FirmPorter: Porting RTOSes at the Binary Level for Firmware Re-hosting

Mingfeng Xin[1,2], Hui Wen[1], Liting Deng[1,2], Hong Li[1], Qiang Li[3], and Limin Sun[1](✉)

[1] Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China
{xinmingfeng,wenhui,dengliting,lihong,sunlimin}@iie.ac.cn,
liqiang@bjtu.edu.cn

Abstract. The rapid growth of the Industrial Internet of Things (IIoT) has brought real-time operating system (RTOS) into focus as major targets for both security analysts and malicious adversaries. Due to the non-standard hardware and diverse software, embedded RTOS devices present unique challenges to security analysts for the accurate analysis of firmware binaries. The diversity in hardware components and tight coupling between firmware and hardware makes it hard to perform dynamic analysis, which must have the ability to execute firmware code in virtualized environments. However, emulating the large expanse of hardware peripherals makes analysts have to frequently modify the emulator for executing various firmware code in different virtualized environments, which greatly limits the ability of analysis.

In this work, we explore the problem of firmware re-hosting related to the RTOS. A device driver is developed by developers so that RTOS can be run on their platform. By providing alternative implementations for device drivers, we can make minimal modifications to the firmware that is to be migrated from its original hardware environment into a virtualized one. We show that an approach is capable of offering the ability to emulate various RTOS firmware in an automated manner without modifying existing emulators. Our approach, called *static binary-level driver porting*, first locates device driver initialization function and identify driver types in the target firmware, then adapts pre-built drivers to the existing emulator hardware. Finally, it replaces the drivers in the firmware with ours by utilizing binary rewriting technique. We demonstrate the practicality of the proposed method on multiple hardware platforms and firmware samples for security analysis. The results show that the approach is flexible enough to emulate firmware for vulnerability assessment and exploit development.

Keywords: Firmware · Re-hosting · Dynamic Analysis · Vulnerability assessment.

1 Introduction

Embedded devices, such as routers and control devices, are being used in an increasing number of industrial applications. These devices are often connected to diverse

networks to augment their functionality while performing specialized computing tasks collaboratively. The increased connectivity of embedded devices, however, has significantly increased their vulnerability to attacks. Therefore, analysts have concentrated on the analysis of embedded device firmware to uncover vulnerabilities for system security assessment and firmware patching.

Dynamic analysis is particularly critical when researchers aim to conduct an exhaustive security analysis of specific firmware. As far as we understand, dynamic analysis can perform a wide range of intricate examinations, including taint analysis and symbolic execution, and overcome the limitations of static analysis, such as code that is packed or obfuscated. In terms of firmware analysis, dynamic analysis of embedded systems can be used to analyze intricate data flows, increasing the capability of analyzing embedded devices for vulnerabilities in a significant manner.

Unfortunately, embedded hardware provides limited introspection capabilities, including limited numbers of breakpoints and watchpoints, significantly restricting the ability of dynamic analysis on firmware. In this case, emulation, also known as firmware re-hosting, enables the host system to execute the firmware of embedded hardware in virtualized environments for successfully performing dynamic analysis on firmware.

Re-hosting firmware, however, poses a number of challenges[39][14]. First, the re-hosting of firmware requires successful emulation of the hardware peripherals. The incorrect emulation of the hardware may result in false results or even a crash. Besides, the number of peripherals is huge. It is impractical for emulators to support each peripheral. Moreover, the firmware information is hard to retrieve. Due to the lack of standardization in firmware development, it is necessary to get firmware information, e.g., memory layout, to make it run correctly in emulators. Therefore, re-hosting firmware is still an open research question.

Previous researches [40][30][26][24][23][35][27] solve the problem of successfully executing hardware-related code by forwarding peripheral operations to a real device. It has been demonstrated that these works enhance the ability of firmware re-hosting, however they are limited in scale since they require real hardware devices. Several researchers[18][16] [29][33][34][28] study the interaction between peripherals and firmware, and then create models for responding to further interactions with hardware. For fuzzing purposes, these models work well, however, they are not suitable for debugging firmware for the purpose of developing exploits or patches. Another approach is re-implementing hardware functions to provide equivalent functionality[9][19][21]. These works provide a flexible method for bypassing the problem of running unsupported code in firmware, but the process is currently time consuming and labor-intensive. Other approaches, such as symbolic execution[5][32][6][22][41], face traditional challenges like path explosion, which makes them not practical for re-hosting complex firmware.

Key Insights We have three key insights that can help us address the three challenges outlined above. First, different firmware can run correctly using the same configurations if their drivers are replaced with the same ones. Second, modern RTOS provide Device Driver Infrastructure for porting drivers, which provides a standard interface for replacing or installing a new driver. Third, while emulators like QEMU support only a limited number of peripherals, they have a complete range of drivers available. In this

case, different firmware with the same type of driver can be replaced with the same emulator-supported driver, such as Ethernet driver.

Our Approach Based on these insights, we address the re-hosting problem by considering firmware re-hosting as porting firmware to the emulator's board. We propose a method called *static binary-level driver porting* and develop a tool named FirmPorter, which modifies driver code at the binary level. The tool automatically substitutes the drivers with suitable ones, which allows the emulator to execute them effectively in a virtual environment.

However, porting a driver to a firmware is nontrivial. First, it is difficult to identify drivers. Because the firmware is usually stripped, drivers cannot be identified based on their names. Second, the drivers for replacement need to be adapted, which includes determining the placeholder in memory and relocating symbols. Otherwise, it may affect the firmware at runtime. Third, the methods of rewriting binary files differ based on the CPU architecture.

We developed and implemented a new algorithm to identify drivers by binary similarity matching to overcome the difficulties associated with porting drivers. Then, we automatically determine the address of the drivers in memory and relocate external symbols. Lastly, we develop a prototype called FirmPorter that implements the driver porting technique at the binary level.

Contributions:

1. We propose a novel technique, *static binary-level driver porting*, that can re-host different RTOS firmware without modifying emulators.

2. We address concrete challenges of static binary-level driver porting, and implement a practical prototype, FirmPorter.

3. We demonstrate the practical application of FirmPorter for firmware re-hosting in our experiment. FirmPorter shows that proofs-of-concepts/exploits developed in the emulator, which runs firmware that has been modified by it, can be applied directly to real-world devices.

