Embedded Device Vulnerability Repair Based on Hot Patches

Zesheng Xi , Bo Zhang * , Yunfan Wang , Chuan He , Tao Zhang

Posted Date: 2 January 2025

doi: 10.20944/preprints202412.2456.v1

Keywords: Hot patching; Embedded devices; Vulnerability localization; Vulnerability remediation; System continuity

Article

Embedded Device Vulnerability Repair Based on Hot Patches

Zesheng Xi [†,‡], Bo Zhang [†,‡*], Yunfan Wang [†,‡], Chuan He [†,‡] and Tao Zhang [†,‡]

State Grid Laboratory of Power Cyber-Security Protection and Monitoring Technology, China Electric Power Research Institute Co., Ltd., Nanjing, China

* Correspondence: zhangbo6@epri.sgcc.com.cn;

† Current address: Nan Rui Road.

‡ These authors contributed equally to this work.

Abstract: Industrial control systems may be affected by vulnerabilities, leading to a series of security issues. However, existing vulnerability remediation methods mainly include static patching and dynamic patching, but due to the hardware resource constraints and complex application environments faced by embedded devices, these traditional methods all suffer from the limitations of high risk of service interruption and high resource overhead, which make it difficult to satisfy the special needs of embedded devices in terms of security and continuity assurance. For this reason, this paper proposes a hot-patch-based vulnerability repair method for embedded devices. The method utilizes hardware built-in functions to perform vulnerability repair during system runtime, effectively reducing system downtime and optimizing resource consumption during the repair process. Specifically, the repair method proposed in this paper contains four steps: vulnerability identification, vulnerability localization, patch preparation and patch switching. By using the onboard debugging unit to replace instructions at runtime, thus avoiding device downtime and system reboot, the system continuity of the embedded device is guaranteed.The experimental results show that this method achieves efficient and secure vulnerability repair while meeting the real-time and resource constraints of embedded devices, providing an innovative and effective way to improve the security and reliability of embedded devices.

Keywords: Hot patching; Embedded devices; Vulnerability localization; Vulnerability remediation; System continuity

1. Introduction

With the wide application of embedded devices in the fields of industrial control, smart home, medical devices, vehicle networking and Internet of Things, their security issues have gradually become the focus of people's attention [1]. During operation, embedded devices often expose various types of security vulnerabilities, which may be exploited by malicious attackers [16], leading to data leakage, interruption of system functionality or even complete paralysis of the device. In mission-critical scenarios, such risks can have significant social and economic impacts [17,18]. Therefore, vulnerability remediation plays an indispensable role in the security assurance of embedded devices, and its goal is to fundamentally eliminate potential threats by timely fixing known vulnerabilities to ensure the stability, reliability and security of devices and systems.

In recent years, academia and industry have conducted a lot of research in the field of vulnerability remediation and proposed a variety of methods to solve the vulnerability problem. Currently, traditional vulnerability remediation methods can be mainly categorized into two types: static patching [2] and dynamic patching [3]. Static patching fixes the problem by replacing the vulnerable module or upgrading the firmware, which usually needs to be done while the device is down, and is characterized by simple implementation and direct fixing. Dynamic patching, on the other hand, allows vulnerability remediation to be performed while the system is running, utilizing technical means to dynamically load the patch code, which avoids the need for total downtime and is more suitable

 doi:10.20944/preprints202412.2456.v1

for system environments that require continuous operation. Although these two types of traditional vulnerability repair methods provide a certain degree of protection for system security, there are still the following limitations in the application of embedded devices:

1. **High risk of service interruption:** The static patching method requires complete downtime to complete the vulnerability repair, and for mission-critical scenarios (e.g., medical monitoring, industrial production, etc.) running in embedded devices, service interruptions may lead to serious consequences, which is not conducive to system continuity assurance.

2. **High resource overhead:** dynamic patching approaches are usually designed for high-performance devices, and embedded devices are difficult to support the operation of complex patches due to resource constraints (e.g., limited memory, storage, and computational power). In addition, dynamic patching may introduce additional computation and storage overheads, further affecting device performance.

To address the above limitations, this paper proposes a hot-patching-based vulnerability remediation method for embedded devices, which is based on the built-in functionality of the hardware to perform hot-patching of embedded systems, retaining strict real-time constraints while minimizing additional resource usage.The method consists of four steps: vulnerability identification; vulnerability localization; patch preparation; and patch switching. The key idea is to replace instructions at runtime using an on-board debugging unit, which is readily available on commercial off-the-shelf processors. The methodology enhances the detection-based hotfixing approach without the need for binary detection and other software layers.

The main contribution of this paper:

1. proposes a customized remediation framework to meet the heterogeneous needs of embedded systems, designs a four-step remediation process (vulnerability identification, vulnerability localization, patch preparation, and patch switching) tailored to the characteristics of embedded devices, and enhances the detection-based hotfixing approach to avoid relying on the binary detection and other software layers, and improves the accuracy and adaptability of the patches.
2. In order to solve the service interruption problem caused by static patching, the framework proposes a hot-patching-based vulnerability remediation method for embedded devices, which utilizes the hardware built-in function to execute hot patches, avoiding device downtime and system restart, and guaranteeing system continuity in mission-critical scenarios.
3. In order to overcome the high overhead of dynamic patching, the framework uses on-board debugging units to dynamically replace instructions at runtime, which greatly reduces resource consumption during the repair process while maintaining the real-time nature of the system.
4. The experimental results show that the hot-patch-based vulnerability repair method proposed in this paper performs well in embedded device vulnerability repair, effectively reduces system downtime and service interruption, and provides strong support for system stability and availability. Through in-depth analysis and optimization, this paper provides an innovative and efficient solution for security maintenance of embedded devices, which has important theoretical significance and practical value.

