Work-in-Progress: Measuring Security Protection in Real-time Embedded Firmware

Yuhao Wu, Yujie Wang, Shixuan Zhai, Zihan Li, Ao Li, Jinwen Wang, Ning Zhang
Washington University in St. Louis, MO, USA

***Abstract*—The proliferation of real-time cyber-physical systems (CPS) is making profound changes to our daily life. Many real-time CPSs are security and safety-critical because of their continuous interactions with the physical world. While the general perception is that the security protection mechanism deployment is often absent in real-time embedded systems, there is no existing empirical study that measures the adoption of these mechanisms in the ecosystem. To bridge this gap, we conduct a measurement study for real-time embedded firmware from both a security perspective and a real-time perspective. To begin with, we collected more than 16 terabytes of embedded firmware and sampled 1,000 of them for the study. Then, we analyzed the adoption of security protection mechanisms and their potential impacts on the timeliness of real-time embedded systems. Besides, we measured the scheduling algorithms supported by real-time embedded systems since they are also security-critical.**

I. Introduction

System security and real-time scheduling are among the most active areas of study in embedded systems for the past decade. Through the arms race between the attacker and defender, various security protection techniques are now increasingly deployed in production systems, such as stack canaries, control-flow integrity (CFI), and trusted execution environment (TEE) [1]. Meanwhile, scheduling policies are critical to ensure the real-time and security properties of real-time embedded systems. It has been demonstrated that denial-of-service (DoS) attacks can be launched on the network stack of a router if there is no scheduling deployed on the device via priority inversion [2], [3]. However, while existing studies focus on the discovery of new vulnerabilities [4], or incorporating security into real-time scheduling [5], little attention has been given to understanding the existing deployment of security protection and real-time mechanisms. Inspired by the recent empirical survey-based study on industry practice and real-world firmware [6], [7], we conducted a measurement study to understand the adoption of security protection and real-time mechanisms in real-time embedded systems. We aim at answering three research questions by the measurement:

Q1. *Are the modern software security protection techniques adopted by real-time embedded systems?*

Q2. *What are the widely used scheduling algorithms in real-time embedded systems?*

Q3. *What are the possible timeliness impacts of security protection mechanisms in real-time embedded systems?*

To answer the three questions, we follow three steps as shown in Fig. 1, firmware collection and pre-processing, security and real-time measurement, and timeliness impact analysis. First, to gain a realistic snapshot of the ecosystem of embedded systems, we collected 16.9 terabytes (TB) of firmware images from more than 200 vendors. Second, to gain insights into the adoption rate of security protection and real-time mechanisms, we develop a set of static analysis tools to determine if a particular mechanism is applied in a firmware image. We found that many real-time embedded firmware images lack security protection as expected. Lastly, to understand the real-time impacts of the security protection mechanisms, we analyze the timing overhead they introduced.

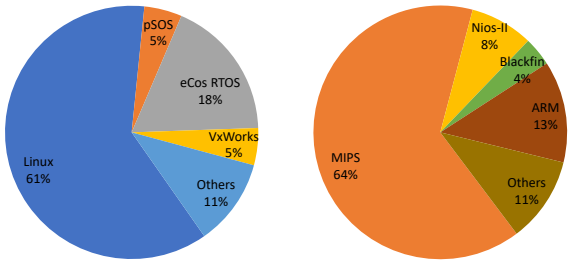
Fig. 1. Overview of the measurement pipeline.

II. Firmware Collection and Pre-processing

Comprehensive analysis of firmware relies on a huge number of firmware images. To conduct such large-scale firmware analysis, a web crawler is developed to collect firmware images from multiple sources and used to profile the collected firmware images to obtain the metadata. A series of scripts are also built to process the unpacking and disassembling of those firmware images for batch analysis.

