Trapped by Your WORDs: (Ab)using Processor Exceptions for Binary Instrumentation on Bare-metal Embedded Firmware

Shipei Qu, Xiaolin Zhang, Chi Zhang, Dawu Gu
{shipeiqu,xiaolinzhang,zcsjtu,dwgu}@sjtu.edu.cn
School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai
State Key Laboratory of Cryptology, Beijing

ABSTRACT

Analyzing the security of closed-source drivers and libraries in embedded systems holds significant importance, given their fundamental role in the supply chain. Unlike x86, embedded platforms lack comprehensive binary manipulating tools, making it difficult for researchers and developers to effectively detect and patch security issues in such closed-source components. Existing works either depend on full-fledged operating system features or suffer from tedious corner cases, restricting their application to bare-metal firmware prevalent in embedded environments.

In this paper, we present PIFER (Practical Instrumenting Framework for Embedded fiRmware) that enables general and fine-grained static binary instrumentation for embedded bare-metal firmware. By abusing the built-in hardware exception-handling mechanism of the embedded processors, PIFER can perform instrumentation on arbitrary target addresses. Additionally, We propose an instruction translation-based scheme to guarantee the correct execution of the original firmware after patching. We evaluate PIFER against real-world, complex firmware, including Zephyr RTOS, CoreMark benchmark, and a close-sourced commercial product. The results indicate that PIFER correctly instrumented 98.9% of the instructions. Further, a comprehensive performance evaluation was conducted, demonstrating the practicality and efficiency of our work.

KEYWORDS

Embedded System Security, Binary Rewriting,

1 INTRODUCTION

In recent years, the security of embedded systems has become increasingly important with their broad applications. Designed for low-cost environments, the majority of embedded devices/chips are *bare-metal*, i.e., running only a single statically linked firmware. However, the lack of advanced operating systems and architectural security features, such as a Memory Mapping Unit (MMU), render bare-metal embedded devices particularly vulnerable to programming errors and malicious attacks [1]. Moreover, bare-metal chips/devices are being actively exploited in sophisticated attacks against high-value targets, such as the compromising of AMD-SP [3], Google Pixel's security co-processor [20], and Tesla's key-fob [26]. Therefore, it is crucial to efficiently detect and defend against vulnerabilities of such devices.

As many widely used libraries or drivers are still closed-source in the IoT-embedded industry, binary instrumentation is one of the fundamental techniques in addressing the above tasks. It involves injecting additional code into the binary program, allowing developers and security researchers to observe or modify its runtime behavior. From a defense perspective, binary instrumentation enables vendors to perform feedback-based fuzzing and custom security patches for low-level closed-source components that are frequently targeted in supply chain attacks. Take Ripple20 [15] for an example, a vulnerable proprietary TCP/IP stack risks hundreds of millions of devices, and 120 days is insufficient for its developers to deliver the correct updates to all affected vendors. For offensive security research, the binary-only firmware also poses a hindrance to reverse engineering and exploitation. For example, hardware-based attacks, which have gained popularity in recent years, typically enable the bypass of secure boot or debugging protections [22]. However, researchers find it tedious to manipulate and analyze the raw binary firmware after a successful attack. In general, efficient modification at the binary level is crucial for both offensive and defensive security purposes.

Related work While many static binary rewriting techniques work well on x86, limited studies has been conducted for embedded architectures[17][23][5]. Existing binary instrumentation techniques can be classified into *static* and *dynamic*, depending on whether the modifications to the program are made before or during execution. For static approaches, the most common trampoline way replaces an original instruction with a *jump* targeting the newly appended code. However, in embedded architectures, the jump or addressing range of a single instruction is quite limited due to the fixed length (i.e., 2 bytes for ARM-Thumb/RISCV-Compressed), failing many rewriters commonly worked on x86 [13][4]. Another approach for static rewriting is to lift the entire binary to mutable Intermediate Representation (IR) or even re-assemble assembly [6], but a recent study reveals that related works are error-prone in real-world applications, as generically distinguishing between pointers and scalars is still an undecidable problem in binary analysis [16].