2. Related Works

With the continuous development of Internet technology, the traditional patching method that requires constant recompilation and deployment no longer meets the rapid iteration needs of enterprises. In order to solve the problems and security vulnerabilities that may occur during the actual use of software products, and to improve the reliability and security of the system through real-time updating and repairing, as well as to reduce the maintenance cost and improve the development efficiency, the academia and the industry have turned their attention to hot patching. Jiang Wen [4] et al. proposed a hot-patching technique for software maintenance in VXWORKS operating system, using ClearCase and ICP-CI to produce hot-patches, introducing the advantages of real-time fixes and improved system reliability. Christian Niesler [1] et al. proposed a framework called HERA to

 doi:10.20944/preprints202412.2456.v1

hot-patch embedded real-time systems without violating real-time constraints. HERA utilizes the hardware features of commercial Cortex-M microcontrollers to perform hot patching without any hardware modifications, preserving hard real-time constraints while keeping additional resource usage to a minimum. Zhou M [5] et al. proposed a reliable real-time patching framework, called RLPatch, which allows for closed-source patching of Programmable Logic Controllers (PLCs) without the need to access the source code. Yi He [6] et al. proposed RapidPatch, which consists of a patch generation module for generating device-specific patches from eBPF patches and configurations, and a patch validation module for checking for malicious behaviors in the patches. Mingquan Wang [7] discusses the remote hot deployment technology based on SpringBoot. The principles of Devtools and Arthas, two Java implementations of hot deployment, are described in detail, including hot deployment methods based on ClassLoader and Java Agent mechanisms. Holmbacka [8] et al. introduce a lightweight runtime update framework that replaces C-based software in embedded systems without interrupting the system. parts of the C-based software in embedded systems without interrupting the system. This framework is based on FreeRTOS and adds mechanisms for dynamically linking and relinking FreeRTOS tasks to the system. Ramaswamy [9] et al. present a new hotfix framework, Katana, which provides an effective hotfix solution for ELF binaries, which is important for improving software security and reliability. Payer [10] et al. introduce a dynamic software update mechanism that a dynamic software update mechanism that dynamically applies security patches without modifying the binary application. This mechanism combines sandboxing techniques to protect system integrity and improves application availability through a dynamic update mechanism. Haibo [11] et al. propose a scheme that provides real-time update capabilities through virtualization, allowing patches and upgrades to be applied at any time without the need to be in a silent state. The advantages of this approach are its wide range of applications and good portability, which makes it particularly suitable for integration into general-purpose virtualized systems.Altekar [12] et al. explored the development and application of the OPUS tool, which is capable of dynamically applying fixes to C programs, especially in safety-critical applications that require frequent updates.

While the above hot-patch based patching approach addresses the hardware architecture constraints of IoT devices and satisfies the embedded device real-time problem, it is difficult to generate patches based on different types of devices due to the fact that vendors have a variety of types of fragmented devices, which hampers the installation of patches on devices that require high real-time and high availability. In view of the above, this paper performs hot patching of embedded systems based on the built-in functionality of the hardware, which preserves the strict real-time constraints while addressing the compatibility issues.

3. Methodology

Hot-patch based vulnerability patching method gives a non-disruptive deployment update method for embedded devices based on ARM processor with built-in debugging function, which mainly contains four steps: (1) vulnerability identification; (2) vulnerability localization; (3) patch preparation; and (4) patch switching.

 doi:10.20944/preprints202412.2456.v1

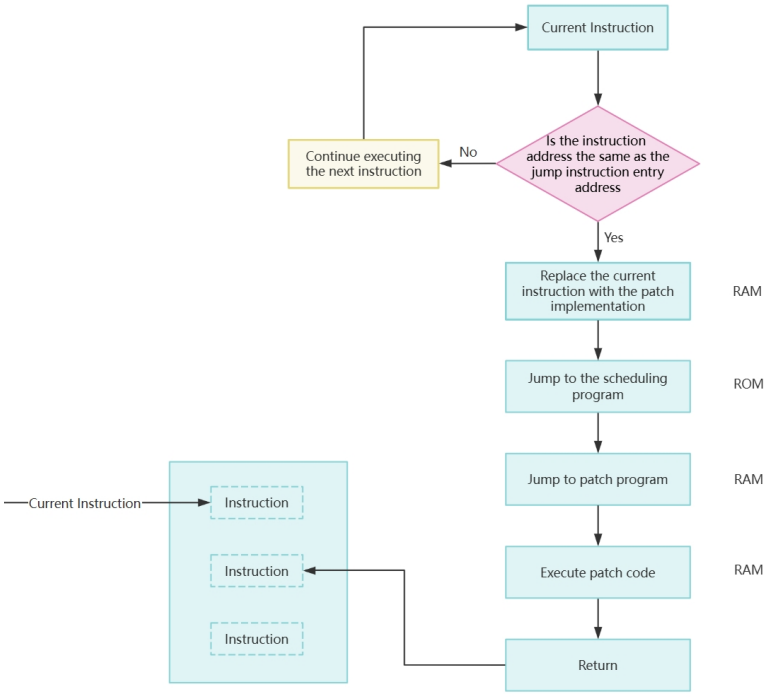

Figure 1. Flowchart of inserting jump instruction through debugging unit

3.1. Vulnerability Identification

Vulnerability identification includes Download the updated report, Extracting Vulnerable Entities, Filtering Vulnerable Entities, Manually Creating Roles, Extracting Logical Instructions, and Obtaining Trigger Conditions.