A. Firmware Collection

To collect publicly released firmware from mainstream vendors, a web crawler is developed to collect firmware images that are in the fields of smart home devices, networking devices, printers, network cameras, etc. Specifically, the web crawler can collect firmware images from the official websites of vendors, open FTP sites, and open-source repositories. Before downloading firmware image files, the web crawler can filter some non-firmware files based on file extensions. Apart from downloading firmware images, the web crawler can also parse and save the metadata of firmware images, including vendor name, product model, device category, download link, firmware version, and release date. To improve the collection efficiency, the web crawler also supports parallel execution. Note that the guidelines for crawling [8] are strictly followed to ensure all operations are legal and meet the ethics criteria.



(a) OS type (b) Architecture type

Fig. 2. The firmware distribution in terms of OS type and architecture type.

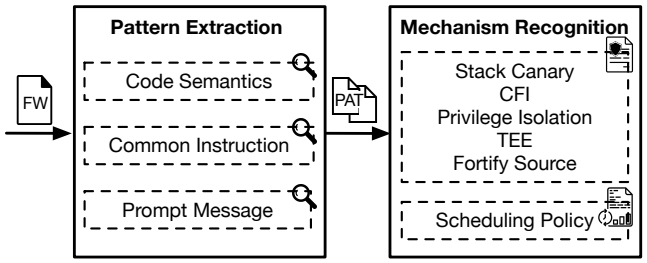


Fig. 3. Overview of the security and real-time mechanism measurement.

B. Firmware Unpacking and Disassembling

To analyze a firmware image, the first step is to unpack the firmware image, if possible. Then, the kernel, application binaries, or binary large object (blob) files are disassembled to obtain the instruction listing of the firmware.

Firmware Unpacking is a process of extracting the kernel, application binaries, bootloader and file system (if any) from the packed firmware. Binwalk [9], an off-the-shelf firmware analysis tool, can unpack firmware images and reveal their types of architecture, operating system (OS), and file system. Therefore, the unpacked files of the firmware image and its corresponding metadata can be obtained through Binwalk. The collected firmware images are categorized into three types: general-purpose OS-based firmware, embedded OS-based firmware, and bare-metal firmware. Note that bare-metal firmware images and some of the embedded OS-based firmware images are single blob files that can not be unpacked, but can be directly disassembled.

Firmware Disassembling is a process of extracting the instruction listing of a firmware image, as well as identifying functions and configurations from the disassembled code. There are two key pieces of information needed for disassembling: architecture (processor type, endianness, and instruction size) and base address. For architecture information of a firmware image, it can be retrieved by parsing the firmware header. If there is no useful information in the firmware header, Binwalk can be used to identify the architecture by trying to disassemble the firmware image using different architectures and output the architecture type with better results.

In the previous work [10], the authors find that the points-to relations of the absolute pointers are useful for base address resolution. For example, an absolute function pointer should point to a function entry and a string pointer should point to a string. To resolve the base addresses of binaries, a number of common base addresses from the reverse engineering community are used to build a base address candidate pool. Then, the base address of a binary can be resolved in three steps: First, disassemble the binary with the base address of 0x0 and search the addresses of all absolute pointers and gadgets, including the addresses of function entries and strings; Second, try to disassemble the firmware with all addresses in the base address candidate pool and select the address that can satisfy the most number of the point-to constraints of absolute pointers. If most constraints can be satisfied with the selected base address, the address will be finally selected; Third, if there is no appropriate base address in the base address candidate pool, the base address with the range of 0x0 to the minimum address that is found in the absolute pointers will be attempted, and the one that satisfies the most number of constraints is selected.

C. Firmware Dataset Statistics

With the developed web crawler, 16.9 TB of firmware images are collected in total. Specifically, there are 6.4 TB of firmware images collected from the official websites of vendors, and 10.5 TB of firmware obtained from open FTP sites and open-source repositories. At the time of writing, 157,141 firmware images (about 6 TB) from 204 vendors have been pre-processed. The corresponding products of these firmware images are commonly used in consumer markets, such as networking devices, cameras, and smart home devices. The pre-processing for other firmware images is still running since these procedures requires a large amount of computation.

