39	35

ADFEmu shows an increase in the total number of paths in most firmware, such as Soldering Station and Stepper Motor, with improvements of 9.03% and 5.88%, respectively. This indicates that ADFEmu can explore more execution paths, which aligns with the design expectations of our method.

Based on the path exploration count vs. time graph in Fig. 3, it can be observed that in the early stages of the fuzzing experiment, ADFEmu, with its path selection algorithm, prioritizes exploring new paths that have not been explored before, and high-weight paths. This results in a significantly faster path exploration rate in the early phase of the experiment compared to DICE.

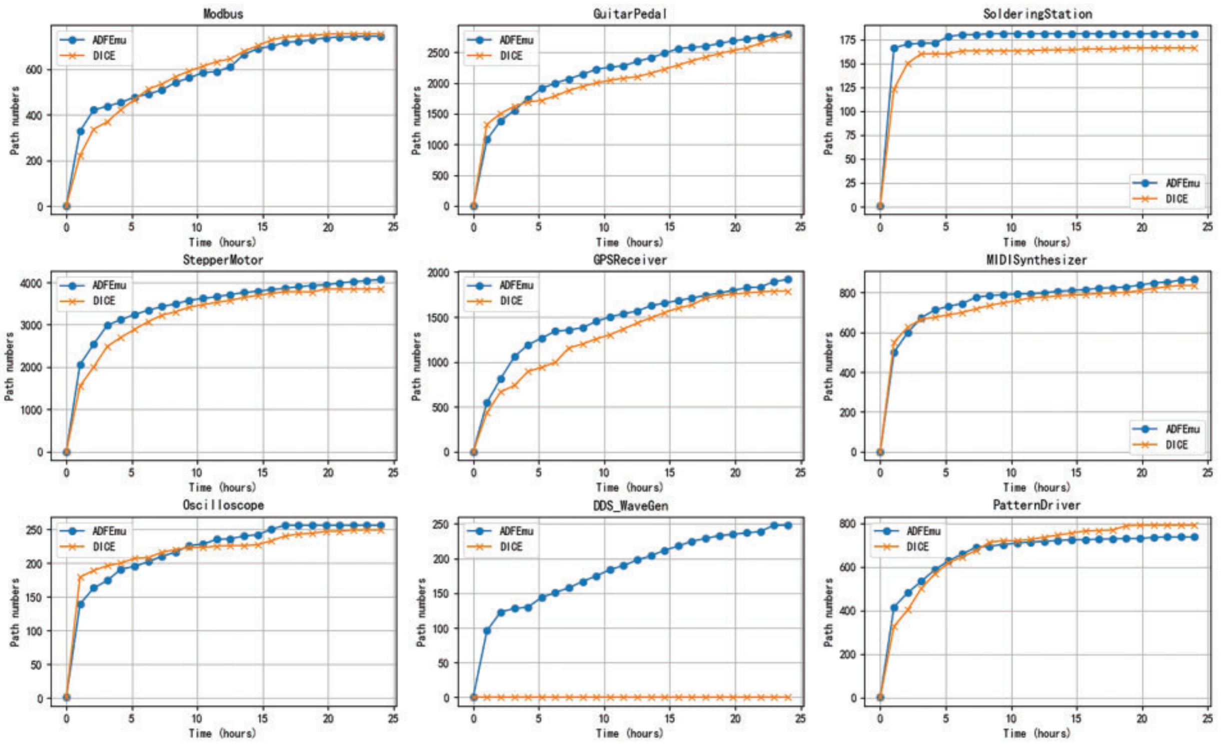

Figure 3: Time dependent graph of the number of exploration paths for different firmware

After a period of exploration, the DMA inputs provided by DICE can no longer explore more potential paths, and in the latter half of the fuzzing experiment, the number of newly explored program paths by DICE approaches zero. This is also correlated with the basic block coverage metric. In contrast, ADFEmu continues to discover new program paths even in the later stages of the experiment, approaching its conclusion. This demonstrates that the path selection algorithm designed in this paper effectively identifies and chooses to explore new potential program paths, and further shows that ADFEmu is more effective than DICE in improving firmware code coverage.