2 Background

In this section, we first introduce how RTOS firmware is emulated in QEMU. Then we briefly introduce the Device Driver Infrastructure and the Device Driver Initialization Function that exist in modern RTOSes, which inspires us to solve the problem of re-hosting in a novel way.

2.1 Emulating RTOS Firmware

QEMU[4] is a famous open-source emulator developed by Fabrice Bellard for hardware virtualization. QEMU emulates a machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, which makes it capable of running a variety of guest operating systems.

When emulating RTOS firmware in QEMU, the firmware initiates accurately on a virtual processor of identical model, with hardware-independent code translated for

host execution. Hardware-specific code accesses the memory-mapped I/O (MMIO) address, aligned with peripheral registers. QEMU manages MMIO response logic during peripheral emulation within defined ranges.

2.2 Device Driver Infrastructure

RTOS has gradually developed device driver infrastructures as Linux has gained popularity and embedded device performance has improved. A device driver infrastructure provides a set of standard interfaces that can make driver development easier. The maintainer of the driver designs a set of mature, standard, and typical driver implementations for each type of driver, and extracts parts of the same hardware drivers from different manufacturers to implement them on their own, leaving the different parts out of the interface for the specific engineer to implement. For example, all upper-level functions call the `read` function, and no longer distinguish between `uart_read` or `spi_read`.

2.3 Device Driver Initialization Function

The *device driver initialization function*, or *device driver entry point*, is a crucial component within the realm of operating systems that allows the OS to recognize and load the corresponding device drivers. This function pointer serves as a starting point for the installation, configuration, and initialization of the device driver when it is loaded into the operating system's kernel. The device driver initialization function is part of the device driver infrastructure. Functions initialized there are called *driver register functions*.

3 Challenge and Solution

The goal of this work is to re-host firmware using RTOS in general emulators such as QEMU, which lays the foundation for vulnerability assessment and exploit development.

3.1 Challenges

Hardware dependency Firmware often relies on direct access to hardware components, which can be challenging to replicate in an emulated environment. Besides, firmware execution trace is usually determined by the value read from the hardware peripheral. If peripherals are not properly emulated in emulators, the re-hosting process may misbehave, stall or crash.
Heterogeneity of hardware Operating environments for firmware are diverse and complex. First of all, there are numerous processor architectures, including widely used platforms such as ARM and PowerPC, as well as niche platforms such as Sparc and SH4. Further, some firmware utilizes Field Programmable Gate Arrays (FPGAs), making it difficult for existing emulators to support these architectures comprehensively. Moreover, even within the same CPU architecture, different System-on-Chip (SoC) designs require emulators to develop peripheral models. Most importantly, there is a large

number of hardware peripherals used in firmware operating environments, making it extremely difficult to accurately emulate each of them. These factors contribute to significant difficulties in firmware emulation.

Lack of information The analysis of firmware emulation requires basic hardware information, such as the CPU architecture and memory address layout. However, most firmware runs on customized SoCs from manufacturers, and the firmware lacks processor information, which makes emulation a difficult task. Hardware device disassembly can often provide device CPU information, but the substantial reliance on physical devices restricts the scope of firmware that can be emulated and analyzed.

3.2 Our solution

To address the aforementioned challenges, our approach includes:

- Precisely emulating hardware peripherals to support firmware emulation.
- Despite the theoretically infinite number of peripherals to emulate, we leverage a finite subset—those supported by QEMU—to provide requisite hardware responses for firmware execution.
- By exploiting the structural nature of the RTOS, we substitute firmware drivers of similar types with those compatible with QEMU, thereby achieving emulation of RTOS firmware within QEMU, which is called *static binary-level driver porting*.

4 Design

4.1 Overview

Solution Overview

We present the design of our proposed approach. As depicted in Fig. 1, the overall workflow is composed of three steps. **(1) Locating Device Driver Initialization Function & Identifying Driver Types.** We locate device drivers by checking those that are registered in the device driver initialization functions. First, we locate the device driver initialization function in stripped binaries by deducing from identified anchor functions. Then, we identify driver types in the firmware in a rule-based way. **(2) Generating Drivers.** We relocate symbols in original driver object file and produce proper driver functions to replace the original ones in firmware, ensuring that these drivers operate as intended on the emulator's board. **(3) Static Binary-level Driver Porting.** We port the driver functions in the object file to the firmware through function redirection. It is important to note that porting strategies differ based on the target CPU architecture.

We have implemented our technique in a tool called *FirmPorter*, which consists of 210 lines of Python code for identifying driver functions, 1046 lines of Python code for porting and integration. Additionally, we have written 95 lines of IDAPython[20] scripts for firmware information extraction. The porting code primarily utilizes the pwntools[1] for binary rewriting and the LIEF[31] library for ELF file manipulation.

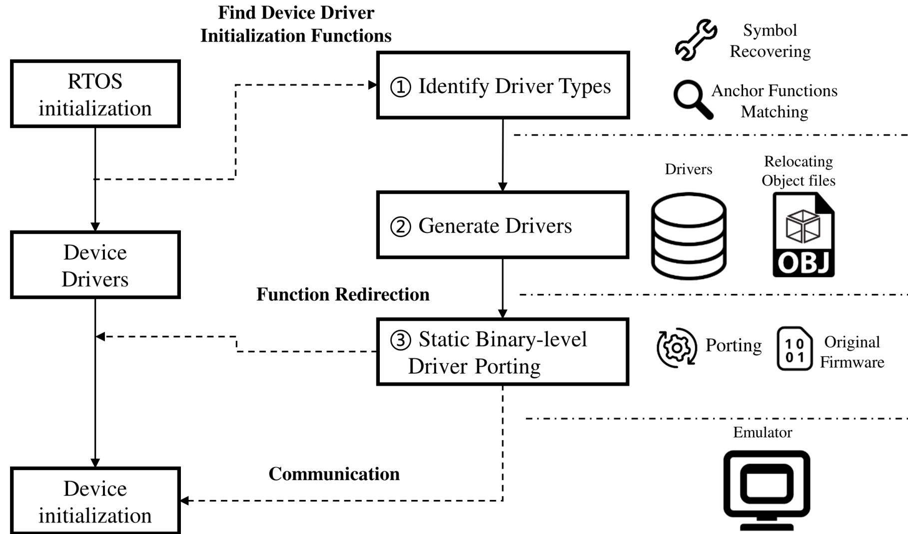

Fig. 1: Solution Overview

4.2 Locating Device Driver Initialization Function & Identifying Driver Types

Locating Device Initialization Function We first introduce the concepts of *Anchor Function* in FirmPorter.