Firmware Profiling is performed to inspect the metadata of the pre-processed firmware images. From the 157,141 firmware images, 111,958 firmware images can be unpacked or disassembled[1]. The remaining files can be either encrypted firmware images or even non-firmware files. The distribution of architecture type and OS type of these firmware images is shown in Fig. 2. Overall, in terms of architecture, MIPS (64.38%) and ARM (12.96%, including Cortex-A, Cortex-M, and Cortex-R) have the highest proportion in our dataset. Although the ARM architecture has been better developed, there are still many devices that use the MIPS architecture, especially older devices. For OS, Linux (61.33%) takes the majority, while a series of real-time OS (e.g., eCos RTOS, VxWorks, pSOS) are also widely used. Note that there are also some firmware images, which can not be profiled by Binwalk.

Firmware Sampling is conducted to construct a representative measurement dataset with diverse firmware. To ensure representativeness, the firmware images are selected from mainstream vendors. Meanwhile, the firmware images with various architecture types and OS types are selected to ensure diversity. Finally, 1,000 firmware images are sampled from the pre-processed firmware images to form the dataset.

[1]The collection of original firmware images is available to the community at https://github.com/WUSTL-CSPL/Firmware-Dataset.

TABLE I
RESULTS OF THE PRELIMINARY MEASUREMENT

Security Protection Mechanism				
Stack Canaries	CFI	Privilege Isolation	TEE	Fortify Source
0.425	0	0.024	0	0.15
Real-time Mechanism				
Priority-based	Time-sliced	Deadline-based	Partial Preemptive	
0.998	0.979	0.500	0.553	

III. SECURITY AND REAL-TIME MECHANISM ANALYSIS

In this section, the security protection and real-time mechanisms of firmware are measured (see overview in Fig. 3). Table I summarizes the preliminary measurement results.

A. Security Mechanism Measurement

With the emergence of various attacks on embedded systems, many security mechanisms are proposed as a relief [11]. As there are kinds of security protection mechanisms, the target security protection mechanisms for measurement are selected based on two criteria: (i) The target attacks of these security protection mechanisms are prevalent in embedded systems; (ii) The security protection mechanisms can be deployed on embedded devices without non-trivial efforts. Therefore, five security protection mechanisms are selected. In the following, the brief introduction, identification approach, and timeliness impact of these mechanisms are described.

Stack Canary is a defense mechanism against stack overflow by placing a secret random canary value at the entry point of the function and checking it before the function returns. Stack overflow is detected if the canary value is modified. Concretely, the message *"*** stack smashing detected ***"* is printed by the checking function *__stac_chk_fail()* in stack canaries [6]. To identify whether stack canaries are deployed, the above-mentioned message string is searched in function definitions, which prompts the location of the function *__stack_chk_fail()* and its references. If this function is invoked, stack canaries are deployed in the firmware. Our initial measurement result indicates nearly half (42.5%) of the firmware images in the dataset have deployed stack canary.

Real-time Impact: Stack canary generates a non-negligible runtime overhead when function invocations are frequent as the canary value is checked at each function return. Additionally, loading canary values can pollute the data cache, thus increasing the cache miss rate.

CFI is a defense against control-flow hijacking attacks, which disarrays the control flow of a program's execution. Integrated into both GCC and Clang compilers, CFI checks the target of every indirect jump according to a pre-computed control-flow graph. To identify whether CFI is deployed in a firmware image, checkers must be inserted before all indirect call sites in the corresponding disassembled code. If such checkers are detected, CFI is deployed in the device. Similarly, this method can be extended to other protection mechanisms such as DFI and Sandboxing. From the preliminary measurement results, although CFI has been integrated into existing compiler toolchains, it still lacks deployment in the real world since none of the firmware images in the dataset has deployed this mechanism.

