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 [4], 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 Simultaneously, some research revealed that many of them are running with outdated firmware and have become the main target of cyber-attacks nowadays [2].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 also, we would like to show the possibilities of adapting RapidPatch into close-sourced systems.

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.

Benefiting from the rapid growth of semiconductor fabrication technologies, competition between different micro-controllers is turning white-hot. Manufacturers started adding more practical features to their products to make them more popular and useful, e.g., WiFi and Bluetooth. Those new fancy features unquestionably enable more potential applications and open a wider market, however, at the same time, they also increased the attack vector significantly and make those RT embedded systems suffer from cyber-criminals more easily. However, it is not totally hopeless, some vendors have already noticed that and tried to add some patching features in their hardware making their products able to be updated in time. The most commonly used strategy is OTA (Over the Air) update. Take ESP32[2] as an example, developers are able to update firmware and software of their micro-controllers by uploading them via Ethernet onto the device and then flash their device with A/B deployment schema. More common examples using OTA technologies are, for instance, smartphones nowadays.

Although having such possibilities updating an RT embedded device, it is in many cases not feasible in the systems in the real world. 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 [5] and HERA [3], 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

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

[2] A low-cost, low-power SoC micro-controller introduced by Espressif Systems

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

Getting deep dive to the bottom of the reason causing so much patching delay is, that conventional patching methods rely on human brains too much: developers have to merge changes firstly from their upstream dependencies very carefully, and then they can finally start to patch their own codes. Even in the testing phase, unit-tests and integration-tests are just like a never-ending story, especially when the vendors have very fragmented product lines.

2 Previous Research Works

In the previous research performed by Yi He et. al., they proposed a new hot-patching framework, RapidPatch, whose ultimate 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.

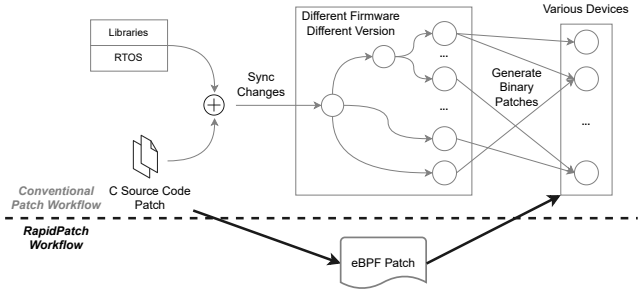

Figure 1: The Improved Patching Development Workflow Proposed by Yi He et.al. [2]

2.1 Minimize Patching Delays

RapidPatch tried to minimize duplicated and complicated tasks during patch developments by introducing a brand new patch development workflow with the following goals:

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 phase:

1. **Patch Development**:
   In this stage, RTOS developers merge changes and develop patches as they usually do, but, by design, developers should now give out eBPF source codes and configurations.

2. **Patch Generation & Verification**:
   Simply to say, developers compile, link, and verify code correctness in this stage. To help out the system engineers, Yi He et. al. developed several useful took-kits for system engineers to accelerate their jobs.

   The most pioneer tool that Yi He et. al. introduced in the RapidPatch system is a code correctness verifier: **Patch Verifier**. The tool defines *danger operations* of patch implementations, *e.g., C-function calls & modification of memory out of eBPF sandboxes*. *Danger operations* are not Taboos, they just represent that, the logic of the code may not be verified automatically and thus should be reviewed again manually properly.

3. **Patch Deployment**:
   In this phase, tested eBPF patches can be published. Patches without side effects can be shipped to vulnerable devices immediately, otherwise, further integration-tests should be performed more carefully before they are shipped.

4. **Patch Execution**:
   Then, the patches can finally run on vulnerable devices and resist cyber-attacks.

   During patch execution, the RapidPatch runtime system will check runtime context in real-time by, e.g., constraining the number of loop iterations...etc., preventing the system from malfunctioning caused by miss-configured patches.

2.2 Compatibility & Least Patching Impacts

In the previous research done by Christian Niesler et. al [3], they successfully proved the feasibility of hot-patch systems with ARM Cortex-M3/M4 processors with the help of their hardware debuggers. However, this feature is unfortunately no more equipped on the more modern ARM architectures, and also, such hardware feature does not exist on one of the most popular embedded system processor-architecture, RISC-V.

Yi He et. al. turn in their RapidPatch research to a different strategy, they tried to add their hot-patching system with mechanisms that purely rely only on software implementations. The concrete implementation of that is to re-compile the RTOS with trampolines, *a.k.a. Fixed Patch Points*, embedded in the entry points of selected functions. However, unlike the previous solutions proposed by Xu et. al [5] and Chen et. al. [1], RapidPatch uses fixed trampolines, i.e., each time a function is called, the trampoline will always be triggered and then it will check among a bitmap deciding whether there are available patches to apply with.

Pure software implementation not only made RapidPatch be compatible with major platforms but also makes the number of patches supported by the hot-patching runtime no more limited to the number of the processor's debug registers (usually 6-8 pieces). And most importantly, in their later evaluations, such design brings only some acceptable overheads.

