Seminar Operating Systems and Virtualization WS2223
Extension of RapidPatch's Patching Model: „Argument Patch“ and Possible RapidPatch Application on Closed-Source Projects

Kai-Chun Hsieh
Technical University of Munich

Abstract

According to a statistic published by Statista [5], it shows, that until 2021, there are already 11.28 billion Internet of Things (IoT) devices connected to the internet. Simultaneously, some research revealed that many of them are running with outdated firmware and have become the main target of cyber-attacks nowadays [2]. Although the dramatic danger exposing to cyber threats, due to some reasons, many of them cannot be patched in time with existing technologies. One example is the fragmentation of hardware specifications; worries about breaking running systems in the production environment are another major reason pulling system engineers back from updating their devices.

In the previous research, Yi He et. al. published a hot-patching framework, RapidPatch, and showed in their paper [2], the possibilities to solve the above-mentioned problems by providing one-for-all patches that can run on major popular systems using different platforms with the help of eBPF[1], and also by the capabilities patching running systems without affecting runtime environments.

On top of RapidPatch, we would like to propose our optimizations in this paper with patching overheads reduced exponentially under some circumstances. And, via our proof of concept, we want to show that it is very possible to adapt RapidPatch to a close-sourced system.

1 Introduction

Due to limitations of hardware capabilities in micro-controllers of real-time embedded systems, operating systems they can use usually lack necessary security features, for instance, firewalls and all other threats isolation features, since they are designed only to react rapidly and just enough for serving client's practical requirements. This undoubtedly increased the risk of the systems being exploited.

To solve this issue, many vendors added OTA (over-the-air) updating features into their products, the most common examples are, ESP32 from Esspressif, and also, smartphones nowadays.

Although having such possibilities updating an RT embedded device, it is in many cases not feasible in real-world systems. Practically, real-time embedded systems may be made to be in charge of time-critical, security-critical and even life-critical tasks, however, with given solutions, rebooting is always demanded and unavoidable and thus always brings unacceptable downtimes. Thankfully, there are already some researchers trying to relieve this terrible situation, for instance, KARMA [1], VULMET [6] and HERA [4], and also some existing commercial solutions, such as LivePatch from Canonical.

In spite of that, the existing solutions are either not applicable for real-time embedded systems, or their application them are very limited in the practical situation [2]. For example, KARMA, and VULMET patch systems by adding trampolines into code memories. But such a strategy is not feasible for RT embedded systems, which usually use NOR-flashes as code memory, and writing on such media requires quite much time because they allow only block-wise operations. Similar obstacles stand also for LivePatch. Although HERA is the first system successfully introduced a hot-patching framework by use of hardware features on ARM Cortex-M3/M4 processors, those features are discontinued in the later ARM CPU architectures and also do not exists in some modern CPU architectures, e.g., RISC-V.

Even, though the system engineers have enough expertise and are able to apply the above-mentioned technology to patch their systems, often, they are still not able to do it since the patches are even not released by their upstream provider yet. Yi He et. al. [2] showed in their research, that even the most active real-time operating systems (Zephyr and FreeRTOS) take about 3-6 months to get a vulnerability patched, and it takes even on average more than one year in the less active real-time operating system's case, e.g., TizenOS. And these phenomena obviously make those RT-embedded systems ex-

[1] eBPF (extended Berkeley Packet Filter), a lightweight kernel VM designed for Unix-like systems

posed to zero-day attacks for a longer time frame.

For that, Yi He et. al. introduced RapidPatch, aiming to reduce the patching delay and to encourage system engineers to update their systems by helping developers make patches that can applicable at once to many devices, even many platforms, and reduces workloads for engineers required to perform integration tests on their products.

Among these, we made some further improvements, we found out that under some circumstances, RapidPatch is less helpful, due to its too-constrained model. And our solution to that is a new patching model, *Argument Patch*, extending an original RapidPatch patching model. With such a patching model, developers will have more flexibility in developing patches, but still, their behaviors will be strictly monitored by the RapidPatch system.

And later, in section 5, we will show, that our idea is totally feasible with a very simple proof of concept, whose source codes can be found on a GitHub page [3].

2 Previous Research Works

In the previous research, Yi He et. al. proposed RapidPatch as a hot-patching framework whose goal is to increase the security of RT embedded systems from 2 directions: **1.** reduces difficulties and testing efforts of patch developments in order to reduce patching delays, and **2.** attracts more system engineers to upgrade their running systems by implementing hot-patching methods supports as many platforms as possible.