Definition 1. *Anchor Functions are a set of functions that form the fundamental structure of an RTOS. Their name and calling relationship remain consistent across any firmware utilizing the same RTOS.*

Anchor functions are typically either standardized or defined by the vendor, and they include common BSP(Board Support Package) functions, kernel functions, device driver infrastructure functions, etc. An anchor function has the property of only being called once in a fixed location, resulting in a deterministic calling relationship. Specifically, anchor functions have a fixed set of caller functions and callee functions. It should be noted that the device driver initialization function belongs to anchor functions. In this case, we find anchor functions continuously until we find the device driver initialization functions.

Identifying Anchor Functions in Firmware Depending on the circumstances, we identify anchor functions in firmware in two ways: 1. When symbol information is preserved in the firmware, it is possible to recover symbols of functions for the purpose of distinguishing anchor functions. 2. Matching and deducing anchor functions when symbols have been removed from the firmware.

a) by Recovering Symbols To recover function symbols, it is necessary to identify the symbol table structure and the location where the symbol table begins. In a specific RTOS, the symbol table is immutable and can be analyzed from the source code. Researchers can search the area following the code section to find the symbol table's

starting point. Upon recognizing all functions in the firmware, we can directly identify the anchor functions.

b) by Lightweight Matching & Deducting Algorithm This algorithm assists researchers in matching anchor functions from a known binary with symbols, which can be collected or compiled by the researcher, and then deduce the anchor functions by reusing a classic binary diffing tool, Diaphora[25]. Diaphora is an open-source tool widely used in program diffing, which helps us locate functions by comparing them with a known binary containing symbols.

First, we employ Diaphora to conduct a conventional binary comparison of the target firmware with the firmware that contains symbol information. We only select the results of *Best Match* and *Partial Match* with a ratio greater than a threshold, such as 0.75. Then, we separate the anchor functions from the results into a set. Using this set as a starting point, we infer the caller of each function in the set iteratively and add these caller functions to the set. Finally, we traverse each anchor function and deduce its unknown callee function by comparing the number of basic blocks, control flow edges, and function calls. This algorithm significantly expands the scope of anchor functions we find in the firmware, and device driver entry functions can be easily identified.

Algorithm 1 Matching & Deducting Algorithm

1: $t = FirstMatching$
2: $T = \varnothing$
3: $d = \varnothing$
4: **while** t is not empty **do**
5: $\quad T = T \cup t$;
6: $\quad$ **for all** s such that $s \in t$ **do**
7: $\quad\quad d = d \cup Caller(s)$;
8: $\quad$ **end for**
9: $\quad t = d$
10: **end while**
11: **for all** s such that $s \in T$ **do**
$\quad$ // DFS means Depth-First-Search
12: $\quad DFS(s)$ and do simple matching;
13: **end for**

Identifying Driver Types By traversing each driver register function in the device driver initialization function, we identify the types of drivers used in the firmware. Following the Call Graphs of driver initialization routines, we examine each function associated with a driver and look for signs that can identify that driver: *Strings* and *Magic numbers*. It is common for drivers to include prompt messages that help engineers identify errors during debugging. Furthermore, when a researcher knows the processor on which the firmware runs, they can search for magic numbers that appear in the driver and are read from or written to as addresses. These numbers could potentially be ad-

dresses in the Memory-mapped I/O (MMIO) area, and their corresponding peripherals can be determined from the chip datasheet.

Due to the diversity of drivers and the numerous ways in which they can be implemented, identifying drivers is not an easy task. In this part of the work, the recognition rate is influenced by heuristic rules, which are based on empirical data.

4.3 Generating Drivers

Collecting Drivers Typically, drivers for specific hardware are collected from online sources such as vendor websites, vendor forums, and GitHub. Usually, official websites only provide drivers for newer hardware models, while drivers for older, discontinued hardware can be found on official forums or GitHub. Verification of the authenticity of these drivers can be confirmed by examining the logs within the driver packages. Besides, it is common for drivers to be stored in object files with ELF format(Executable and Linkable Format). The object file contains machine code and data, symbol table, and relocation information. When we get the source codes of a driver, we modify and compile them into the target object file.
Modifing Drivers FirmPorter adapts firmware by relocating external symbols in the object file. Typically, object files contain external symbols, such as library functions like `malloc`, `bcopy`, `memcpy`, etc., that necessitate further relocation. For instance, an instruction in an Arm-based object file is ”`BL malloc`”, with the machine code ”`00 00 00 EB`”. The address of `malloc` is unknown during compilation, so the compiler employs a placeholder offset `0x000000`. To relocate these symbols, researchers must first choose a memory location for the object file and then adjust the placeholder offsets in the instructions.

To maintain object file integrity at runtime, it must not overwrite memory spaces used by the firmware, such as `.text` segments, `.data` segments, `.bss` segments, heap space, etc. Therefore, the object file can be placed after the heap space. It is important to note that the heap space boundary can be parsed from the firmware or defined in the BSP routines for replacement.

Based on the starting address of the functions in the object file, we calculate the offset between the instruction to be modified and the symbols it contains. As FirmPorter traverses the object file, the offsets in instructions are corrected. After the object file has been processed, it can be immediately loaded into memory without requiring any further modifications.

4.4 Static Binary-level Driver Porting

Static binary-level driver porting involves redirecting control flow from original functions to the correct ones. In this part, we introduce two common function redirection strategies used in FirmPorter.
Strategies for Function Redirection

a) Modifying Function. FirmPorter changes the first instruction in the firmware to jump to the correct one in the object file. The modification strategies differ according to the architecture of the CPU. For illustration purposes, we use examples from x86-based and Arm-based firmware in this section.