Real-time Impact: CFI introduces high runtime overhead when an application has a large number of indirect jump instructions. Not only does checking each indirect jump target cost additional time, but cache and branch prediction history pollution during target checking can also affect application performance.

Privilege Isolation aims at enforcing the principle of least privilege. For example, a user space program can only access kernel resources via specific system calls, a set of privilege-switching instructions. For example, *SVC* and *syscall* instructions are adopted in ARM and MIPS architecture respectively to invoke system calls. The existence of privilege-switching instructions in the disassembled code indicates that the privilege isolation scheme is deployed in the firmware image. The preliminary result demonstrates that only a small portion (2.4%) of the firmware images in the dataset separate kernel and user space. Most of them execute real-time tasks in kernel mode. Hence, any vulnerability in a real-time task can lead to the whole system being compromised.

Real-time Impact: The runtime overhead of privilege isolation generated by privilege isolation is high because of frequent context switches when switching privileges.

TEE provides a hardware-level isolation environment for secure computing. Specifically, in the devices protected by TEE, the CPU can execute instructions either inside the TEE or a rich execution environment (REE). ARM TrustZone is the most common TEE in embedded devices while there is no TEE support on MIPS or PowerPC architecture. If TrustZone is enabled, a particular instruction *SMC* allows the CPU to switch between the Secure world (TEE) and Normal world (REE). To identify the adoption of TrustZone, the existence of *SMC* instructions in the disassembled code is checked. Although TEE provides a strong hardware-level separation, the preliminary measurement shows TEE is not deployed in any of the firmware images in the dataset.

Real-time Impact: Similar to privilege isolation, TEE has high runtime overhead since frequent execution environment switching results in frequent context switching.

Fortify Source is a mechanism that replaces the unsafe functions, such as *memcpy* and *strcpy*, with their safe variants at compile time. For example, *strcpy* is replaced by *__strcpy_chk*, a secure variant. Similar to stack canaries, a message *"*** buffer overflow detected ***"* is printed by the replacement function if any violation is detected [6]. A similar method is adopted to detect fortify source as that of stack canaries. The preliminary measurement shows that a portion (15%) of firmware images in the dataset have deployed fortify source.

Real-time Impact: Fortify source has a predicted low runtime overhead, as it only comes from additional input checking for the use of safe functions.

B. Real-time Mechanism Measurement

Firmware usually incorporates many real-time scheduling strategies to satisfy its real-time constraints. The adoption of different real-time strategies and design choices in firmware images are presented as follows.

Real-time Scheduling is of high importance in meeting real-time requirements. The adopted real-time scheduling algorithms can be identified based on the detection of specific functions. For example, VxWorks supports two scheduling algorithms, each with a distinct scheduler [12]. In the preemptive scheduler, high-priority tasks are handled first, while the tasks with identical priority are handled in the order of arrival. Round-robin (RR) scheduler handles high-priority tasks first, yet tasks with the same priority are handled in a round-robin manner with the same execution time slice. The invocation of the *kernelTimeSlice()* function indicates the activation of the RR scheduling algorithm.

A two-step approach is proposed to identify which real-time scheduling algorithms are in use: First, recognize all scheduler activation functions in the firmware image; Second, identify the invocation of any specific scheduler activation functions. To recognize the activation functions, the complete definitions of the activation functions from numerous firmware samples accompanied with a symbol table are extracted. Inspired by FirmUp [13], the activation functions are then decomposed into canonical representations as the corpus for function matching. To detect the activation functions in an arbitrary firmware image, its functions are converted into canonical representations, which are used to perform a similarity matching with the constructed corpus (see details in [13]). To identify the function invocation, instructions using the activation function address as an operand can be searched in the disassembled code of the firmware image. Upon the detection of a specific activation function and its invocation, the corresponding real-time scheduling algorithm in use can be confirmed. Note that due to the challenge of dynamically emulating the firmware images, the identification of adopted scheduling algorithms in firmware images is solely based on the static analysis.