Start by obtaining the latest update report from the PLC's vendor, which contains information on known vulnerabilities. The main focus is on Major Non-Recoverable Failure (MNRF) vulnerabilities that can lead to memory corruption, such as code-level security vulnerabilities, access control attacks, time-based authentication vulnerabilities, and so on. The presence of these threats may not only lead to data leakage or corruption, but may also cause serious physical damage to industrial control systems and even affect the stable operation of the entire infrastructure.

Named Entity Recognition (NER) techniques are then used to extract vulnerability information from the update report, including vulnerability ID, vulnerability type, affected firmware version, affected controller, logical instructions and trigger conditions [13], which is critical to understanding the scope of the vulnerability.

Next, vulnerability entities are screened. Vulnerability entities that do not meet the requirements can be rejected, such as vulnerabilities that are missing critical information. For vulnerabilities that are missing required entities, roles can be created manually to ensure the integrity of the vulnerability.

The logical instructions are then extracted by consulting the programming manual provided by the PLC manufacturer for information related to the controller's features. In reading the programming manual, special attention should be paid to those sections that describe how the PLC responds to specific inputs, how it processes data, and how it executes control logic. This information helps to extract vulnerable logic instructions.

Finally to obtain trigger conditions, focus on Arabic numerals and programming logic keywords to identify and obtain trigger conditions for vulnerabilities. In vulnerability detection, Arabic numerals may be used as parameters to certain specific functions or blocks of code, or as part of certain conditional judgments. For example, in a loop, if the number of loops is a fixed Arabic number, then the loop may trigger a security vulnerability for some reason, such as an overflow. Programming logic keywords can also lead to security vulnerabilities in some cases when improperly used or

 doi:10.20944/preprints202412.2456.v1

misconfigured. For example, an incorrect loop termination condition could lead to resource exhaustion or denial-of-service attacks.

3.2. Vulnerability Localization

Vulnerability localization consists of inserting vulnerable logic instructions, setting observation points, tracing code access paths, and logging vulnerability information.

Before turning on the PLC, it is first necessary to insert a vulnerable logic instruction into the control program with the unique value tagged controller label. This is done to trigger potential vulnerabilities that can be exploited by an attacker for illegal operations or data tampering. Used to trigger potential vulnerabilities [5] . In addition, complex logic controls can be implemented in PLC programming by using the logic flag value method, which is simple and reliable.

Watchpoints are then set at the vulnerable instruction through the PLC's built-in debugger in order to keep track of which code accessed the dangerous instruction. This step can be accomplished by modifying the source code of the PLC program or by dynamically inserting watchpoints at runtime.

When the program runs to a set observation point, the debugging tool records all code paths that access the instruction. This helps to analyze which parts of the code actually access these dangerous instructions, thus providing a basis for subsequent security hardening.

Finally, the vulnerability information is recorded, and the discovered vulnerability information is recorded in the vulnerability remapping table, including the firmware version number, the vulnerability ID, the logical instructions involved, and the entry address of the vulnerable instructions.

3.3. Patch Preparation

Patch preparation consists of accepting patches from external sources and verifying the integrity of the patch, parsing the patch code and meta-information (such as patch size and patch insertion points) from the patch format, and performing a static check of the patch code to avoid dangerous functions.

First accept patches from external sources and verify the integrity of the patches. The system needs to securely receive external patches and verify their integrity and trustworthiness through checksums or hashes to prevent security risks. After the patch data is received, a checksum calculation is performed or a cryptographic hash function is used to generate a hash value that is compared to the expected value to ensure integrity. A match confirms the integrity of the patch.

The patch code and meta-information (such as patch size and patch insertion point) are then parsed from the patch format. Parsing the patch code is the fundamental step. It involves extracting the actual modifications from the patch file, including lines of code that were added, deleted, or modified. This process often requires an in-depth analysis of the source code to determine how the patch affects existing program structure and logic. The parsing of meta-information is equally important. Patch size is directly related to the amount of storage space required, which is especially important in resource-constrained environments. Correct patch sizing can help system administrators anticipate the impact of a patch deployment and thus make sound resource planning [16] . Additionally, patch insertion point information is critical to ensure that patches are installed correctly. The insertion point usually refers to a specific location in the original code where the patch will be applied. If the insertion point is inaccurate, the patch may not be able to perform its fixes effectively or even introduce new bugs.

Finally the patch code is statically checked. With ICP-CI, the patch code can be statically checked using PC-Lint and Coverity Prevent [4] to ensure syntactical correctness, compliance with coding specifications, consistency of style, and to check for security vulnerabilities in order to avoid introducing new risks and to assess code quality.