For dynamic approaches, using hardware debugging protocols (JTAG or SWD [7]) to modify firmware logic will introduce significant communication overhead and rarely be deployed in released production. Rehosting under emulated environments is also a potential dynamic approach [10], but limited scalability and poor performance may arise. Another dynamic solution is the trap-based approach, which works by modifying the target instructions to trap raisers (e.g., the **`INT3`** in x86) and performs the instrumentation

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3655687

with pre-registered handlers. Traditional trap-based approaches like GDB require the restoration of modified instructions to ensure correctness. However, the ROM/FLASH memory where firmware typically resides does not allow efficient on-the-fly modifications at the byte level. For specific tasks like control flow integrity [21], a workaround is to manually emulate a small set of instructions (call, return) in other locations. However, for general instrumentation, it is challenging to correctly handle all possible 16/32 bits instructions. Furthermore, bare-metal firmware lacks a comprehensive operating system like Linux that enables the registration of custom handlers, and integrating an efficient self-contained hook system into the compact binary firmware is also a challenging task.

Contributions In this paper, we propose PIFER, a fine-grained and practical binary instrumenting framework designed for bare-metal firmware. In general, PIFER combines the advantages of the dynamic trap-based rewriting and the static patching. Utilizing the exception-handling mechanism of embedded processors, PIFER achieves execution flow hijacking within a compact 2-byte (1-*WORD*) sequence, which is fitted into fixed-length instructions and thus applicable at arbitrary addresses. To ensure correct re-execution of the modified original instructions, we design a novel single instruction translator capable of converting arbitrary regular instructions into relocated equivalents. Overcoming the limitations of traditional binary instrumenting within embedded firmware, PIFER offers a concrete solution to facilitate late-stage code modifications on bare-metal firmware for both security and developmental purposes. Furthermore, a comprehensive evaluation demonstrates that PIFER is able to handle real-world complex firmware with acceptable runtime and memory overhead.

To summarize, our core contributions are:

- We present the first general and fine-grained static binary instrumenting framework for bare-metal embedded firmware.
- We propose an instruction translation scheme that transforms trap-based rewriting into a static method, enabling its application on embedded devices. Combining it with the abusable exceptions we identified, we build a practical hooking system focusing on bare-metal embedded firmware.
- We develop and open source[1] an out-of-box prototype of PIFER for ARM supporting all Cortex-M microprocessors, which are the most widely used architectures in IoT industry.
- We conduct a comprehensive evaluation on both benchmarks and real-world firmware under diverse devices, as well as a discussion of the applicability and limitations of PIFER.

2 BACKGROUND AND MOTIVATION

In this Section, we discuss binary rewriting for embedded firmware and the limitations of existing work.

2.1 Binary Rewriting for Embedded Firmware

The firmware of an embedded device refers to the software stored in non-volatile memory and is responsible for controlling the application logic of specific hardware, such as microcontrollers. Crafted by different vendors for dedicated tasks, the majority of such firmware is proprietary and not open-source. This situation poses a challenge to both security researchers and downstream developers in the embedded ecosystem, as several important applications require modifications of the target, such as bug fixes, fault observability, performance profiling, and custom feature additions [23]. Direct binary rewriting could accomplish the above tasks without source code but requires extremely delicate construction to avoid breaking the original code and causing errors. Unfortunately, for bare-metal embedded firmware, existing binary rewriters all have limitations in both methodology and implementation.

[1]https://github.com/PIFER-release/PIFER

```
// arm-none-eabi-gcc -c -O2  add.c -o add.o
int add_offset(int a){
    return 0x2000AAAA + a;
}
```

```
add_offset:
0x00 03 46        MOV     R3, R0
0x02 01 48        LDR     R0, [PC, #4]
0x04 18 44        ADD     R0, R3, R0
0x06 70 47        BX      LR
0x08 AA AA 00 20  DCD 0x2000AAAA
```

(a) Mixed code/data in ARM.

```
add_offset:
0x00 03 46          MOV     R3, R0
0x02 01 48          LDR     R0, [PC, #4]
0x04 ?? ??          ;inserted instructions
0x06 18 44          ADD     R0, R3, R0
0x08 70 47          BX      LR
0x0A AA AA 00 20    DCD 0x2000AAAA  Wrong value!
```

(b) After instrumenting.