However, in the case of Pattern Driver, the number of paths decreased by 6.83%. This may be due to the prioritization of ADFEmu's path selection algorithm, which chose different paths than DICE on this firmware, ultimately leading to a focus on depth exploration at the expense of breadth exploration.

ADFEmu achieves greater max depth on most firmware, with particularly significant improvements on the Modbus and Soldering Station firmware. The metric of max depth often reflects the effectiveness of the input we provide to the firmware. This is because the responsible logic often requires satisfying a series of branching conditions to be reached. If the testing remains at the surface logic and cannot pass the relevant condition checks, it will not be able to reach the deeper code logic hidden within, thereby missing the potential bugs.

Overall, ADFEmu achieves higher code coverage on most firmware, demonstrating stronger code exploration capabilities. ADFEmu generally explores more paths and reaches greater depths, surpassing DICE in both the total number of paths explored and the total depth reached across all firmware. Specifically, the total number of paths increased by 5.31%. On certain firmware, ADFEmu also discovers more crashes, with 4 more crashes found compared to DICE, indicating that ADFEmu is more effective in identifying potential vulnerabilities.

5 Discussion

The current evaluation of ADFEmu focuses on monolithic firmware running on bare-metal or lightweight embedded systems such as FreeRTOS. While ADFEmu demonstrates strong performance in these settings, extending its capabilities to support Linux-based firmware introduces new challenges. Linux-based firmware features more complex interactions among the kernel, user-space applications, and device drivers. These interactions introduce greater structural complexity and execution dynamics, making accurate emulation more difficult. Furthermore, Linux systems employ advanced memory management schemes–such as virtual memory and protection mechanisms–which ADFEmu must support to maintain fidelity. Emulating specific kernel components and device drivers also requires modeling diverse subsystem behaviors, increasing development complexity.

To address these challenges, we propose an incremental extension strategy: starting with lightweight Linux-based firmware and progressively enhancing ADFEmu's capabilities to support more complex systems. This approach allows gradual adaptation while maintaining system robustness. Additionally, we plan to integrate existing kernel and driver emulation tools to accelerate support for Linux features. Such collaboration will enable ADFEmu to benefit from mature implementations and broaden its applicability.

Large Language Models (LLMs) play a crucial role in ADFEmu, particularly in guiding path selection and generating DMA inputs. As LLM technology continues to evolve, the performance and efficiency of ADFEmu are expected to improve. For instance, more advanced LLMs may offer a deeper understanding of program semantics, enabling more accurate path selection and input generation. Additionally, leveraging domain-specific LLMs may further enhance ADFEmu's effectiveness in specialized application scenarios. It is worth noting that the LLM is used as a heuristic guide (one part of a scoring strategy). Even if LLM ranks a path poorly, bad path suggestions may slow down progress slightly but do not block exploration. The symbolic executor can eventually cover all paths due to fallback mechanisms.

In the future, we will continue our research along the following three directions:

(1) Supporting virtual memory and protection features for Linux-based memory management;
(2) Simulation of core kernel functionalities, such as process scheduling and I/O subsystems;
(3) Exploring more powerful Large Language Models (LLMs) to further enhance ADFEmu performance in path selection and input generation.

By extending ADFEmu along these directions, we aim to build a more general-purpose framework capable of supporting both lightweight and full-featured embedded firmware, including Linux-based systems.

6 Conclusion

We introduce ADFEmu, a firmware re-hosting framework that integrates dynamic symbolic execution, LLM-assisted path selection, and DMA input emulation for enhanced firmware fuzzing. Our experimental results demonstrate superior code coverage, execution path exploration, and vulnerability detection compared to existing methods. Future work includes extending ADFEmu to handle complex embedded peripherals and enhancing LLM path selection strategies for broader firmware architectures.

Acknowledgement: Not applicable.