3.4. Patch Switching

Patch switching includes copying the patch code to the patch preselection area of the Random Access Memory (RAM) and adding the contents of the meta-information to the corresponding data structure; adding the patch task to the scheduling program and configuring the jump instruction; updating the program to notify the hardware of the activation of the patch through an atomic instruction;

 doi:10.20944/preprints202412.2456.v1

monitoring the instructions executed by the CPU, comparing the current memory location of the CPU with the insertion point of the jump instruction, and triggering an interrupt when they are consistent; jumping to the patch code through the jump instruction; and returning to and executing the patch code when it is finished. Trigger an interrupt; jump to the patch code through the jump instruction; execute the patch code and return to normal execution after completion.

3.4.1. Debugging Unit FPB Configuration

Copy the patch code to the patch preselection area of RAM and add the contents of the meta-information to the corresponding data structure [15], i.e., the debug unit FPB (Flexible Priority Blocker) configuration. The FPB unit configuration requires a C library to be executed, and this library contains the following functions:

- The fpb_Init function is responsible for initializing the FPB unit, ensuring that the FPB unit has been loaded correctly, and preparing for subsequent task scheduling and priority management;
- The fpb_enable function is used to set the FPB unit to be enabled and disabled; if not set, the device will ignore any breakpoints that have been configured and enabled;
- The enable_single_patch function utilizes code remapping and breakpoints to enable the creation of jump instructions for control flow redirection. The jump instructions can be provided directly by the patch or generated by calculating to get the specified offset. The function sets the breakpoint at the insertion location of the jump instruction and sets the breakpoint behavior to code remapping. Atomic switches are implemented to enable hot patching by writing in a single register;
- The load_patch_and_dispatcher function is responsible for patch loading and preparation, and the basic scheduler copies the patch into RAM by loading and modifying it according to the given patch location.

3.4.2. Add Patch Tasks to the Scheduler and Configure Jump Instructions

The introduction of dynamic patching technology requires extensions to handle patch tasks, including loading, execution and subsequent maintenance work. By inserting jump directives, control flow can be transferred to the patched code at runtime, but it needs to be carefully configured to ensure correct execution and avoid security issues.

3.4.3. The Update Program Notifies the Hardware to Activate the Patch through Atomic Instructions

The update program can activate patches efficiently in real time through atomic instructions, which at the hardware level usually involves cache coherence protection and prohibiting instruction reordering to ensure data consistency and correctness during the patching process.

3.4.4. Monitor the Instructions Executed by the CPU

To realize atomic switching, hardware breakpoints can be used to enable patching. Hardware breakpoints are a special debugging feature [14], which can be set to trigger a patch switch. The patch switch is triggered when the CPU executes to the set hardware breakpoint. Since hardware breakpoints are atomic instructions, the atomicity of the patch switch can be guaranteed. The process of patch switching is a sophisticated and complex operation that involves CPU instruction monitoring and interrupt handling. In this process, the CPU plays a crucial role, continuously monitoring all instructions being executed. Whenever the CPU executes an instruction, it compares the current instruction address with a preset insertion point for jump instructions. This comparison process is extremely critical. If the CPU finds an exact match between the current instruction address and the jump instruction insertion point, it immediately triggers an interrupt to pause the current instruction flow. This is an exact breakpoint hit, meaning that a patch switch is required. At this point, the CPU stops executing the current instruction, and by writing pre-configured data to the specified hardware registers, it can realize the insertion of the jump instruction and switch to patch loading and replacement. If it does not match then the execution of the next instruction continues.

 doi:10.20944/preprints202412.2456.v1

3.4.5. Jump to Patch Code by Jump Instruction

As in Figure 1, after a breakpoint hit, the CPU will quickly replace the current instruction with a predefined patch table entry. This patch table entry not only contains the information of the jump instruction that needs to be executed, but also specifies the target address of the jump. the CPU will extract the jump instruction from this patch table entry and execute it, thus realizing a smooth transition of the instruction flow. Inserting the patch code into the target code can be achieved by configuring the insertion point of the jump instruction and the branch instruction. The jump instruction insertion point refers to the location where the patch code needs to be inserted into the target code, and the branch instruction refers to the location where the patch code needs to jump to after its execution is completed. By configuring the jump instruction insertion point and branch instruction, the modification of the target code can be realized. However, due to the characteristics of our chosen platform, it does not support jumping directly from ROM to RAM, which requires us to realize this jump through a special area in ROM, the jump_section. This jump_section is actually a small patch dispatcher that redirects the executed code to a patch in RAM. In this way, we are able to bypass platform limitations and achieve flexible jumping of code.

3.4.6. Execute Patch Code

When the CPU finishes executing the patch code in RAM, it returns to the original instruction stream and continues to execute the next instruction. This switching process is transparent to the user, and they hardly feel any interruption or delay. This is the essence of patch switching, which enables fast replacement and execution of specific instructions without affecting the overall operation of the system.

4. Experiments

4.1. Experimental Environment

Table 1 shows a detailed list of experimental environment configurations.