Figure 1: Example of table-based approach failure on ARM. A PC-relative addressing instruction (at 0x02) is used to fetch data from the literal pools (at 0x10). After inserting the new code into this basic block, the offset between the literal pools and the original addressing instruction is changed and an unexpected value (`0x47704418`) is loaded into `R0`.

2.2 Limitations of Existing Approaches

Existing static binary rewriting techniques can be classified into three categories[16]: (i) detour-based; (ii) table-based; (iii) symbolization/reassembly.

Detour-based Instrumentation Detour-based instrumentation techniques, such as Detours[13], Bistro[4], and E9Patch[8], overwrite the original instruction at the target location to a *jump* which redirects the control flow to a *trampoline* consisting of newly inserted instructions. As identified in [17], this approach struggles to work on embedded architectures due to the short jump range of fixed-length instructions. Take Cortex-M for an example, when the trampoline is more than 2048 bytes away from the target, a long jump (`B.W`, consuming 4 bytes) has to be used, which will not fit into the 16-bit Thumb instructions.

Table-based Instrumentation Table-based approaches such as Multiverse[2], AflIot[7], μSBS[23], and ARMore [5], restore address offsets corrupted by instrumented binary code by redirecting control flow transfer instructions to a pre-computed mapping table. However, this approach assumes that data references remain static. As illustrated in Figure 1, the table-based approach fails to handle the literal-pools feature in embedded architectures like ARM, where the data and code could be mixed in `.text` segment for memory optimization [19]. ARMore is the only work that considers mixed data/code, but their segfault-catching workaround cannot be applied on bare-metal firmware.

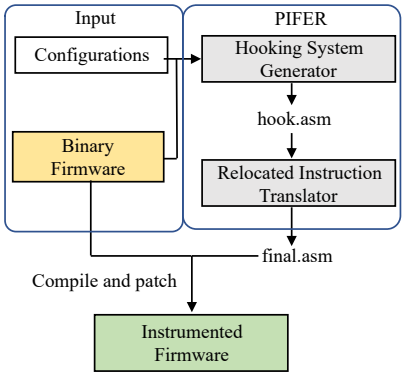


Figure 2: Overview of PIFER's workflow.

Reassembly Reassembly-based technologies such as RetroWrite[6], work by translating the binary into a relocatable IR or even the equivalent assembly produced by compiler[16]. Specifically, the aim is to generate the equivalent assembly code with symbolic labels as the compiler produces. However, recovering symbolic labels from the binary is difficult in theory. As pointed out by recent work[16], all existing reassembly-based frameworks fail to achieve the soundness they claimed.

3 DESIGN

Our aim is to perform *practical* static binary instrumentation for bare metal firmware of embedded devices, with the following requirements:

(1) **Sound:** The firmware must be working correctly after patching.
(2) **Fine-grained:** Instrumentation at arbitrary binary positions.
(3) **Scalable:** Methodologically covers as many embedded devices as possible.
(4) **Efficient:** Fast instrumentation of the original firmware without excessive human effort and knowledge.
(5) **Real-world Usable:** Works out of the box in the raw binary-only context (i.e., without any compiler features or debugging information), which is typical in real-world analysis.

In the rest of this Section, we will first introduce the overview of PIFER's workflow together with the structure of the hooking system, and then dive into the details that support them.