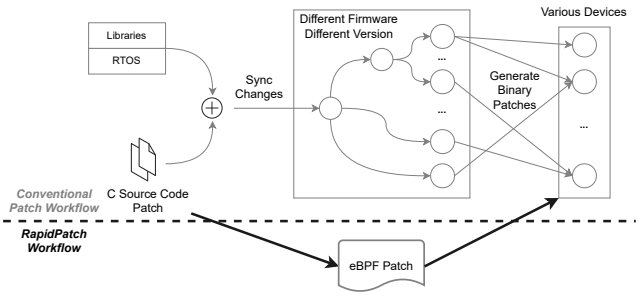

Figure 1: The Improved Patching Development Workflow Proposed by Yi He et.al. [2]

2.1 Minimize Patching Delays

To reduce the patching delay, RapidPatch's idea was to try to reduce the duplicated, complex workflows existing in the conventional patching developments, as shown in figure 1.

Their design goals are:

1. **One code for patching all various devices.**
2. **Automated test processes.**

To achieve the first goal, RapidPatch patches are made up of eBPF bytecodes, which are lightweight and portable among different platforms, just like Java bytecodes.

For the second goal, Yi He et. al. defined a complete workflow containing various tool-kits automatizing the patch development workflow, the workflow is defined in 4 phases:

1. **Patch Development**
2. **Patch Generation & Verification**
3. **Patch Deployment**
4. **Patch Execution**

Which, one of the most important utilities they proposed is *Code Verifier*. The *Code Verifier* will scan the implemented patches, and decide automatically, whether they are logically valid and do they have unsafe operations. If they are not the case, *Code Verifier* would mark them as safe and can be deployed to the devices immediately, otherwise, further careful manual code reviews and tests shall be performed.

Unsafe operations are defined by RapidPatch as operations that can produce side effects, e.g., C-function calls, and memory accesses outside of the eBPF VM sandboxes.

2.2 Compatibility & Least Patching Impacts

On top of research done by Christian Niesler et. al. [4], Xu et. al. [6] and Chen et. al. [1], RapidPatch provides two patch triggering methods, one is based on the ideas come from Christian Niesler et. al., dynamic patch triggers, whose implementation is based on hardware debuggers; the other one is extended from the ideas of Xu et. al. and Chen et. al., which resulted in a pure software patch triggering implementation more suitable for RT embedded systems, fixed patch triggers. The fixed patch triggers Yi He et. al. proposed are based on fixed trampolines implanted into the selected functions by recompiling the project and patch dispatching is based on bitmaps put in the memory.

3 Problem Statements

3.1 High Overhead with Code Replace Patch

RapidPatch contains two code patching models:

1. **Filter Patch**
2. **Code Replace Patch**

In Which, *Filter Patch* are the patches triggered whenever the vulnerable function is called. When the vulnerable function is called, the patch dispatcher of the function will decide whether the vulnerable function shall be executed, redirected to somewhere else, or skipped. Exactly like a filter. *Filter*

Patch is by design light-weight and friendly for patch developers, since it usually does not bring side effects and thus has greater possibilities to pass the automatic patch verifier.

Code Replace Patch is to replace the whole vulnerable function and is used when the vulnerable function cannot be patched by *Filter Patch* because of some fundamental logic errors in the middle of a function.

However the space complexity of *Code Replace Patch* is huge, compared to the filter patch: it is proportional to the length of the function to be patched, i.e.$O(n)$. And also, when implementing a Code Replace Patch, developers should re-implementing the whole piece of the function, not only the bugged block. This may not only bring more side-effects resulting in more potential bugs and risks but also make RapidPatch's code verifier poorly applicable since re-implementing a function will unavoidably perform a lot of *unsafe operations* defined by RapidPatch.

Our solution for this problem is an enhanced patching model extended from Filter Patch: ***Argument Patch***, which will, in the optimal cases, reduce the complexity of existing patches implemented with *Code Replace Patches* from $O(n)$ to $O(1)$.

3.2 Unknown Applicability to Closed Sourced Projects

RapidPatch is research aimed at open-sourced projects, they made a lot of assumptions in the way that patch developers are able to reach the source code. Practically, it may be useful if we can apply RapidPatch on software projects whose source codes are no more available (e.g, end-of-life products) or the projects can no longer be built (e.g., use of dependencies can no longer be built/found).

Later, we will discuss this in the section 6 and will also provide a model helping developers decide, whether RapidPatch is applicable to their systems, even when the target projects are close-sourced.