Table 1. Experimental configuration

Categories	Name	Description
Hardware	Multiprocessor	Intel Core i7-10870H
	GPUs	NVIDIA GeForce RTX 3060
	Random access memory (RAM)	16GB DDR4
Software	Development and operating environments	Ubuntu 20.04 LTS
	Integrated development environment (IDE)	Eclipse IDE 2021-09
	programming language	C, C++
	Debugging Tools	GDB 10.1
	version control	Git 2.25.1
	network tool	Wireshark 3.4.8
	Traffic Tools	Scapy 2.4.5
	virtualized environment	VMware Workstation Pro

4.2. Data Sets

In this study, the vulnerability fixing dataset shown in Table 2 below was used, with each vulnerability corresponding to a specific component and description, as well as a corresponding filter.

 doi:10.20944/preprints202412.2456.v1

Table 2. Data sets

Vulnerability ID	subassemblies	descriptive	(machine) filte
2020-17441	PicoTCP	Validate IPv6 payload length field against actual size for function pico_ipv6_process_in	Filter Patch
2020-17442	PicoTCP	Validate hop-by-hop IPv6 extension header length field for function pico_ipv6_process_hop-by-hop	Filter Patch
2020-17443	PicoTCP	Restrict that echo->transport_len is no less than 8 in pico_icmp6_send_echoreply	Filter Patch
2020-17444	PicoTCP	Check possible overflow of header extension length field for pico_ipv6_check_headers_sequence	Filter Patch
2020-17445	PicoTCP	Validate optlen using a loop prior to function pico_ipv6_process_destopt	Filter Patch

 doi:10.20944/preprints202412.2456.v1

4.3. Experiment and Result Analysis

4.3.1. Vulnerability Fixing Time Comparison Experiment and Result Analysis

In the vulnerability remediation process for embedded devices, remediation time is a key metric. In order to evaluate the advantages of hot-patching methods in terms of fixing time, we conducted comparative experiments covering three cases: no patch added, traditional patching method and hot-patching method. The experimental results are shown in Figure 2.

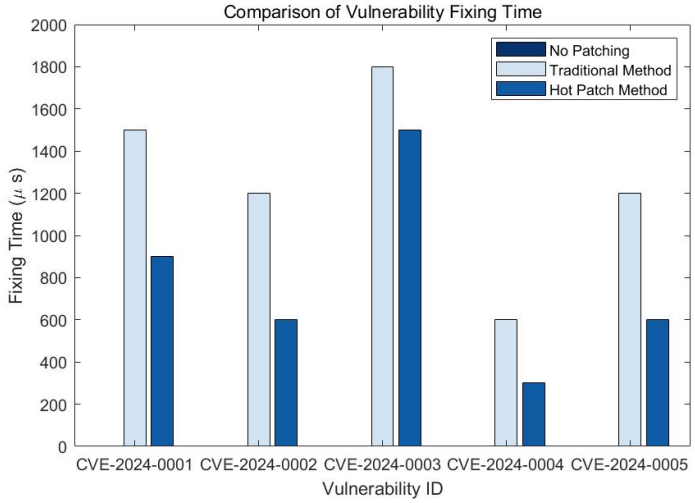

Figure 2. Comparison of vulnerability fixing time

As seen in Figure 2, the hot-patching approach has significantly shorter remediation times than the traditional update approach for all tested vulnerability IDs. This significant difference indicates that the hot-patching approach has a clear advantage in reducing system downtime. Shorter remediation time is critical for improving system availability and minimizing service interruptions. Traditional update methods require long system downtime, which challenges the real-time and reliability of embedded devices, while hot-patching methods effectively solve this problem, allowing the system to remain running efficiently during the vulnerability remediation process.

4.3.2. Vulnerability Repair Success Rate Comparison Experiment and Result Analysis

In order to further evaluate the effectiveness of different restoration methods, we compared the performance of different methods in terms of restoration success rate. The experimental results are shown in Figure 3.

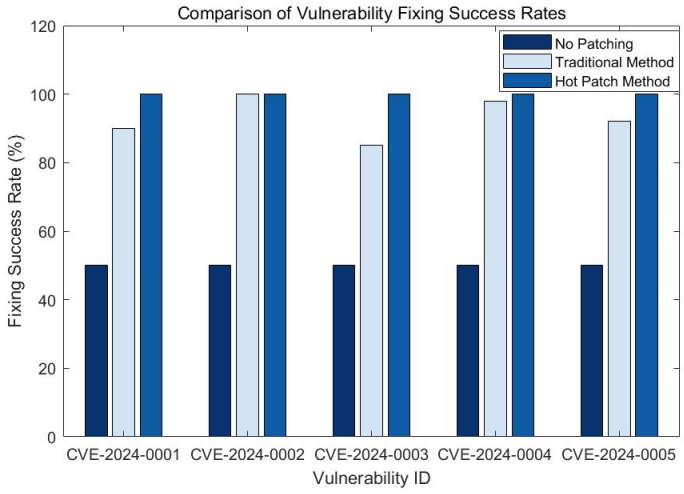

Figure 3. Comparison of Vulnerability Fixing Success Rate

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 January 2025 doi:10.20944/preprints202412.2456.v1

As can be seen in Figure 3, in terms of fixing success rate, the hot-patching method has a success rate of 100% on all vulnerability IDs, which is better than the traditional update method and the method without adding patch. The experimental results show that the hot-patching method can ensure a higher success rate of vulnerability fixing, which is crucial for the security and stable operation of embedded devices. Traditional update methods may not be able to completely fix the vulnerabilities in some cases, resulting in the system still having security risks. The hotfix method, on the other hand, ensures that every vulnerability is completely fixed through a dynamic repair mechanism.