3.1 Overview

As illustrated in Figure 2, the workflow of PIFER can be broken down into three steps:

- **Step 1: Generating the hooking system.** The input of PIFER consists of the target binary firmware and a configuration file. The configuration contains information about the base address and architecture of the firmware, the offset of the exception vector table (EVT)[1], and the instrument targets with new code to be inserted. PIFER will then parse the raw firmware and generate a self-contained hooking system in assembly (Section 3.2).
- **Step 2: Translating the modified instructions.** Next, PIFER will translate the instructions at instrumenting locations to their relocated equivalents and incorporate them into the hooking system thus ensuring correctness (Section 3.3).
- **Step 3: Binary patching.** In this step, the assembly generated in Step 2 is compiled and appended to the original firmware. Next, PIFER modifies the addresses of the selected exception handler in the vector table according to the configuration input (Section 3.4).

[1]We assume that these attributes are already known to the user, which is the most basic step in analyzing a binary firmware and is facilitated by tools like Binwalk[18].

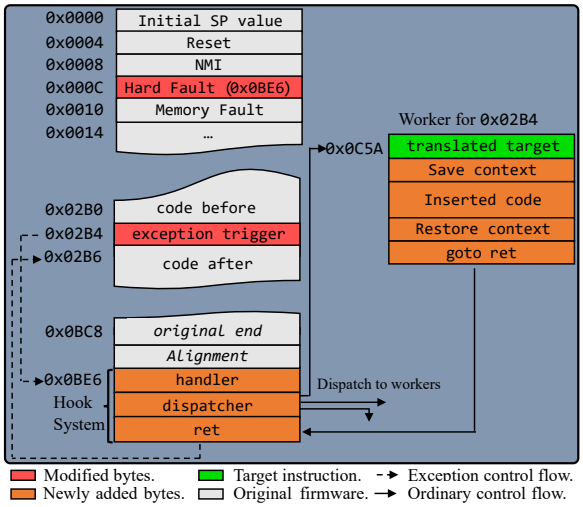


Figure 3: The designed hook system for ARM architecture.

3.2 The Hooking System

Driven by the processor exception handling mechanism, PIFER constructs a self-contained hooking system as illustrated by the example in Figure 3. The instruction at the target address is replaced with an exception-raising one and the corresponding handler is modified to the handler of the hook system (detailed in Section 3.4). As the program executes to the target address instruction, it will be redirected to the following procedure:

- **handler:** This step will save the information in the exception context, including the exception occurred address and the registers state. It will determine whether the exception was intentionally raised by the PIFER, and thus choose whether to continue to the **`dispatcher`** or fall back to the original exception handler.
- **dispatcher:** In this step, according to the address where the exception occurred, a jump will be made to the corresponding instrument code, which is referred to as **`worker`** in our design.
- **worker:** The worker will first execute the translated instruction, thus ensuring that the overwritten instruction (the original one at **`0x02B4`** in Figure 3) still executes correctly. It then saves the program context and executes the newly inserted instructions. Afterward, it restores the context and returns to the **`ret`** module.
- **ret:** Finally, we update the new context to the stored pre-exception one and return to the original next instruction.

Thus, we build a self-contained hooking system that operates simply by utilizing the exception-handling mechanisms common in most embedded architectures. However, as we discussed earlier, there are two critical challenges to make it truly operational: (**RC1**) correctly re-execution of the overwritten instruction, and (**RC2**) arbitrarily triggering desired exception within 2 bytes.

3.3 Translating the Overwritten Instructions

Considering the combination of different operands and operations, the number of valid 16/32 bits instructions is quite large. Therefore, it is a tedious challenge to correctly handle all the possible overwritten instructions for general fine-grained rewriting. To address it, we first group all possible instructions into the following two classes:

- **C1: Instructions not involving PC register.** For example, arithmetic operations and data transfer instructions.
- **C2: Instructions involving PC register.** Examples include control flow-related instructions (e.g., jumps, call to/return from functions), and PC-relative addressing.