4 System Design

4.1 Argument Patch

Argument Patch is not a new patching model, but an enhancement on top of the *Filter Patch*. In the original design, the Filter Patch was too constrained, making it not flexible enough to deal with some complex situations in practice and thus developers could have no choice but to rewrite the whole vulnerable functions with *Code Replace Patch*, which as mentioned may increase the security risks and will increase the volumes of tasks and space complexity of the patch size of a lot.

The core idea of the Argument Patch is that we can have more control and more knowledge about the patch runtime, for instance, the unique identification of a caller function, however, under the control of the RapidPatch runtime system. Since Filter Patch was basically designed to receive only input arguments of the callee function according to the patch's requirement and we would like thus to provide more flexibility.

Exactly way to achieve that is, that, with an Argument Patch, the Patch Dispatcher will push only a pointer pointing to the runtime context as a handle, and the Patch Dispatcher will always guarantee that such pointer points to the initial context of the callee function. Simultaneously, the Patch Runtime will keep specification/data structure regarding such context, e.g., the function signature of the callee function. And a patch is only allowed to read/write the callee function's context via some runtime APIs under the regularisation of the given specification; execution contexts outside of the callee function's scope are principally read-only, since the correctness may not able to be confirmed by the Patch Runtime, and can be only accessed via specific APIs also.

With such a relaxed access model, we can solve more vulnerabilities via a much simpler model with much cheaper overhead, instead of the heavy Code Replace Patches.

Some practical examples using such an Argument Patch can be:

1. A vulnerable function as a caller function is not allowed to call a callee function with a specific context, although the input variables are correct. (With Filter Patch, such call request will be approved since the input variable is correct anyway)

2. A vulnerable function as a caller function is calling a callee function with wrongly constructed arguments and however is not fatal and can be reconstructed regarding to the context. (Such patch due to logical issues from a caller function can never be achieved with Filter Patch)

And more notable is that, if we observe the given usage example, we reverted the responsibility of a function being patched from the vulnerable function to the callee function where the vulnerable function made mistakes. And this is exactly how the Argument Patch is useful and can thus replace Code Replace Patch in several situations.

So, why are we taking a fresh stack-pointer from a callee function as a runtime execution context and how do we achieve acquire the identification of the caller function and also the context of the caller/callee functions?

The key is the stack layout during a function call. As an x86+GCC example shown in figure 2: the stack shown on the right-hand side is precisely the status when a function is just being called and has not started its execution yet with a fixed patch trigger; with a dynamic patch trigger, layout may be different by the ideas are the same. When the stack-pointer is precisely in the state we mentioned as a *fresh stack-pointer*. And the trick is that, with such *fresh stack-pointer*, the position of the return address (EIP as shown in the figure), is always the same and which points always to the address of the next

instruction from a *call* instruction in the caller function calling the callee function. Since the "EIP" value is a unique address in the code segment (in the kernel space at least), we can take it thus as a unique identification of the caller function. Plus, if we can know about the number of arguments the callee function needs, we can always back-calculate the addresses pointing to the execution context of the caller function and the trick for that is, GCC always pushes arguments with the data width aligning to the architectural design of the processor (e.g., 4 bytes for x86 systems).

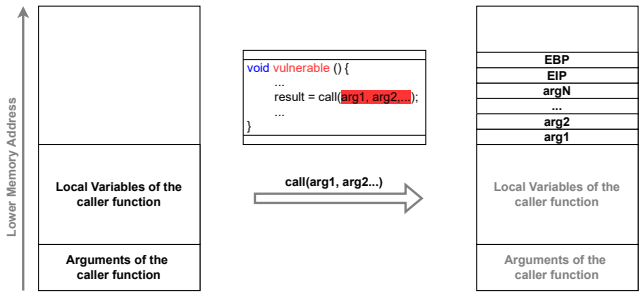

Figure 2: Stack-Frame When a GCC Compiled Function Called Another Function on an x86 System

5 Implementation

In this section, we are showing you a simple PoC implementing Argument Patches with two RapidPatch patch triggering methods of the RapidPatch:

1. **Fixed Patch Triggers**, i.e., we implant patch triggers at the entry point of selected functions.
2. **Dynamic Patch Triggers**, i.e., we set up hardware debuggers with the entry point address of the selected functions.

In our demonstrated implementation, we selected Linux as our target platform and we always have a "vulnerable" function, *add4*, which simply just adds 4 numbers a, b, c, and d provided to the function and the processes of the demonstration are always in this way:

1. Execute and show you the original result of the *add4* function before being patched.
2. Install the patch for the function.
3. Execute the patched function once again and print the difference on the console.