Funding Statement: This research was funded by the Science and Technology Project of State Grid Jiangsu Electric Power Company Ltd., grant number J2024169.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Yixin Ding and Xinjian Zhao; methodology, Yixin Ding; software, Zicheng Wu; validation, Yixin Ding, Xinjian Zhao and Hao Han;

formal analysis, Yixin Ding; investigation, Xinjian Zhao; resources, Yixin Ding; data curation, Yichen Zhu; writing—original draft preparation, Yichen Zhu; writing—review and editing, Longkun Bai; visualization, Hao Han; supervision, Yixin Ding; project administration, Yixin Ding; funding acquisition, Yixin Ding. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Global Embedded Security Market Statistical Analysis and Forecast. 2024. [Internet]. [cited 2024 Jul 5]. Available from: https://www.reportsinsights.com/industry-forecast/global-embedded-security-marketstatistical-analysis-673783.
2. Andrew F, Ballo T, Muench M, Leek T, Bulekov A, Gavitt BD, et al. SoK: Enabling security analyses of embedded systems via rehosting. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (AsiaCCS '21); 2021 Nov 15–19; Hong Kong, China. p. 687–701. doi:10.1145/3433210.3453093.
3. Zhou W, Guan L, Zhang Y. Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th USENIX Security Symposium (USENIX Security '21); 2021 Aug 11–13; Virtual Event (originally planned for Vancouver, BC, Canada). p. 2007–24. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou.
4. Johnson E, Bland M, Zhu Y, Mason J, Savage S, Levchenko K. Jetset: targeted firmware rehosting for embedded systems. In: 30th USENIX Security Symposium; 2021 Aug 11–13; Online. p. 321–38. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson.
5. Clements AA, Gustafson E, Scharnowski T, Grosen P, Fritz D, Kruegel C, et al. HALucinator: firmware re-hosting through abstraction layer emulation. In: 29th USENIX Security Symposium (USENIX Security 20); 2020 Aug 12–14; Boston, MA, USA. p. 1201–18. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.
6. Wei Y, Wang Y, Zhou L, Zhou X, Jiang Z. IEmu: interrupt modeling from the logic hidden in the firmware. J Syst Archit. 2024;154:103237. doi:10.1016/j.sysarc.2024.103237.
7. Scharnowski T, Bars N, Schloegel M, Gustafson E, Muench M, Vigna G, et al. Fuzzware: using precise MMIO modeling for effective firmware fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22); 2022 Aug 10–12; Boston, MA, USA. p. 1239–56. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.
8. Feng B, Mera A, Lu L. P2IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th USENIX Conference on Security Symposium; 2020 Aug 12–14; Boston, MA, USA. p. 1237–54. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.
9. Mera A, Feng B, Lu L, Kirda E. DICE: automatic emulation of DMA input channels for dynamic firmware analysis. In: Proceedings of the IEEE Symposium on Security and Privacy (SP); 2021 May 24–27; San Francisco, CA, USA. p. 1938–54. doi:10.1109/SP40001.2021.00018.
10. Muench M, Stijohann J, Kargl F, Balzarotti D. What you corrupt is not what you crash: challenges in fuzzing embedded devices. In: Proceedings of the Network and Distributed Systems Security (NDSS); 2018 Feb 18–21; San Diego, CA, USA. p. 1–15. doi:10.14722/ndss.2018.23166.
11. Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: 28th USENIX Security Symposium; 2019 Aug 14–16; Santa Clara, CA, USA. p. 1099–114. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