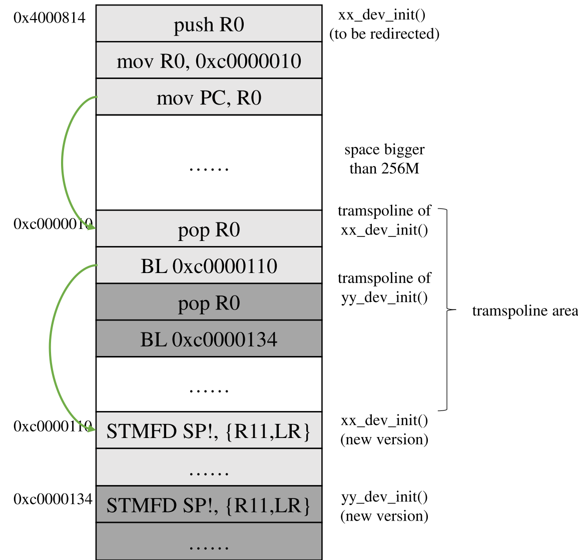

Fig. 2: Static binary-level driver porting on Arm processor with trampolines

In the x86 instruction set, the `jmp` instruction jumps to an absolute address. Specifically, the *far jump* instruction is used to access any location within 0x00000000∼0xffffffff, with the machine code of *far jump* being `0xea`. The redirection strategy for x86-based firmware involves changing the first instruction of the function, usually `push ebp`, to `jmp 0x1234`, where `0x1234` is the address of the function in the object file.

In the Arm instruction set, we employ different strategies based on the characteristics of instructions. Take *AArch32* as an example. There are four unconditional jump instructions in Arm: `B`, `BL`, and their corresponding `BX`, `BLX`, with the latter to change the working mode between *Arm* and *Thumb*. `B` is followed by an offset, equivalent to `MOV PC, [PC+offset]`, similar to the `jmp` instruction in the x86 instruction set. `BL` is also followed by an offset, equivalent to the combination of`MOV LR, BP` and `MOV PC, [PC+offset]`, which is similar to the `call` instruction in the x86 instruction set. Depending on the chip's architecture, Arm's working modes include *Arm mode* and *Thumb/Thumb-2 mode*. In *Arm mode*, the address space is ±32M, while in *Thumb/Thumb-2 mode*, the address space is ±4M. Therefore, if the heap space is too large, the code in the object file will exceed the `B` and `BL`'s address space. This means that we cannot simply jump from one function to another.

In Arm, we use an indirect jump approach, which utilizes a general-purpose register (such as `R0`) as a stepping stone. As depicted in Fig. 2, FirmPorter preserves the value of the register `R0`, then uses the `MOV` instruction to copy the value of `R0` to the register `PC`, enabling it to jump to an absolute address. Next, it creates a *trampoline area* before the object file, where each part contains a trampoline. A trampoline is a small piece of code that is inserted into a program's execution flow to redirect the control flow from the original code to a new location, typically a patch or a replacement function, and

then back to the original code after the new code has executed. Within a trampoline, FirmPorter restores the value of `R0` and calls the appropriate function.

b) Rewriting Function Pointers. FirmPorter rewrites function pointers to call the function in the object file, as some RTOSes use function pointers to invoke initialization routines for improved OS abstraction.

The final step involves concatenating the object file and the firmware. FirmPorter generates the object file in binary format by executing the `objcopy` command in the toolchain. It then fills the `bss` segment, heap space, and other non-allocated areas with zeros. Subsequently, FirmPorter concatenates the object file and the firmware by simply running the *cat* command. Note that FirmPorter must allocate an area for the *trampoline* when handling firmware running on an Arm processor.

5 Evaluation

We evaluated FirmPorter in order to answer the following research questions:
RQ1: Is FirmPorter able to locate device driver initialization functions and identify driver types used in the firmware?
RQ2: Is FirmPorter able to emulate firmware?
RQ3: Is FirmPorter able to provides enough re-hosting fidelity that PoCs/exploits developed under emulation can be applied on real devices?

5.1 Dataset

Our targets include 11 firmware with 4 types of RTOS (VxWorks, RT-Thread, Nuttx, and Zephyr) and run on 5 different processors (x86, Arm Cortex-M3/M4, Arm1176, Arm926EJ-S, and MIPS32). We re-host firmware via 3 open-source emulators: *QEMU*, *xpack QEMU Arm*, and *QEMU for MIPS*. The RTU and PLC samples are based on VxWorks, while the wireless temperature sensor network samples are based on RT-Thread. These firmware run on AMD boards, SPEAr 680 boards and STM32 boards, respectively. We also build 1 firmware that runs on the STM32 board, 2 firmware that run on the NXP board, 1 firmware that runs on the Samsung board, 1 firmware that runs on the Raspberry Pi board, and 1 firmware that runs on the PIC32 board. These firmware contain different vulnerabilities.

5.2 Locating Device Driver Initialization Functions & Identifying Driver Types (RQ1)

In this part, we first evaluate the effectiveness of the algorithm to locate device driver initialization functions(DDIFs). Then we evaluate the accuracy of drivers identified in firmware. Among the firmware we tested, there are 6 firmware that contain a complete symbol table, and these symbol tables are removed before testing.

Locating Device Driver Initialization Functions In this study, we compare the stripped firmware samples with the existing firmware using Diaphora with the same CPU architecture. After obtaining the initial matching result, we perform a secondary

identification of the device driver initialization functions by using the *Matching & Deducting* algorithm. When processing the *NAS* firmware, we directly use the symbol recovering algorithm to restore the symbol table and then identify the device driver initialization functions. The results are listed in Table 1.

Table 1: Results of Locating Device Driver Initialization Functions

Firmware	Syms	Initial	DDIFs	Missing	Final
Music Player	532	120	1	1	2
Car	1031	226	1	1	2
Web Server	811	82	0	5	5
UART Server	1075	413	1	4	5
TCP Echo Server	732	75	0	4	4

Initial is the result of functions matched by Diaphora in the first round. It can be seen that the number of functions in the first matching is relatively small. This is because we need to ensure that the result of the first matching is 100% correct, thus only the results of *Best Match* and *Partial Match* with scores greater than 0.75 are counted. In fact, if we include all the results in the *Partial Match* and the results in the *Unreliable Match* (a few matching results are correct), the number of matching functions for the first time increases by about 20-40%. After manual checks, we confirm the result of the first matching is completely correct.