The given fixed-point patch demo is currently designed as only a userspace implementation but can be easily rewritten into a kernel space application without too many obstacles. In our cases, the only difference we found between a userspace implementation and kernel space one is *Calling Conventions*, i.e, the layout of the provided context from caller to callee functions. In our cases, our test device (Ubuntu 22.04, x86_64) would always compile applications in userspace using traditional x86 calling conventions, i.e., pushes arguments onto the stack; kernel space applications are always compiled with x64 calling conventions, i.e., pushes arguments firstly into registers, then, onto the stacks.

The given dynamic-point patch demo is implemented with the help of x86's debug registers, In our implementation, only DR0 and DR7 registers are used, one is for the breakpoint address, and the other is a debugger controller. In our implementation, we hooked also our own dynamic patch trigger as an ISR (Interrupt Service Routine) attached to "interrupt 1", i.e., x86 Debug Exception interrupt, which will be triggered automatically when a breakpoint set in the debug registers is hit (once enabled).

With the hardware debuggers, the methods to acquire the execution context of the function being patched are slightly different than as shown in figure 2, i.e. a fixed patch trigger one. As shown in figure 3, when an interrupt is triggered due to the patched function being called, x86 processors will firstly save some minimum context for us automatically, which are:

- **EFlags**: The arithmetic status and some permission status of the original function.
- **CS**: Code Segment, related to permissions (Ring-0, Ring-1...etc.).
- **EIP**: (RIP in x64) An address on the code segment of our caller function triggered the exception, i.e., return address ISR shall return after interrupt handling.

Along with patch triggers, we implement a patch dispatcher deciding whether take a patch or not (in the demo, we hard-encoded it as always-take). If yes, the designed patch function will be triggered and modify the second argument of the *add4* function (i.e., the *b*) by adding an extra hundred to it making the final result a hundred more than the expected value.

To be cautious here, with dynamic patch triggers, interrupted programs do not expect themselves to be interrupted at all, we should thus keep registers untampered before we return to the interrupted programs. Otherwise, the system will crash due to inconsistent context.

Also as mentioned by RapidPatch, we have to disable the hardware debugger before we return to the original program, otherwise, the exception will be kept being triggered and go into infinite loops. In our implementation, the missing part is to reenable the hardware debugger after our patch, however, here are some ideas we think probably will work:

1. Implement a new function *reEnableDebugger*. The function will be in charge of re-enabling the debugger after the patched function has done its job.

2. As shown in figure 4, we insert the original arguments (*Arguments*), the address of the function *reEnableDebugger* (EIP_2) and also the ID of the hardware debugger (ID_{DR}) indicating which hardware debugger shall we re-enable.

And then, here is the trick. After we set everything up, we will return to the original execution flow again, in this case, we will land in the callee function with patched arguments. Then, after the callee function finishes its job, he will return, in the demonstrated way, not back to the caller function, but to the function *reEnableDebugger*. In the *reEnableDebugger* function, we will pop out the ID_{DR} expected on the stack and re-enable debug register as demanded, and then return again. This time, back to the caller function (with help of the caller function's return address on top of the stack, i.e., EIP_1).