12. Kim M, Kim D, Kim E, Kim Y. FirmAE: towards large-scale emulation of IoT firmware for dynamic analysis. In: Proceedings of the 36th Annual Computer Security Applications Conference; 2020 Dec 7–11; Austin, TX, USA. p. 733–45. doi:10.1145/3427228.3427294.
13. Chen DD, Egele M, Brumley D. Towards automated dynamic analysis for Linux-based embedded firmware. In: Proceedings of the Network and Distributed System Security Symposium; 2016 Feb 21–24; San Diego, CA, USA. doi:10.14722/ndss.2016.23415.
14. Chen J, Deng L, Qiu Y, Zhao P, Lei H, Song J, et al. LLM-based automated modeling in symbolic execution for securing medical software; 2024. [cited 2024 Jun 25]. Available from: https://ssrn.com/abstract=4938953.
15. Wang W, Liu K, Chen A, Li G, Jin Z, Huang G, et al. Python symbolic execution with LLM-powered code generation. arXiv:2409.09271. 2024. doi:10.48550/arXiv.2409.09271.
16. Jiang Y, Liang J, Ma F, Chen Y, Zhou C, Shen Y, et al. When fuzzing meets LLMs: challenges and opportunities. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering; 2024 Jul 15–19; Porto de Galinhas, Brazil. p. 492–6. doi:10.1145/3663529.3663784.
17. Xia C, Paltenghi M, Le Tian J, Pradel M, Zhang L. Fuzz4all: universal fuzzing with large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering; 2024 Apr 14–20; Lisbon, Portugal. p. 126. doi:10.1145/3597503.3639121.
18. Cha S, Hong S, Bak J, Kim J, Lee J, Oh H. Enhancing dynamic symbolic execution by automatically learning search heuristics. IEEE Trans Softw Eng. 2021;48(9):3640–63. doi:10.1109/TSE.2021.3101870.
19. Jaffar J, Maghareh R, Godboley S, Ha X. TracerX: dynamic symbolic execution with interpolation (Competition Contribution). In: Fundamental Approaches to Software Engineering (FASE 2020). Cham, Switzerland: Springer; 2020. p. 530–4. doi:10.48550/arXiv.2012.00556.
20. Saravanan R, Pudukotai Dinakarrao SM. The fuzz odyssey: a survey on hardware fuzzing frameworks for hardware design verification. In: Proceedings of the Great Lakes Symposium on VLSI 2024; 2024 Jun 12–14; Clearwater, FL, USA. p. 192–7. doi:10.1145/3649476.3658697.
21. Mera A, Chen YH, Sun R, Kirda E, Lu L. D-box: DMA-enabled cocmpartmentalization for embedded applications; 2022. arXiv:2201.05199. doi:10.14722/ndss.2022.24053.
22. Gross M, Jacob N, Zankl A, Sigl G. Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC. J Cryptographic Eng. 2022;12(2):181–96. doi:10.1007/s13389-021-00273-8.
23. Bellard F. QEMU, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference; 2005 Apr 13; Anaheim, CA, USA. p. 41–6. [cited 2025 Jun 25]. Available from: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.
24. Gpt-4 documentation. [Internet]. [cited 2025 Jun 25]. Available from: https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.
25. Wang F, Shoshitaishvili Y. Angr-the next generation of binary analysis. In: 2017 IEEE Cybersecurity Development (SecDev); 2017 Sep 24–26; Cambridge, MA, USA. p. 8–9. doi:10.1109/SecDev.2017.14.
26. Modbus firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/DoHelloWorld/stm32f3_Modbus_Slave_UART-DMA-FreeRTOS.
27. Guitar pedal firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/Guitarman9119/Nucleo_Guitar_Effects_Pedal.
28. Soldering station firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/PTDreamer/stm32_soldering_iron_controller.
29. Stepper motor firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/omuzychko/StepperHub.
30. GPS receiver firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/MaJerle/stm32-usart-uart-dma-rx-tx.
31. MIDI synthesizer firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/mondaugen/stm32-codec-midi-mmdsp-test.
32. Oscilloscope firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/pingumacpenguin/STM32-O-Scope.

33. DDS-WaveGen firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/BojanSof/BluePillWaveGen.git.
34. GPS-Logger firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/MaJerle/stm32f429/blob/main/PROJECT-04-GPS_LOGGER/User/main.c.
35. PatternDriver firmware image. [Internet]. [cited 2025 Jun 25]. Available from: https://github.com/mnemocron/STM32_PatternDriver.
36. Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: combining incremental steps of fuzzing research. In: Proceedings 14th USENIX Workshop Offensive Technology; 2020 Aug 11; Boston, MA, USA. p. 1–12. [cited 2025 Jun 25]. Available from: https://www.usenix.org/conference/woot20/presentation/fioraldi.