We use this method to process 5 real-world firmware. The first thing is to load the firmware correctly and generate the corresponding CFG(Control-Flow Graph). This is because the ELF header of the firmware is damaged, or the firmware is a pure binary file. We use the method in [3] to identify functions in the disassembler according to the function prologue and epilogue to generate a control-flow graph. The results are listed in 4.

For the two VxWorks-based firmware, we recovered most of the function symbols, and the device driver initialization functions identified are completely correct. For the remaining 3 firmware used in WSN(Wireless Sensor Network), we can see from the recovery results that the anchor functions are also correctly identified.

Table 2: Results of Recovering Symbols in the VxWorks-based firmware

Firmware	Syms	Recovered	Rate	TP
NAS	7798	6630	85%	100%

The result shows that VxWorks has more device driver initialization functions than those in other RTOSes. This is because VxWorks supports both Legacy device drivers and VxBus device drivers for compatibility requirements, and the two different drivers are registered in separate locations. For example, the disk driver initialization routine

`ataXbdDevCreate` does not initialize in the `hardwareInterfaceBusInit`, because the disk driver does not support VxBus method in VxWorks according to the manual. Therefore, porting disk driver in VxWorks is equivalent to replacing the `ataXbdDevCreate` function with the correct one.

Identifying Drivers We traverse Call Graph of the device driver initialization functions to identify driver types. We check the error message strings present in the driver function to determine the category of the driver, by comparing the strings with those in the database. We classify the driver types into the following 13 categories:

GPIO, UART, I2C, SPI, ADC, PWN, RTC, WATCHDOG, Ethernet, HWTIMER, Sensor, Storage, Other

There are many approaches to identifying which peripheral drivers are used in the firmware. For example, in *RTU*, *PLC* and *NAS* firmware, we can not only look for common names like `fei0`, `sm`, `lnc0` that clearly represent a NIC driver, but also by searching the firmware for strings like *inet on ethernet* or *socket is not connected* to match the `inet` and `socket`, which indicates the firmware uses an Ethernet adapter. We collected a large number of driver names from the open-source code and stored them in the database for heuristic search.

Besides, we deal with a special case when locating drivers in the *PLC* sample, which is based on VxWorks. According to the manual, the ADC and I2C drivers are classified as `Other Classes` in VxWorks, which initializes in the `usrIosCoreInit`. They do not follow any device driver models and exist independently. The results are listed in 3 and 4.

Table 3: Identify Driver Types in Stripped Binaries

Firmware	RTOS	Drivers	Identified
Music Player	RT-Thread	2	2
NAS	VxWorks	3	3
Car	RT-Thread	3	3
Web Server	Nuttx	3	3
UART Server	Zephyr	1	1
TCP Echo Server	Nuttx	2	2

Table 4: Identifying Driver Types in Real Firmware Samples

Firmware	RTOS	Syms	Recovered	DDIFs	Drivers
RTU	VxWorks	19221	17683	7	5
PLC	VxWorks	15757	13968	7	7
WSN-sample	RT-Thread	405	130	2	3
WSN-store	RT-Thread	860	212	2	3
WSN-upload	RT-Thread	514	157	2	3

5.3 Re-hosting Firmware (RQ2)

In this experiment, first, we demonstrate that firmware ported by FirmPorter can run correctly in unmodified emulators. Then we explore the scalability and benefits of static binary-level driver porting.

Re-hosting Capability To evaluate the effectiveness of FirmPorter, we compare the behavior of the firmware on the emulator with that on the real device. Specifically, we consider the firmware re-hosting is successful if each peripheral functionality can perform equivalently on the emulated system as on the real hardware. As firmware's functionality differs, the *success* standards for the execution of each firmware are set accordingly.

First, we list the set-up of the firmware we are going to port, which is listed in Table 5.

Table 5: Firmware for re-hosting

Detailed Information of the Firmware for Testing					Re-hosting Setup	
Firmware	Processor	Architecture	RTOS	Peripherals	Emulator	Template Board
RTU	AMD LX-800	x86	VxWorks	Ethernet, Disk, UART, ADC, GPIO	QEMU	i440FX
WSN-sample	STM32F103	Cortex-M3	RT-Thread	RF Transceiver, UART, ADC	xPack QEMU Arm	STM32F429
WSN-store	STM32L475	Cortex-M3	RT-Thread	RF Transceiver, UART, SD Card	xPack QEMU Arm	STM32F429
WSN-upload	STM32F429	Cortex-M4	RT-Thread	RF Transceiver, UART, Ethernet	xPack QEMU Arm	STM32F429
Music Player	STM32F401	Cortex-M4	RT-Thread	UART, GPIO	xPack QEMU Arm	STM32F429
PLC	SPEAr 680	Arm926EJ-S	VxWorks	Ethernet, SD, UART, ADC, USB, CAN, GPIO	QEMU	versatilepb
NAS	S3C6410	Arm 1176	VxWorks	Ethernet, Disk, UART	QEMU	versatilepb
Car	BCM2835	Arm 1176	RT-Thread	GPIO, Zigbee, UART	QEMU	versatilepb
Web Server	LPC3130	Arm926EJ-S	Nuttx	Ethernet, SD Card, UART	QEMU	versatilepb
UART Server	LPC1700	Cortex-M3	Zephyr	UART Server	xPack QEMU Arm	STM32F429
TCP Echo Server	PIC32MZ EF	MIPS32 M5150	Nuttx	UART, Ethernet	QEMU for MIPS	pic32mz-meb2

Template Board is the template board to which the firmware is ported. We can see that all STM32-based firmware samples are ported to `STM32F429` development board, while all Arm7-based firmware samples are ported to `versatilepb` board.

Emulation of peripheral functions is dependent upon the emulator and the development board. In the experiment, UART is supported on every emulator and every development board, while Ethernet and GPIO are supported on a few development boards, and RF and Zigbee are not supported. Therefore, we can reuse the existing peripheral functions for communication, which also means we no longer study the security issues among the replaced drivers. In our experiments, ADC, RF, and Zigbee are implemented by reusing the UART, and the GPIO of *RTU*, *PLC*, and *Car* are also implemented by reusing the UART.