For class C1, relocation of these instructions does not impact the runtime effect, thus simple replay suffices. For instructions in C2, we propose a *register proxy* method to build transformed instructions producing the same effect as the original ones. The pseudo-code in Algorithm 1 describes how PIFER constructs the assembly code to handle C2 instructions from a high-level perspective, taking the target instruction, its location **context**.*PC*, and the address of the original next instruction **context**.*RA* (i.e., where the handler should return after processing the exception). Specifically, we first convert all C2 instructions into their equivalents but *explicitly* using the PC register (line 4). For example, the PC-relative address generating instruction in ARM **ADR Rd label** can be transformed to **MOV Rd, PC; ADD Rd, Rd, label**. In our implementation for ARM architecture, the rest C2 instructions that non-explicitly use the PC are transformed with the following rules:

- **B, BL, BX, and BLX**: The branch instructions can be universally represented as **B{L}{X}{cond} label**/**Rn**. Noting that it is actually a conditional assignment to the PC instruction, we can use the **B{cond}**+**LDR**+**MOV** instruction to translate it. For the branch with link instruction **BL**, an additional step in the handler is required to set the **LR** register in the saved context to the address of the next instruction.
- **CBZ, CBNZ**: The **CBZ/CBNZ** instruction does not change condition flags but is otherwise equivalent to a combination of **CMP**+**BEQ**. We apply this solution directly by saving all flags (**xPSR**) on the stack before execution and restoring them afterward.
- **TBB, TBH**: The **TBB** [**Rn**, **Rm**] instruction can be broken into two steps: 1) Get a byte at the address of **Rn**+**Rm**. 2) Add the byte to the **PC**. Correspondingly, our translation can be split into two parts: first use the proxy register **Rx** to get the value of that byte(**LDRB Rx**, [**Rn**, **Rm**]), then add its value to the **PC** stored in the context by a dedicated **Worker** under handler mode. The same strategy also applies to **TBH**, which is just a half-word version of **TBB**.
- **IT**: The If-Then instruction can control up to 4 subsequent instructions. However, it does not directly alter the control flow but rather informs the processor whether each controlled instruction should take effect based on the flag bits. Since the **xPSR** storing the flags is automatically saved in the exception stack, we can simply treat the **IT** as a PC-independent instruction.

Then, we search for a register **rx** that is not used by the overwritten instruction (lines 5-10). The searching implementation could be improved in the future as it may fail when a single instruction occupies *all* available registers. However, we did not find any such instructions in real-world firmware in our experiments. Next, we

Input parameters:
- **ins**: The overwritten instruction at target location;
- **context**: All exception contexts saved by the handler, including:
  - *PC*: the address where the exception occurred;
  - *RA*: the address where we return after processing;

Local variables:
- **asm**: The constructed assembly code for handler of C2;
- **ce**: The code that explicitly uses the PC register;
- **rx**: A register not used in the overwritten instruction;

Figure 4: Definitions of variables used in Algorithms 1.

Algorithm 1 Generating equivalent assembly for C3 instructions

```
1:  function ConstructC3Handler(ins, context)
2:      asm := empty string
3:      rx := Null
4:      ce := Translate(ins)              ▷ Translate to assembly that explicitly using PC register.
5:      for r in {All general registers} do          ▷ Find a proxy register not used by ce.
6:          if r not in ce then
7:              rx := r
8:              break
9:          end if
10:     end for
11:     ce = Replace(ce, "PC", rx)                   ▷ Replace all "PC"s in ce with rx.
12:     asm.append(SaveRx(rx))                       ▷ Backup the content in rx.
13:     asm.append(ContextReadPC(context.PC, rx))    ▷ Overwrite the content in rx.
14:     asm.append(ce)                               ▷ Translated target instruction.
15:     asm.append(ContextWriteRA(rx, context.RA))   ▷ Set the new target address.
16:     asm.append(RestoreRx(rx) )                   ▷ Restore the original content in rx.
17:     return asm                                   ▷ return the constructed assembly code.
18: end function
```

replace the literal "PC" in the translated assembly with the **rx** (line 11,14) and add the exchange code between **rx** and **context**.*PC* in the saved context before and after it (lines 12-13, 15-16). The subprocedures SaveRx, RestoreRx, ContextReadPC, ContextWriteRA will automatically generate the assembly code for swapping in and out the content of **rx** and **context**.*PC*/**context**.*RA*. In other words, we use **rx** as a *proxy* for the original **PC**, thus avoiding the incorrect execution after relocation thereby addressing **RC1**.