According to Yi He et. al.'s research, although in some extreme cases, RapidPatch under software implementation mode will bring about 37.7% overheads (in a concrete Co-AP application example, it brings in which case 2 ms overhead per request), However, if the reasonable strategy is chosen, overheads are negligible (around 2%); same for RapidPatch system implemented with hardware features (overhead: 0.01% - 0.6%, worst case: 1.5%).

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

Our solution for this problem is a new 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 $O(1)$. And the reason to have such a new patching model is that the Filter Patch is originally designed as stateless and works rely only on inputs and cannot thus be applicable to the situations where patches must have been implemented via Code Replace Patches.

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).

In this paper, we will build a model in the section 4.2, with which developers can decide, whether RapidPatch is applicable to their systems, even when the target projects are close-sourced, and then, we will prove the correctness of the model with a PoC in the Implementation section 5.

4 System Design

4.1 Argument Patch

We define *Argument Patch* as following: *Argument Patch* is a patching model of RapidPatch, that intercepts function calls made by a vulnerable function where the parameters are not correct and thus making the function vulnerable. And also, to make *Argument Patch* compatible with the RapidPatch's code verifier, we have to relax the definition of *danger operations* a little bit by allowing under-controlled stack-frame accesses.

In an Argument Patch, instead of patching the vulnerable function making faulty function calls with Code Replace Patch, we patch the callee function where the invalid arguments will be given. The challenge to achieve this is to get knowledge of the current execution context, i.e., from which

line/which function the function has been called since we only want to fix the arguments if and only if the function is called by the line causing bugs. So the question is: can a function knows about its caller function? **Yes!**

Think about the stack layout during a function call. As an x86 example shown in the figure 2: if we can know how many spaces the arguments take (in x86 + GCC's case it takes always 4*n* bytes, *n* equals the number of arguments), we can back-calculate the **EIP** value shown in the figure 2, i.e., the address pointing to the next address of the *call* instruction in the caller function calling the callee function. And then, we can take such address as the signature of a caller function, since the value is pointing to a unique address in the code segment.

The context of the caller function can be retrieved in the exactly same way as the caller's signature. In the practical design, we can even decide whether to deploy the bug or not regarding the pattern of the caller function's context.

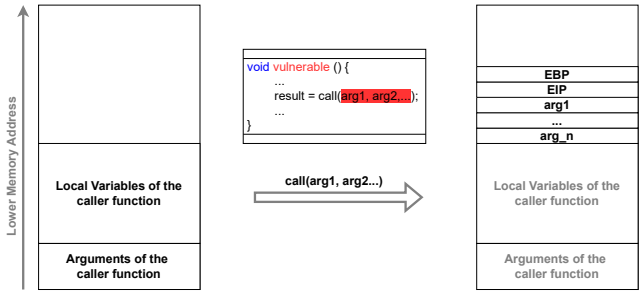

Figure 2: Stack Frame When a GCC Compiled Function Called Another Function on an x86 System

4.2 RapidPatch with Close-Sourced Project

From our experience of building prototypes for RapidPatch, we summarized the situations that can be applicable to RapidPatch.

In open-sourced projects, it's clear, all RapidPatch patching models can be applied, including recompiling the whole project with *Fixed Patch Triggers* on selected functions, and also *Dynamic Patch Triggers* implemented with hardware debug features.

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

- Hardware debugger is available on the target platform.
- We know the entry-point of the vulnerable function.

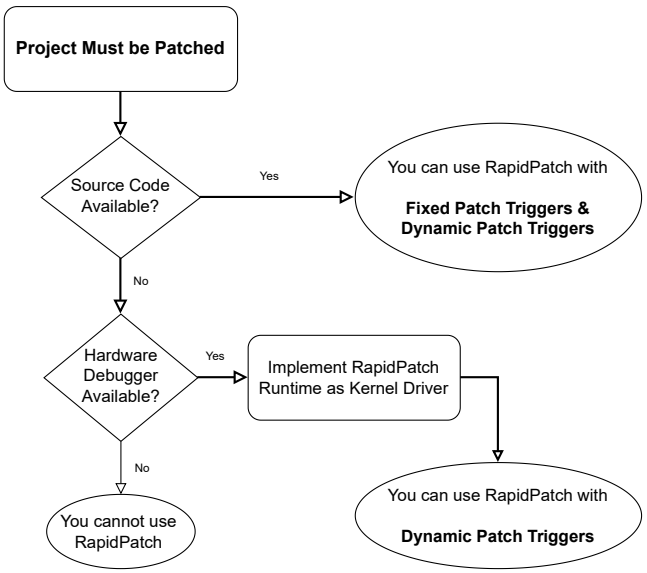

Figure 3: A Chart Helping You Decide Whether RapidPatch is Applicable to Your Project

- We can implement kernel modules/kernel drivers so that RapidPatch's ***Patch Installer*** can be run under CPU modes where hardware debuggers can be configured (e.g., Ring-0).
- Once patching system-call routines, especially on platforms with memory protection functions enabled (e.g., Paging), utility functions coping userspace's stack memory to RapidPatch runtime must be available, e.g., *copy_from_user(...)* function in the Linux system.

Later, we will prove this model with concrete implementations.

5 Implementation

TODO

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.

[3] 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.

[4] Lionel Sujay Vailshery. Number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030, Nov 2022.

[5] 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.