4.3.3. System Stability Comparison Experiment and Result Analysis

In order to evaluate the impact of different repair methods on system stability, we compared the number of system crashes during the repair process of different methods. The experimental results are shown in Figure 4.

As can be seen from the figure, in terms of the number of system crashes, the hot-patching approach has 0 crashes on most vulnerability IDs, which is significantly lower than the traditional update approach and the approach with no patches added. The experimental results show that the hot-patching method can significantly improve the stability of the system and reduce the number of system crashes during the repair process, thus improving the overall reliability of the system. The traditional update method may lead to system crashes and service interruptions during the repair process, whereas the hot-patching method, due to its feature of not requiring system restart, enables the system to still operate normally during the repair process, which greatly improves the stability and availability of the system

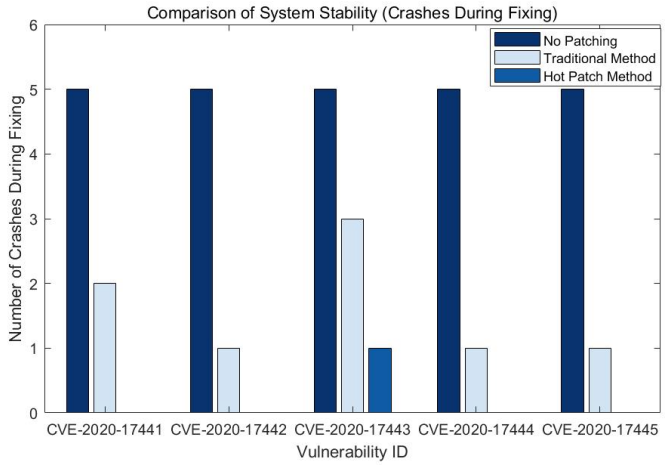

Figure 4. System Stability Comparison Chart

4.3.4. System Usability Comparison Experiment and Result Analysis

In order to comprehensively evaluate the impact of different vulnerability remediation methods on system availability, we compare the system availability after remediating a vulnerability with no patch, traditional patching methods, and hotfixing methods.

$$\begin{aligned} \text{System Availability} = 100\% - (\text{Number of Crashes} \times \text{Crash Penalty}) \\ - \left(\frac{\text{Repair Time}}{\text{Maximum Tolerance Time}} \times 10\% \right) \end{aligned} \tag{1}$$

Assumptions: crash penalty is 2%. The maximum tolerance time is 2000 μs. The experimental results are shown in Figure 5.

 doi:10.20944/preprints202412.2456.v1

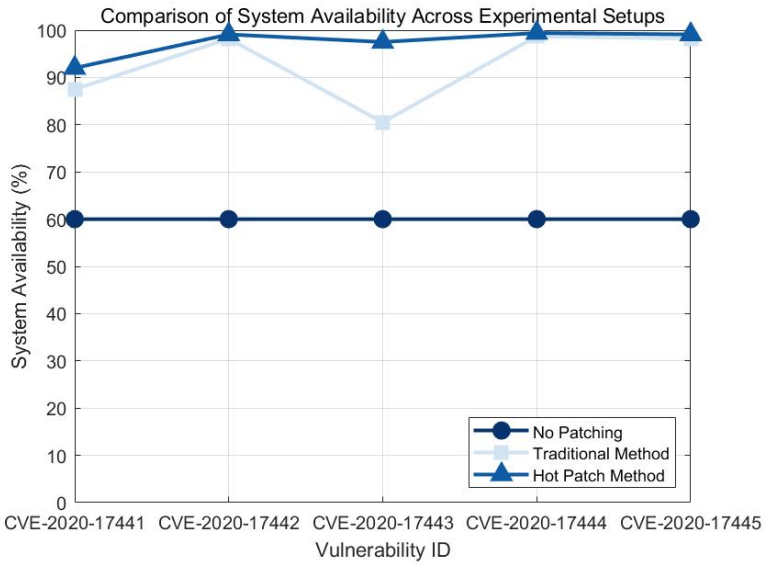

Figure 5. System Stability Comparison Chart

According to Figure 5, the hotfix method outperforms both the traditional patching method and the no-patch method in terms of system availability after remediation. For example, after the CVE-2020-17441 vulnerability was fixed, the hotfix method achieved 92% system availability, compared to 87.5% for the traditional patching method and only 60% without adding a patch. In the case of the CVE-2020-17442 vulnerability, the hotfix method had 99.1% system availability, almost a perfect score.

The results of such experiments show that the hot-patching approach is not only effective in fixing vulnerabilities, but also maximizes system availability and reduces the downtime and impact of the fixing operation. Traditional patching methods may lead to longer system downtime during the repair process, thus affecting the overall system availability. The hotfix method, however, greatly improves system stability and availability due to the fact that it does not require a system reboot, allowing the system to continue to operate normally during the remediation process.