3.4 Exception Raising and Binary Patching

The hook system in Section 3.2 requires patching the target instruction with a designated exception-causing one and subsequently adjusting the corresponding EVT handler function. We conduct a thorough research and finally choose the *illegal instruction* exception for the following reasons:

- **Size:** A modification within only 2 bytes is enough to make a normal instruction illegal.
- **Universal:** Architectures such as ARM and RISCV have "standard" undefined instructions that can reliably trigger exceptions. For example, the opcode pattern **0x00 0xDE** will raise an exception across ArmV6-M, Armv7-M, and Armv8-M devices.
- **Priority:** The undefined/illegal instruction exception typically has a high priority and could be further elevated to **Hard Fault**, which is desirable since no regular exceptions or interrupts may preempt the hooking system.

After modifying the target instruction and corresponding handlers in the EVT, PIFER compiles the assembly produced by the previous steps to binary form and appends it to the original firmware. The layout of the instrumented firmware is also illustrated in Figure 3 (best viewed in color).

Table 1: The evaluation settings.

Item	Setting
OS Version	Zephyr 3.2.99
Toolchain	GNU Arm Embedded Toolchain 12.2.0
Applications	Blinky, Button, Basic Thread Example, File system shell, Logging, TinyCrypt

Table 2: Target platforms

Device MCU	Architecture	Frequency	RAM/FLASH
STM32L073	Cortex-M0	32MHz	20KB/192KB
STM32F103	Cortex-M3	72MHz	20KB/64KB
nRF52840	Cortex-M4	64MHz	256KB/1MB
LPC1549	Cortex-M3	72MHz	36KB/256KB
LPC55S69	Cortex-M33	150MHz	320KB/640KB

Table 3: Instructions that PIFER cannot instrument.

Instruction	Description
BKPT	Breakpoint
CPSIE/CPSID	Disable Interrupts/Enable Interrupts
MRS/MSR	Read from/Write to special register
SEV	Send Event
SVC	Supervisor Call
WFE	Wait For Event
WFI	Wait For Interrupt

4 EVALUATION

Our evaluation was conducted to answer the following research questions that support our early claims of PIFER:

- **Correctness**: Is the behavior of the firmware consistent before and after the instrumentation?
- **Performace**: Does its runtime and memory overhead suffice for security applications such as vulnerability patching?
- **Scalability**: Does it scale to real-world, complex firmware?

To answer the above questions, we implement a prototype of PIFER for the ARM architecture using about 2,000 lines of Python code and carry out diverse measurements and case studies.

4.1 RTOS instrumentation on multiple platforms (Correctness, Scalability)

Experiment settings. To investigate the scalability of PIFER, we perform instrumenting experiments against a non-trivial, representative embedded firmware: the Zephyr RTOS [11], on different devices. The experimental settings and target platforms are shown in Table 1 and Table 2. In the experiment, multiple sample applications from the Zephyr project are collected for a comprehensive evaluation. For each sample firmware, we instrument all of its instructions with a batch size of 1000. We use IDA [12] to extract the address of each instruction from the compiled firmware and instrument them using PIFER with `NOP` payload. An external debugger is used to reflash the patched firmware into the target device and monitor the execution. When the program fails to execute correctly it will fall back to the original Hard Fault handler, which can be confirmed by a debugger script.

Result and analysis. The results show that the overall correct instrumenting rate of PIFER was 98.9% on all platforms. Specifically, the instructions that PIFER could not handle are listed in Table 3. In general, these instructions also work through interrupts/exceptions but cannot preempt the priority of the hook system. Nevertheless, as all these instructions have dedicated functions and are not typically the target of instrumenting for security purposes, their impact on the practicality of PIFER is negligible.