In order to prove that FirmPorter can adapt the firmware to the hardware correctly, we test each peripheral functionality separately. For example, to test the Web Server sample, we put different web pages on the fake SD Card and visited every HTML file in the browser correctly. Besides, we log onto the `nsh` shell from the serial terminal and list the running tasks and execute commands successfully. Therefore, we consider

Table 6: Result of Firmware Re-hosting

Firmware	Ether.	Disk	UART	ADC	RF	GPIO	Zigbee	SD
RTU	✓	✓	✓	✓	-	✓	-	✓
WSN-sample	-	-	✓	✓	✓	-	-	-
WSN-store	-	-	✓	-	✓	-	-	✓
WSN-upload	✓	-	✓	-	✓	-	-	-
Music Player	-	-	✓	-	-	✓	-	-
PLC	✓	✓	✓	✓	-	✓	-	✓
NAS	✓	✓	✓	-	-	-	-	-
Car	-	-	✓	-	-	✓	✓	-
Web Server	✓	-	✓	-	-	-	-	✓
UART Server	-	-	✓	-	-	-	-	-
TCP Echo Server	✓	-	✓	-	-	-	-	-

the ”peripherals” behaving correctly after porting. We can see from Table 6 that all the firmware are successfully re-hosted.

The *RTU* is the most complicated firmware among the test cases, which has 19221 functions. It provides hundreds of protocol conversion functionalities, and we cannot test all of them. As the primary goal is to test URGENT/11 vulnerabilities on the emulated firmware, we define *success* as *correct execution from bootloader to the start of application*. In the experiment, we can log into the system through the default telnet service and visit the web pages located on the virtual disk. Moreover, the *RTU* application starts correctly and the ADC driver behaves as expected in the application, which uses the Driver Hacking method.

FirmPorter is capable of porting firmware with similar CPU architectures to a specific template board, using the same drivers. In our experiments, all 5 Arm Cortex-M3/M4 firmware samples were ported to the STM32F429 Development Board, which belongs to the Arm Cortex-M4 architecture. In fact, the difference between Arm Cortex-M3/M4 is little (e.g. Cortex-M4 has more features such as floating point calculations than Cortex M3, and the memory layout is slightly different). Since the RTOS follows a standardized and structured development approach, access to specific registers in the application layer is encapsulated within driver functions, so the driver replacement will not have any impact on the application layer. In addition, the VxWorks firmware requires a bootloader, and we can boot any VxWorks-based firmware running on this development board with an all-in-one bootloader only.

5.4 Fidelity & Ability of Vulnerability Assessment (RQ3)

In this experiment, we demonstrate that FirmPorter provides enough re-hosting fidelity to allow PoCs/exploits developed under emulation to be deployed on real devices. Then we examine whether the fidelity is sufficient for verifying vulnerabilities using the firmware ported by the FirmPorter.

Vulnerability Exploitation *Vuln* shows that the vulnerabilities (Buffer Overflow, BOF) are built into each firmware except for the real-world ones. We do not set the NX bit (no execute) which enables attackers to execute codes on the stack or heap. The goal

Table 7: Exploitation Reproducibility

Firmware	Vuln	Vuln func	Reproducible
RTU	Urgent/11	iptcp_usr_get_from_recv_queue()	✓
WSN-sample	BOF	rd_tmp_entry()	✓
WSN-store	BOF	nrf24l01_recv()	✓
WSN-upload	BOF	net_upload_data()	✓
Music Player	BOF	header_parse()	✓
PLC	RCE	***	✓
NAS	Urgent/11	iptcp_usr_get_from_recv_queue()	✓
Car	BOF	radio_recv()	✓
Web Server	BOF	simple_xml_parse()	✓
UART Server	BOF	data_read()	✓
TCP Echo Server	BOF	data_read()	✓

of exploiting is to output the string `exploited!` through UART by exploiting the vulnerability. For the two VxWorks-based systems, we tested CVE-2019-12255 in URGENT/11, which is a heap overflow.

Reproducibility shows that whether the PoCs/exploits work on the physical device. After successfully triggering each vulnerability, we send exploits to physical hardware through real peripherals. As shown in Table 7, the result indicates that the PoCs/exploits also trigger vulnerabilities on the devices correctly. In addition, we send the POCs to the two VxWorks-based firmwares, and we were able to successfully corrupt the telnet service.

We emulated real-world firmware, Modicon M241 PLC, and verified the Stuxnet-like attack proposed by AirBus[12]. We insert a malicious snipplet that invoke the socket system call and connect to a specified IP/port to send the `exploited!` messages into a PLC program and download it to the PLC. The development work of the shellcode is completed in the emulation environment. After downloading the program to the real PLC, the target PC successfully receives the `exploited!` message.

Fidelity Assessment FirmPorter was designed to establish an environment for dynamic firmware analysis, and we are specifically concerned with whether vulnerabilities are triggered in the same manner as they are on the real device. Therefore, we evaluate fidelity from two perspectives: (1). Similarity of execution paths in vulnerable functions (2). Similarity between memory regions after triggering the vulnerabilities. In order to make data collection more convenient, we testify 5 firmware running on the development board.

Table 8: Memory Fidelity in Vulnerable Function

Firmware	Vuln Func	SHE	SHD
Music Player	header_parse()	ece4288a	ece4288a
Car	radio_recv()	1a35a8e5	1a35a8e5
Web Server	simple_xml_parse()	e7fafbc1	e7fafbc1
UART Server	data_read()	acca0cd3	acca0cd3
TCP Echo Server	data_read()	213f4a53	213f4a53

* **SHE** means Stack Hash in Emulator, **SHD** means Stack Hash in Device

Execution and memory fidelity We pay more attention to the part that requires vulnerability analysis, that is, whether the code separated from the hardware abstraction layer, such as the protocol stack, application, or kernel, behaves the same as that running on the real device. In this case, we can develop exploits or patches on the re-hosting platform, which can be directly applied to the real device.

We propose a scheme for calculating fidelity on-demand. In the case of triggering vulnerabilities/attacks, the fidelity is compared by specifying the functions that will be analyzed. However, It is difficult to collect execution data from real devices (such as a PLC) through JTAG. For validation, we use the ETM module (Embedded Trace Macrocell) module on the STM32F4 Discovery, similar to Laelap, while using QEMU's /texttt command (-d exec, nochain) to get execution paths. In addition, we set a breakpoint after the vulnerability is triggered. When the `PC` pointer is executed there, the data in registers and the task stack will be dumped.