4.3.5. Experiment and Result Analysis of Time and CPU Cycle Required for Different Patch Triggering Methods

In the experiment of different patch triggering methods, we compare the CPU cycles and time of Fixed_Patch, FPB and Debug_Monitor under different operations. The experimental results are shown in the following figures.

 doi:10.20944/preprints202412.2456.v1

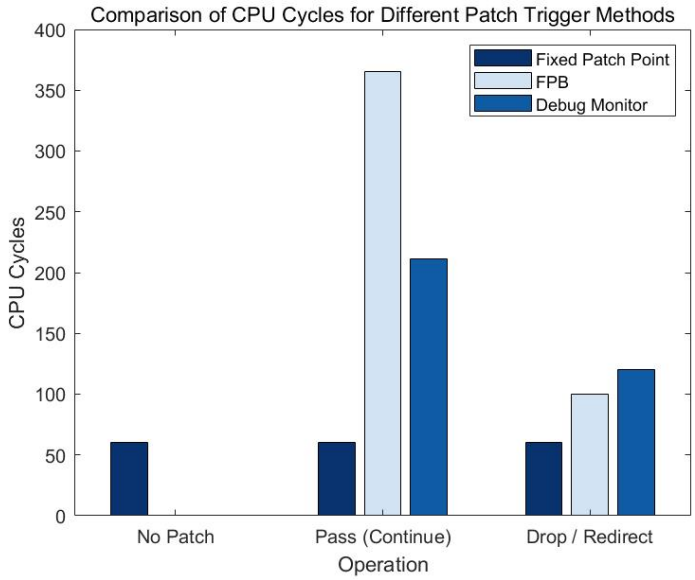

Figure 6. Comparison of CPU cycles required for different patch triggering methods

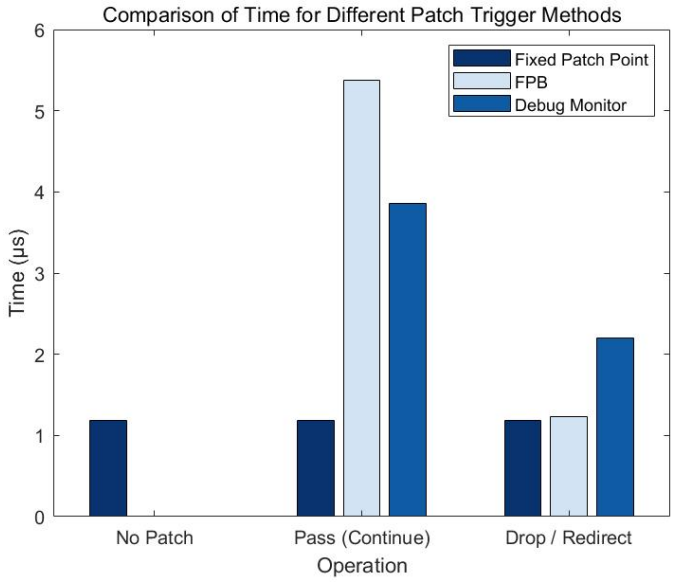

Figure 7. Comparison of time required for different patch triggering methods

From the experimental data, it can be seen that Fixed_Patch performs consistently in all cases, requires the least amount of time and CPU cycles, and is suitable for scenarios with high performance requirements. FPB mode has a higher overhead when resuming and continuing function execution (Pass), but a lower overhead when skipping or redirecting (Drop/redirect), which is suitable for scenarios with higher security requirements. Debug_Monitor has a performance between Fixed_Patch and FPB, and is suitable for scenarios that require a balance between security and performance.

Through comparative analysis, Fixed_Patch has the best performance, while FPB and Debug_Monitor provide higher flexibility and security. In practical applications, which patch triggering method to choose needs to be decided based on specific requirements and system performance indicators.

 doi:10.20944/preprints202412.2456.v1

4.3.6. Patch Delay Experiments with Different Devices and Analysis of Results

In order to evaluate the performance of different devices when applying patches, we compare the patch latency of three devices, NRF52840, STM32L475 and STM32F429, under different operations. The experimental results are shown in Figure 8.

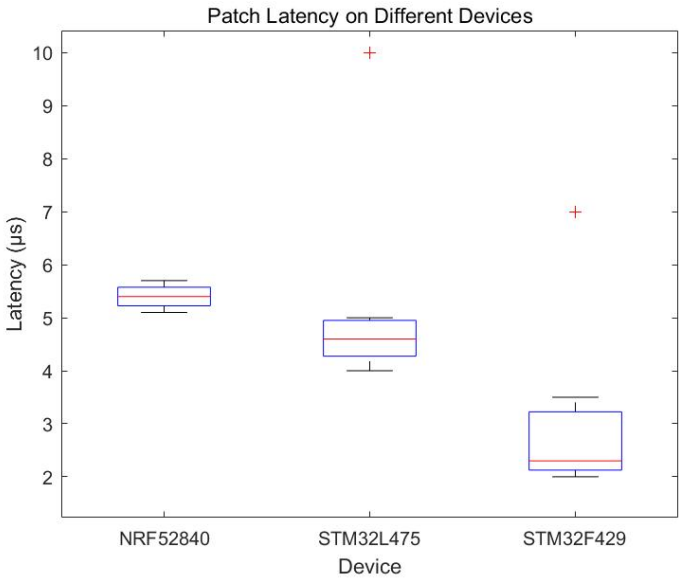

Figure 8. Patch latency on different devices

The experimental data show that the latency of the NRF52840 is relatively stable under all test conditions, with latency fluctuating between 5.1 μs and 5.7 μs, demonstrating better real-time performance.The STM32L475 has a latency between 4.0 μs and 5.0 μs for the first five tests, with a higher latency (10 μs) occurring in the seventh test, suggesting that in some cases performance may There are performance fluctuations.The latency of the STM32F429 is below 3 μs in the first five tests, but increases significantly in the sixth and seventh tests to 3.5 μs and 7 μs, respectively.