4.2 Measurement of the Overhead (Correctness, Performace)

Experiment settings. CoreMark [9] is a standardized benchmark suite designed to measure the performance of microcontrollers using a set of representative workloads, including matrix operation, state machine, linked list manipulation and CRC checksum calculation. Our target platform is the LPC55S69 from Table 2. The CoreMark implementation followed the official guide [25] and is also open-sourced in our GitHub repository. The device under test will pull up the output levels of two pins before and after the benchmark test, with an oscilloscope to measure the time. In each experiment, we use PIFER to randomly instrument N instructions in the firmware with an empty payload, and a counter is integrated into the hook system to monitor the number of received traps.

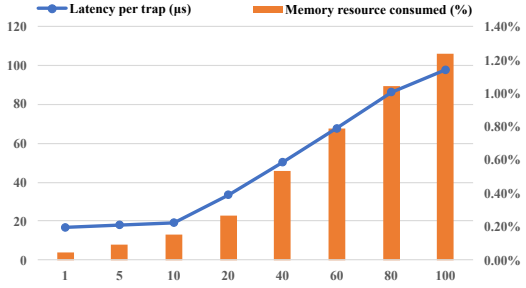
Result and analysis. The results are shown in Figure 5a. The blue line represents the average latency caused by processing each trap, obtained by dividing the total time overhead by the number of received traps. The bar chart represents extra memory consumed by the instrumentation, including the increase of the binary firmware size and the stack space occurred by the hook system. From the results, we find that the performance loss of the PIFER rises with the number of instrumented instructions, and each additional hooking point will bring an overhead of $\sim 1\mu s$. We attribute this phenomenon to the $O(N)$ lookup complexity in our dispatcher. Implementing a more efficient algorithm could help improve this issue, but requires trade-offs between time and space overheads. In general, PIFER has the ability to correctly instrument the firmware at thousands of addresses. For tasks requiring only dozens of instrumenting points at certain functions, such as vulnerability patching or performance profiling, the overhead of PIFER is negligible.

4.3 Case Study: Real-world Firmware Customization (Correctness, Scalability)

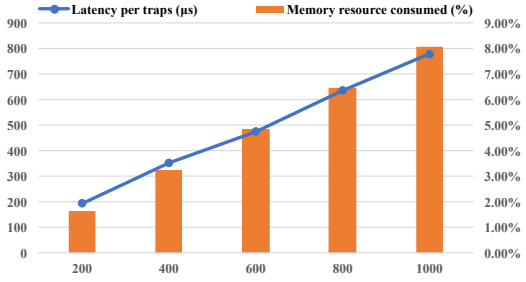
Firmware acquisition and basic analysis. To further demonstrate PIFER's ability to handle real-world complex firmware, we apply it to the Apple Airtag [14], a product for which no source code is available. For the sake of reproducibility, our experiments were conducted with a brand-new device with firmware version 1.0.225. We gain the dump and flash ability on Airtag using a public glitch tool [22] and extract the binary firmware[1]. After a manual analysis, we identify the location of the EVT and recover some of the application logic. To verify PIFER's ability for firmware customization, we apply it to insert a proof-of-concept protocol into the firmware, which leaks the long-term key (LTK) for Bluetooth communication[24] through a pin shown in Fig. 6a.

Results. After patching the key leak function into the firmware and re-flashing it back to an Airtag, we connect the corresponding pin to an oscilloscope to visualize the output. As shown in Fig. 6b, the first byte of LTK was successfully encoded in the voltage level, proving the practicality and correctness of PIFER.

[1]Note that we are not trying to "attack" the Airtag, but verifying PIFER's ability to handle unknown firmware. Thus, the glitch tool itself is not our focus.

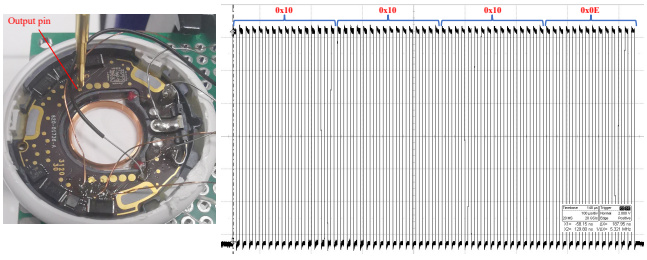


(a) Small scale: N=1-100



(b) Larger scale: N=200-1000

Figure 5: CoreMark experiment results.