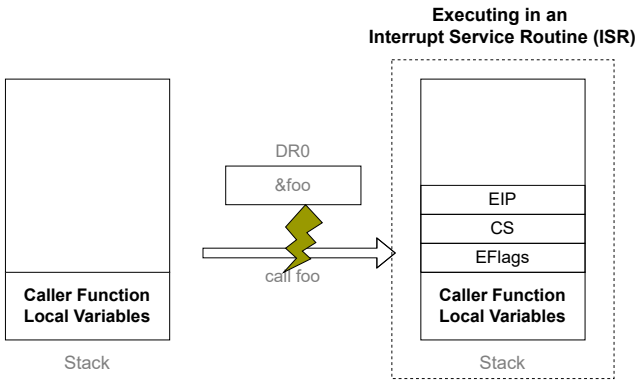

Figure 3: Stack-Frame When a Function Call Triggering a Debug Exception

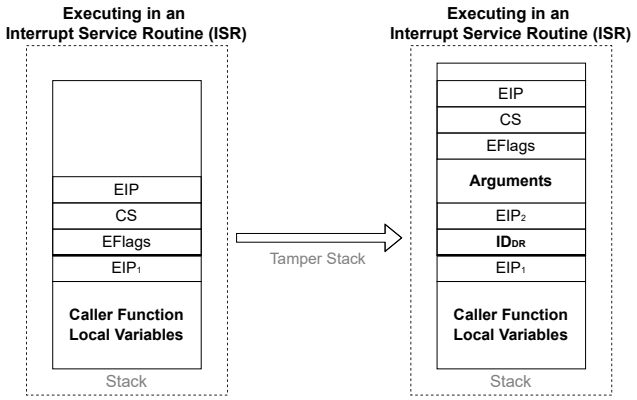

Figure 4: The Stack Layout of The Possible Solution We Found Out

6 Further Discussions

Observed from our implementation, we found out RapidPatch may also be able to be applied to close-soured systems because our implementation was only depending on hardware specifications and compiler features (i.e., calling conventions).

In open-sourced projects, it's clear, all RapidPatch patching models can be applied, including recompiling the whole project with *Fixed Patch Triggers* and *Dynamic Patch Triggers*.

Nevertheless, in close-sourced projects, the applicability of RapidPatch still depends. *Fixed Patch Triggers* are in this case no more available since re-compilation is impossible; the *Dynamic Patch Triggers* RapidPatch is however still available if the target system satisfies the following conditions according to our observation:

- Hardware debugger is available on the target platform.
- Entry point of the vulnerable function is known.
- Custom kernel modules/kernel drivers can be made (which are always available for major OSes) so that RapidPatch's ***Patch Installer*** can be run under CPU modes where hardware debuggers can be configured (e.g., Ring-0).
- Once implementing RapidPatch on platforms with memory protection functions enabled (e.g., Paging), utility functions coping userspace's stack memory to RapidPatch runtime may be required. Such as *copy_from_user(...)* function from a Linux system.

The researchers, who would like to check whether their system is applicable, can apply figure 5 to get a quick result.

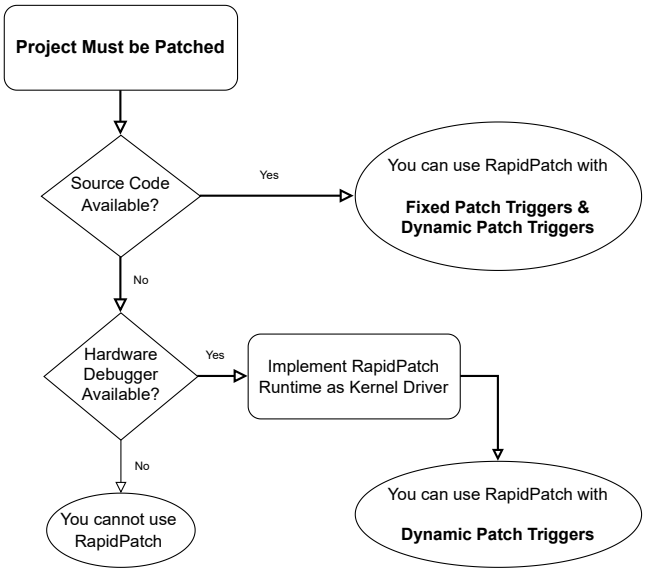

Figure 5: A Chart Helping You Decide Whether RapidPatch is Applicable to Your Project

7 Conclusions

In this paper, we proposed **Argument Patch**, which extended the original *Filter Patch* and relaxed RapidPatch's code verification model with some mandatory preconditions. With help of *Argument Patch*, patch delays can be reduced even further by avoiding unnecessary *Code Replace Patches*. And also with a PoC, we proved, that our ideas regarding **Argument Patch** are totally feasible.

References

[1] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1253–1270, 2017.

[2] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. RapidPatch: Firmware hotpatching for Real-Time embedded devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2225–2242, Boston, MA, August 2022. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/he-yi.

[3] Kai-Chun Hsieh. Github repository: Argument patch, a rapidpatch extension. URL: https://github.com/hsiehken/Argument-Patch-A-RapidPatch-Extension.

[4] Christian Niesler, Sebastian Surminski, and Lucas Davi. Hera: Hotpatching of embedded real-time applications. In *Proc. of 28th Network and Distributed System Security Symposium (NDSS)*. feb 2021. URL: https://www.ndss-symposium.org/wp-content/uploads/2021-159b-paper.pdf, doi:doi:10.14722/ndss.2021.24159.

[5] Lionel Sujay Vailshery. Number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030, Nov 2022. URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[6] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2397–2414, 2020.