The samples we tested are RT-Thread firmware with stack-overflow vulnerabilities. To avoid execution uncertainty caused by hardware interrupts and context switching, we manually disable the interrupt before the function starts and enable the interrupt after the execution is over. The recording of execution starts after the interrupt is disabled, and the entire process of exploiting the vulnerability is recorded. We extracted contents from the task stack after the exploitation was over.

Execution fidelity The execution paths in the device and the emulator are completely overlapped.

Memory fidelity We calculate the SHA-1 hash of contents in the task stack after the attack and find them the same (The value listed in Table 8 remains the last 8 digits only). During the test, we find that a few stack addresses are different. This is because the newly introduced driver uses a different stack size than the original firmware. However, it is meaningless to keep the starting address of the stack the same, especially when it comes to verification and analysis of heap vulnerabilities.

6 Limitations & Discussions

6.1 RTOS without Device Driver Infrastructure

We observe that some RTOSes, such as FreeRTOS, lack an obvious device driver infrastructure and follow a process-oriented design approach. In this scenario, the identification of driver functions via the device driver infrastructure becomes ineffective, necessitating manual localization by analysts. Nevertheless, firmware emulation can still be achieved by replacing driver functions with proper ones. The challenge remains in the full identification of functions belonging to one driver.

6.2 Identifying Functions in Stripped Binaries

The lightweight matching & deducting algorithm works well because we restrict the potential matching results to a small range, thus reducing the collision or mismatching results. Although the matching algorithm is somewhat naive where it may fail in more complicated situations. In actual scenarios, we can further add constraints or just compare the intermediate representation (IR) of the function. The advantage of this method

is that it utilizes the contextual information of skeleton functions for comparing without re-implementing a heavy-weight binary similarity matching tools, which is not the core of this work.

7 Related Work

Hardware-in-the-loop These works forward peripheral operations to a real device when executing hardware-related code. Avatar[40] proposed a framework that forwards MMIO operations to a device through the JTAG channel, and it integrates it with S^2E[8] to conduct concolic execution on firmware. The following work Avatar2[30] optimizes the original framework and integrates with PANDA[13] so that it can replay recorded I/O operations without hardware. SURROGATES[26] enables near real-time dynamic analyses of firmware by providing a low-latency FPGA bridge between the host's PCI Express bus and the system. PROSPECT[24] forwards peripheral hardware accessed from the original host system into a virtual machine. [23] optimize the performance of forwarding by utilizing a cache for peripheral devices and communication to approximate firmware states but suffers from state explosion. Charm[35] addresses the emulation problem of smartphone drivers using the forwarding approach. muAFL[27] leverages ARM ETM hardware debugging feature to collect code coverage information, which transparently collects the instruction trace and streams the results to the PC.
Functions Re-implementing The common idea of these works is to replace the problematic functions with correct or equivalent ones. [37][38] and the following work[2] boots the IOS kernel successfully by manually patching the functions that are incorrectly executed by QEMU. PowerFL[17] manually and automatically identify problematic code and stub it out with function hooks. Firmadyne[7] provides an instrumented kernel to run Linux applications in firmware and researchers can analyze firmware dynamically. Costin et al.[10] replace the kernel in firmware with a stock kernel and emulate the whole userland of the firmware using QEMU to perform extensive web-interface testing. HALucinator replaces functions in Hardware Abstraction Layer with equal ones, then Clements et al.[9] extend HALucinator to work with real-world systems that use the popular VxWorks RTOS. Partemu[19] addressed the problem of emulation of firmware with TrustZone, and it replaced selected components in TZOSes with a model or stub that sufficiently mimics the original components to the target. ECMO[21] uses a technique called peripheral transplantation is to transplant the device drivers of designated peripherals into the Linux kernel binary.
Emulation with Symbolic Execution These works address the problems of firmware re-hosting by applying a symbolic execution technique. FIE[11] performs symbolic execution of (MSP430 16-bit) source code and relies on KLEE[5]. Symdrive[32] aims to discover vulnerabilities in drivers and makes device input to the driver symbolic thereby allowing execution on the complete range of device inputs. Laelaps[6] infers the expected behavior of firmware via symbolic-execution-assisted peripheral emulation and generates proper inputs to steer concrete execution on the fly. Jetset[22] uses symbolic execution to infer what behavior firmware expects from a target device and generate devices models for hardware peripherals in C. μEmu[41] emulates firmware with unknown peripherals by learning how to correctly emulate firmware execution at

individual peripheral access points. Greenhouse[36] employs user-space single-service rehosting techniques to overcome common roadblocks in emulating firmware binaries.
Model-based Emulation These works emulate firmware by modeling the MMIO operations of peripherals directly. Pretender[18] records the original device's MMIO operation then models them so that it can provide the right value to firmware when executing hardware-related code that needs a response from peripherals. P^2IM[16] abstract diverse peripherals and handles firmware I/O on the fly basing on automatically generated models. AIM[15] provides a novel interrupt modeling mechanism that help fuzzer efficiently discover interrupt-dependent code. DICE[29] models DMA channels and manipulates the input transferred through DMA on behalf of the firmware analyzer. Fuzzware[33] is a novel tool to fuzz test unmodified monolithic firmware in a scalable and efficient way. It can automatically generate models that help focusing the fuzzing process on mutating the inputs that matter. Hoedur[34] reorganizes flat, sequential, and opaque firmware fuzzing inputs, which are dynamically generated as peripheral values, into multiple strictly typed and cohesive streams. Firmguide[28] combines the peripheral abstractions in the Linux kernel and kernel-peripheral interactions to semi-automatically generate peripheral models that are then used to synthesize new QEMU virtual machines.

8 Conclusion

In this work, we propose a novel technique called static binary-level driver porting to solve firmware re-hosting problems. We implement this technique in a tool, FirmPorter. It can identify drivers in the firmware, automatically generate proper drivers, and modify the firmware to run correctly in the emulator. We prove that our method has sufficient fidelity for vulnerability assessment, that is, the PoCs or exploits developed in the emulated environment can be directly applied to real devices. We evaluated 11 firmware with 4 types of vulnerabilities, re-hosted in emulators without modification. The result shows that our tool can re-host the firmware correctly, and the PoCs or exploits can be executed correctly both on emulators and real devices.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments and suggestions. This project is supported by Beijing Natural Science Foundation (Grant No. L234033).