The experimental results show that there are differences in the performance of different devices when applying patches. the NRF52840 has a more stable latency, which is suitable for application scenarios with higher real-time requirements. the STM32L475 has a lower latency, but performance fluctuations may occur in some cases, which needs to be paid attention to in practical applications. the STM32F429 has a lower latency in general but increases in longer tests, which needs to be selected according to specific needs when applying the patches. The STM32F429 has a lower latency time overall, but the latency increases in longer tests, so it should be selected according to specific needs in the application.

5. Conclusions

Analyzed by the above experimental results, the hot patching approach shows obvious advantages in several key indicators. Specifically, hot patching can significantly shorten the vulnerability fixing time while improving the stability and availability of the system. In contrast, the traditional update method may lead to long system downtime and affect normal operation, while the unapplied patches may trigger frequent system crashes and service interruptions. Overall, the hot-patching approach performs well in embedded device vulnerability remediation, effectively reducing system downtime and service interruptions, and providing strong support for system stability and availability. The performance difference between different devices when applying patches also provides an important reference for choosing appropriate remediation strategies, which helps optimize the effect of patch application and overall system performance.

 doi:10.20944/preprints202412.2456.v1

References

1. Niesler C, Surminski S, Davi L. HERA: Hotpatching of Embedded Real-time Applications. *Proc. Network and Distributed Systems Security (NDSS) Symp.* 2021: 21-25.
2. Ye H, Gu J, Martinez M, et al. Automated classification of overfitting patches with statically extracted code features. *IEEE Trans. Software Eng.* **2021**, *48*(8), 2920-2938.
3. Xiong Y F, Liu X Y, Zeng M H, et al. Identifying patch correctness in test-based program repair. In *Proc. 40th Int. Conf. Software Eng.*; ACM: New York, 2018; pp. 789-799.
4. Jiang W, Liu L. Research on hot-patching technology based on VXWORKS system. *Comput. Technol. Dev.* **2017**, *27*(03), 18-22+28.
5. Zhou M, Wang H, Li K, et al. Save the Bruised Striver: A Reliable Live Patching Framework for Protecting Real-World PLCs. In *Proc. 19th Eur. Conf. Comput. Syst.* 2024: 1192-1207.
6. He Y, Zou Z, Sun K, Liu Z, Xu K, Wang Q, Shen C, Wang Z, Li Q. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *Proc. USENIX Security Symp.* 2022: 10-12.
7. Wang M. Exploration and application of remote hot deployment based on SpringBoot. *Inf. Comput. (Theor. Ed.)* **2023**, *35*(07), 1-4.
8. Holmbacka S, et al. Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems. In *Workshop on Hot Topics in Software Upgrades*, 2013.
9. Ramaswamy A, Bratus S, et al. Katana: A Hot Patching Framework for ELF Executables. In *Proc. ARES* 2010: 507-512.
10. Payer M, et al. DynSec: On-the-fly Code Rewriting and Repair. In *Workshop on Hot Topics in Software Upgrades*, 2013.
11. Chen H, Chen R-X, et al. Live updating operating systems using virtualization. In *Proc. Int. Conf. Virtual Execution Environments*, 2006.
12. Altekar G, et al. OPUS: Online Patches and Updates for Security. In *Proc. USENIX Security Symp.* 2005.
13. Dong Y, et al. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *Proc. USENIX Security Symp.* 2019.
14. Shi J, Wang Y, Su Y, et al. Analysis of MQX interrupt mechanism and interrupt program framework design based on ARM Cortex-M4. *Comput. Sci.* **2013**, *40*(06), 41-44+79.
15. Zhang J, Ling X, Wang Y. Design of a generic framework for assembly engineering for ARM Cortex-M series cores. *Small Microcomput. Syst.* **2021**, *42*(11), 2440-2445.
16. Zhou C. Design and Implementation of Server/Client Based Patch Management System. *Microcomput. Appl.* **2009**, *30*(06), 53-57.
17. Li Z, Wang P, Wang Z, et al. Flowganomaly: Flow-based anomaly network intrusion detection with adversarial learning. *Chin. J. Electron.* **2024**, *33*(1), 58-71.
18. Wang Z X, Li Z Y, Fu M Y, et al. Network traffic classification based on federated semi-supervised learning. *J. Syst. Archit.* **2024**, *149*, 103091.
19. Li Z, Zhang Z, Fu M, et al. A novel network flow feature scaling method based on cloud-edge collaboration. In *Proc. 2023 IEEE 22nd Int. Conf. Trust, Security and Privacy in Computing and Communications (TrustCom)*; IEEE: 2023; pp. 1947-1953.