(a) The pin adopted. **(b) Leaking the first byte of LTK (0x3E).**

Figure 6: Experiment with Airtag.

5 LIMITATIONS AND FUTURE WORK

More robust instruction translation. As noted in Section 3.3, although no samples are observed in real-word firmware, the current free register searching has corner cases in theory. Improvements to this scheme are left to future work, such as identifying usable registers with advanced program analysis.

Improvements in dispatchers. We currently use an $O(n)$ lookup for finding the handler for a specific address, and the performance will degrade as the number of hooks increases. In the future, we can explore more efficient implementations.

6 CONCLUSION

In this paper, we present the design of PIFER, a static binary instrumenting tool for bare-metal embedded firmware. By abusing the exception handling mechanism, PIFER builds a self-contained hooking system providing a fine-grained instrumenting functionality. We also propose an instruction translating scheme to overcome the unrecoverable modifications in FLASH code, ensuring that the firmware works correctly after patching. Moreover, we implement and open source a full prototype of PIFER on the ARM architecture. Comprehensive experiments confirm PIFER's capacity to process complex, real-world targets with acceptable overhead, demonstrating the effectiveness and practicality of our work.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No.U2336210 and the Startup Fund for Young Faculty at SJTU (SFYF at SJTU) under Grant No.24X010500123. Chi Zhang and Dawu Gu are the corresponding authors.

REFERENCES

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*. IEEE, 1362–1380.
[2] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.
[3] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2875–2889.
[4] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. Bistro: Binary component extraction and embedding for software security applications. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer, 200–218.
[5] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. 2023. ARMore: Pushing Love Back Into Binaries. In *Proceedings of the 32nd USENIX Security Symposium*.
[6] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.
[7] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. 2022. AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation. *Computers & Security* 122 (2022), 102889.
[8] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 151–163.
[9] EEMBC. 2009. *MCU Benchmark, CoreMark*.
[10] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. 2022. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity* 5, 1 (2022), 18.
[11] Linux Foundation. 2016. *Zephyr: a new generation, scalable, optimized, secure RTOS for multiple hardware architectures*.
[12] Hex-Rays. 2023. *IDA Pro: A powerful disassembler and a versatile debugger*.
[13] Galen Hunt and Doug Brubacher. 1999. Detours: Binaryinterception ofwin 3 2 functions. In *3rd usenix windows nt symposium*.
[14] Apple Inc. 2021. *AirTag*.
[15] JSOF. 2020. *19 Zero-Day Vulnerabilities Amplified by the Supply Chain*.
[16] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. 2023. Reassembly is Hard: A Reflection on Challenges and Strategies. In *Proceedings of the 32nd USENIX Security Symposium*.
[17] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A platform-agnostic ARM binary rewriter for security applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 412–424.
[18] ReFirm Labs. 2010. *Binwalk: Firmware Analysis Tool*.
[19] Arm Ltd. 2016. *Literal pools, Arm Compiler armasm User Guide, Version 6.6.5*.
[20] Damiano Melotti Maxime Rossi Bellom and Philippe Teuwen. 2021. *Blackhat USA 2021: A Titan M Odyssey*.
[21] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*. Springer, 259–284.
[22] Thomas Roth. 2021. *Airtag glitcher*.
[23] Majid Salehi, Danny Hughes, and Bruno Crispo. 2020. μSBS: Static binary sanitization of bare-metal embedded devices for fault observability. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, 381–395.
[24] Karen Scarfone, John Padgette, et al. 2008. Guide to bluetooth security. *NIST Special Publication* 800, 2008 (2008), 121.
[25] NXP Semiconductors. 2020. *Running coremark benchmark with dual CM33 cores and PowerQuad on LPC5500*.
[26] Lennert Wouters, Benedikt Gierlichs, and Bart Preneel. 2021. My other car is your car: compromising the Tesla Model X keyless entry system. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 149–172.