References

1. Pwntools github repository (2024), `https://github.com/Gallopsled/pwntools`
2. Afek, J.: Booting the iOS Kernel to an Interactive Bash Shell on QEMU (2019), `https://www.blackhat.com/eu-19/briefings/schedule/#booting-the-ios-kernel-to-an-interactive-bash-shell-on-qemu-17498`
3. Basnight, Z., Butts, J., Lopez Jr, J., Dube, T.: Firmware modification attacks on programmable logic controllers. In: International Journal of Critical Infrastructure Protection (2013)

4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference (2005)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: USENIX Symposium on Operating Systems Design and Implementation (2008)
6. Cao, C., Guan, L., Ming, J., Liu, P.: Device-agnostic Firmware Execution is Possible A Concolic Execution Approach for Peripheral Emulation. In: Annual Computer Security Applications Conference (2020)
7. Chen, D.D., Egele, M., Woo, M., Brumley, D.: Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In: Network and Distributed System Security Symposium (2016)
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In: Architectural support for programming languages and operating systems (2011)
9. Clements, A.A., Carpenter, L., Moeglein, W., Wright, C.M.: Is your firmware real or re-hosted?. In: NDSS 2021 Binary Analysis Research (BAR) Workshop (2021)
10. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (2016)
11. Davidson, D., Moench, B., Jha, S., Ristenpart, T.: FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution. In: USENIX Security Symposium (2013)
12. Dola, F.: Applying a Stuxnet Type Attack to a Modicon PLC (2020), https://airbus-cyber-security.com/applying-a-stuxnet-type-attack-to-a-modicon-plc
13. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable Reverse Engineering with PANDA. In: Program Protection and Reverse Engineering Workshop (2015)
14. Fasano, A., Ballo, T., Muench, M., Leek, T., Bulekov, A., Dolan-Gavitt, B., Egele, M., Francillon, A., Lu, L., Gregory, N., et al.: Sok: Enabling security analyses of embedded systems via rehosting. In: Proceedings of the 2021 ACM Asia conference on computer and communications security (2021)
15. Feng, B., Luo, M., Liu, C., Lu, L., Kirda, E.: Aim: Automatic interrupt modeling for dynamic firmware analysis. IEEE Transactions on Dependable and Secure Computing (2023)
16. Feng, B., Mera, A., Lu, L.: P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In: USENIX Security Symposium (2020)
17. Goodman, P., Dinaburg, A., Brunson, T.: PowerFL:Fuzzing VxWorks embedded systems (2019), https://www.petergoodman.me/docs/qpss-2019-slides.pdf
18. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., Vigna, G.: Toward the Analysis of Embedded Firmware through Automated Re-hosting. In: International Symposium on Research in Attacks, Intrusions and Defenses (2019)
19. Harrison, L., Vijayakumar, H., Padhye, R., Sen, K., Grace, M.: PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In: USENIX Security Symposium (2020)
20. Hex-Rays: IDAPython, https://www.hex-rays.com/products/ida/support/idapython_docs/
21. Jiang, M., Ma, L., Zhou, Y., Liu, Q., Zhang, C., Wang, Z., Luo, X., Wu, L., Ren, K.: Ecmo: Peripheral transplantation to rehost embedded linux kernels. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (2021)
22. Johnson, E., Bland, M., Zhu, Y., Mason, J., Checkoway, S., Savage, S., Levchenko, K.: Jetset: Targeted firmware rehosting for embedded systems. In: 30th USENIX Security Symposium (USENIX Security 21) (2021)

23. Kammerstetter, M., Burian, D., Kastner, W.: Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In: International Conference on Emerging Security Information, Systems and Technologies (SECUWARE) (2016)
24. Kammerstetter, M., Platzer, C., Platzer, C.: Prospect: peripheral proxying supported embedded code testing. In: ACM symposium on Information, computer and communications security (2014)
25. Koret, J.: Diaphora, `https://github.com/joxeankoret/diaphora`
26. Koscher, K., Kohno, T., Molnar, D.: SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In: USENIX Workshop on Offensive Technologies (2015)
27. Li, W., Shi, J., Li, F., Lin, J., Wang, W., Guan, L.: μafl: non-intrusive feedback-driven fuzzing for microcontroller firmware. In: Proceedings of the 44th International Conference on Software Engineering (2022)
28. Liu, Q., Zhang, C., Ma, L., Jiang, M., Zhou, Y., Wu, L., Shen, W., Luo, X., Liu, Y., Ren, K.: Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2021)
29. Mera, A., Feng, B., Lu, L., Kirda, E.: Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In: 2021 IEEE Symposium on Security and Privacy (SP) (2021)
30. Muench, M., Nisi, D., Francillon, A., Balzarotti, D.: Avatar2: A Multi-Target Orchestration Platform. In: Binary Analysis Research (2018)
31. Quarkslab: LIEF, `https://github.com/lief-project/LIEF`
32. Renzelmann, M.J., Kadav, A., Swift, M.M.: SymDrive: testing drivers without devices. In: USENIX Symposium on Operating Systems Design and Implementation (2012)
33. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22) (2022)
34. Scharnowski, T., Wörner, S., Buchmann, F., Bars, N., Schloegel, M., Holz, T.: Hoedur: embedded firmware fuzzing using multi-stream inputs. In: USENIX Security Symposium (USENIX Sec) (2023)
35. Talebi, S.M.S., Zhang, H.T., Hang, Zhang, Z., Sani, A.A., Qian, Z.: Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In: USENIX Security Symposium (2018)
36. Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al.: Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5791–5808 (2023)
37. Wei, Z.: Almost booting an iOS kernel in QEMU (2018), `https://worthdoingbadly.com/xnuqemu/`
38. Wei, Z.: Tutorial - emulate an iOS kernel in QEMU up to launchd and userspace (2018), `https://worthdoingbadly.com/xnuqemu2/`
39. Wright, C., Moeglein, W.A., Bagchi, S., Kulkarni, M., Clements, A.A.: Challenges in firmware re-hosting, emulation, and analysis. ACM Computing Surveys (CSUR) (2021)
40. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In: Network and Distributed System Security Symposium (2014)
41. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th USENIX Security Symposium (USENIX Security 21) (2021)