From the measurement results, for non-Linux-based images, most of their scheduling mechanisms are priority-based (99.8%) scheduling and none of them supports deadline-based scheduling. Besides, all firmware images support full-preemptive scheduling but nearly half of them (55.3%) use partial preemptive scheduling as default.

C. Measurement Result Analysis

From a security perspective, stack canaries are more prevalent than the other four security mechanisms while CFI and TEE are the least widely used security mechanisms. From a real-time perspective, priority-based RR scheduling is the most widely used scheduling algorithm. Though many firmware images are designed for peripherals, few of them deploy scheduling policies for I/O devices. These devices can be vulnerable to DoS attacks [14], [15] due to lack of scheduling policy as we mentioned before.

IV. Conclusion and Future Work

In this paper, we presented our effort to understand the adoption of security protection in real-time embedded firmware. A firmware dataset with more than 16.9 TB individual firmware images as well as a set of static analysis tools is created to facilitate the exploration of the subject. Observing the low adoption rate of these security protection techniques, we also analyze the real-time impacts behind them in an attempt to understand the possible reason for the lack of deployment. We have made the firmware dataset available to benefit the research community.

In the future, we plan to conduct empirical measurements on more firmware samples varying the application domain, architecture, and OS to study security protection and real-time measurement deployment. Moreover, we hope to work with the community to improve our understanding of (i) the timing overheads of security protection in the real world; (ii) the actual attack surfaces (especially pertaining to timing) and other security protection mechanisms (e.g., virtualization) of the real-time embedded firmware; (iii) how the real-time requirements present practical challenges for security deployment in the real world, to shed light on the direction of real-time-aware security protection.

Acknowledgment

We thank the reviewers for their feedback. This work is supported in part by US National Science Foundation under grants CNS-1837519, CNS-1916926, CNS-2038995, CNS-2154930, and CNS-2229427, by Army Research Office under contract W911NF-20-1-0141, and by US Department of Energy under award DE-EE0009338.

References

[1] J. Wang *et al.*, "RT-TEE: Real-time system availability for cyber-physical systems using arm trustzone," in *S&P*, IEEE, 2022.
[2] N. Zhang *et al.*, "Memory forensic challenges under misused architectural features," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 9, pp. 2345–2358, 2018.
[3] A. Li *et al.*, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *RTSS*, IEEE, 2022.
[4] S. Hounsinou *et al.*, "Vulnerability of controller area network to schedule-based attacks," in *RTSS*, IEEE, 2021.
[5] M. Hasan *et al.*, "Exploring opportunistic execution for integrating security into legacy hard real-time systems," in *RTSS*, IEEE, 2016.
[6] R. Yu *et al.*, "Building embedded systems like it's 1996," in *NDSS*, ISOC, 2022.
[7] B. Akesson *et al.*, "An empirical survey-based study into industry practice in real-time systems," in *RTSS*, IEEE, 2020.
[8] M. Thelwall *et al.*, "Web crawling ethics revisited: Cost, privacy, and denial of service," *J. Assoc. Inf. Sci. Technol.*, vol. 57, no. 13, pp. 1771–1779, 2006.
[9] "Binwalk." https://github.com/ReFirmLabs/binwalk.
[10] H. Wen *et al.*, "FirmXRay: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," in *CCS*, ACM, 2020.
[11] L. Szekeres *et al.*, "SoK: Eternal war in memory," in *S&P*, IEEE, 2013.
[12] "Vxworks schedulings." https://sites.google.com/site/alperkultur/home/c-c-1/vxworks-process-scheduling.
[13] Y. David *et al.*, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *ASPLOS*, ACM, 2018.
[14] A. Li *et al.*, "From timing variations to performance degradation: Understanding and mitigating the impact of software execution timing in slam," in *IROS*, IEEE/RSJ, 2022.
[15] A. Li *et al.*, "Chronos: Timing interference as a new attack vector on autonomous cyber-physical systems," in *CCS*, ACM, 2021.