FortifyPatch: Towards Tamper-Resistant Live Patching in Linux-Based Hypervisor

Zhenyu Ye
Hunan University, China
yezhenyu@hnu.edu.cn

Lei Zhou
National University of Defense Technology, China
zhoulcs@nudt.edu.cn

Fengwei Zhang
Southern University of Science and Technology, China
zhangfw@sustech.edu.cn

Wenqiang Jin
Hunan University, China
wqjin@hnu.edu.cn

Zhenyu Ning*
Hunan University, China
Xinchuang Haihe Laboratory, China
zning@hnu.edu.cn

Yupeng Hu*
Hunan University, China
yphu@hnu.edu.cn

Zheng Qin
Hunan University, China
zqin@hnu.edu.cn

ABSTRACT

Linux-based hypervisors in the cloud server suffer from an increasing number of vulnerabilities in the Linux kernel. To address these vulnerabilities in a timely manner while avoiding the economic loss caused by unplanned shutdowns, live patching schemes have been developed. Unfortunately, existing live patching solutions have failed to protect patches from post-deployment attacks. In addition, patches that involve changes to global variables can lead to practical issues with existing solutions. To address these problems, we present FortifyPatch, a tamper-resistant live patching solution for Linux-based hypervisors in cloud environments. Specifically, FortifyPatch employs multiple Granule Protection Tables from Arm Confidential Computing Architecture to protect the integrity of deployed patches. TrustZone Address Space Controller and Performance Monitor Unit are used to prevent the bypassing of the Patch via kernel code protection and timely page table verification. FortifyPatch is also able to patch global variables via well-designed data access traps. We prototype FortifyPatch and evaluate it using real-world CVE patches. The result shows that FortifyPatch is capable of deploying 81.5% of CVE patches. The performance evaluation indicates that FortifyPatch protects deployed patches with 0.98% and 3.1% overhead on average across indicative benchmarks and real-world applications, respectively.

CCS CONCEPTS

• **Security and privacy → Systems security**.

*Corresponding authors.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652108

KEYWORDS

Live Patch, Arm, Confidential Compute Architecture

ACM Reference Format:
Zhenyu Ye, Lei Zhou, Fengwei Zhang, Wenqiang Jin, Zhenyu Ning, Yupeng Hu, and Zheng Qin. 2024. FortifyPatch: Towards Tamper-Resistant Live Patching in Linux-Based Hypervisor. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652108

1 INTRODUCTION

Nowadays, the security of cloud servers is a significant concern, particularly since most hypervisors in the cloud are based on the Linux kernel [20, 59, 72], which has seen an increase in the number of vulnerabilities in recent years [35, 37, 44, 55, 82]. Although these vulnerabilities are usually repaired with security patches, applying these patches to deployed cloud servers is not always simple due to the economic losses caused by unplanned shutdowns. For example, IT downtime can cost an average of $5,600 per minute, and business downtime can cost up to $300,000 per hour [63, 91].

Kernel live patching [15, 33, 41, 60, 74, 91] is a solution designed for patching the Linux kernel at runtime. However, traditional live patching solutions [15, 33, 41, 60, 74] mainly rely on the kernel to accomplish the patching process. However, the kernel might be compromised due to vulnerabilities, and the kernel code used for live patching can even be used as a weapon for Advanced Persistent Threat (APT) [39] attacks [36]. A recent solution [91] aims to fill the gap by leveraging Trusted Execution Environments (TEEs) such as Intel Software Guard eXtension (SGX) and x86 System Management Mode (SMM) to eliminate trust in the vulnerable kernel. However, it only focuses on security during the patching process, but fails to protect the patch afterwards, leaving it vulnerable to exploitation by advanced attackers. These attackers could exploit the corresponding vulnerability before the patching process to gain kernel privilege and then compromise or revert the patch after it is applied [54]. For example, "Stuxnet" [56] exploits multiple vulnerabilities for remote code execution and self-spreading. If one of them is live-patched, the attacker may be able to revert the patch to finalize the attack.

Deploying live patches to hypervisors in cloud environments poses a similar security challenge. While hypervisor protection mechanisms have been extensively studied [1, 2, 11, 31, 53] to ensure code and control-flow integrity, these protections typically focus on safeguarding the original kernel code or verified kernel modules. Moreover, adversaries might exploit legitimate approaches for dynamic code generation (e.g. eBPF [24]) or code modification (e.g. kprobes [40]) to threaten the security of live patches.

Though using a memory access control mechanism enforced by a higher privilege (e.g., x86 System Management Mode (SMM) [69], Arm TrustZone Address Space Controller (TZC) [4], RISC-V Physical Memory Protection (PMP) [25]) offers a more robust security guarantee, these mechanisms often lack flexibility and are not suitable for protecting live patches. Typically, such mechanisms divide the physical memory into a few continuous regions and assign varying access permissions to each region. However, a single patch may require multiple contiguous regions. Therefore, the limited region number supported by these mechanisms may not be sufficient to meet the requirements of live patching scenarios.

Recently, Arm has introduced Confidential Compute Architecture (CCA) [5] to protect virtual machines under untrusted hypervisors. It offers fine-grained memory access control through the Granule Protection Table (GPT) [9]. The ability to protect against privileged attackers and the flexible memory access control make CCA an excellent choice for patch protection in live patching scenarios. However, the access control mechanism in GPT is specifically designed for secure VMs solutions, where the attacker and victim are separate and located at different privilege levels. In contrast, the deployed patch is coupled with the vulnerable hypervisor in the live patching scenario, introducing additional complexity.

In this paper, we present FortifyPatch, a tamper-resistant live patching system that aims to prevent post-deployment manipulations of the patches with a set of Arm hardware features. Specifically, FortifyPatch utilizes the GPT to isolate live patches from the vulnerable hypervisor. TZC and Performance Monitor Unit (PMU) are used to protect the patch from being bypassed. In order to implement FortifyPatch, we have identified three major technical challenges. **C1: The attacker and the patch are sitting in the same privilege.** Sophisticated attackers can exploit vulnerabilities in the hypervisor and obtain privileged access at the hypervisor level, which can be further used to compromise or bypass the applied patches. Existing techniques cannot fully address this challenge and we employ a group of hardware features to achieve the protection. (Detailed in §4.3). **C2: Specific patches may cause changes in the memory layout.** If a patch [22, 23, 77] alters the size of a global variable or inserts/deletes a global variable, the addresses of data following the changed variable would change as well. However, updating all references to these data would not be practical in real-world scenarios. To mitigate this issue, FortifyPatch leverages well-designed traps to achieve a practical deployment (Detailed in §4.1.2 and §4.3.3). **C3: Adopting the security policies introduces notable performance overhead.** Protecting deployed patches in FortifyPatch requires context switches to swap the GPTs, which can cause a notable slowdown. To improve the performance, we design specific strategies to decrease the number of switches and minimize the overhead of each switch (Detailed in §4.5).

We implement a prototype of FortifyPatch on the Arm Fixed Virtual Platform (FVP) [8] to demonstrate its functionality. The evaluation shows that FortifyPatch is capable of deploying 81.5% of recent CVE patches. We also integrate FortifyPatch with Samsung Islet [67] to show its ability to work with industry-level CCA-enabled hypervisors and secure VMs. While FVP ensures functionally accurate results, it is not cycle-accurate. Thus, we implement another prototype on Raspberry PI 3B+ [61] to evaluate the performance overhead. The evaluation with indicative benchmarks and real-world applications indicates that FortifyPatch introduces 0.98% and 3.1% performance overhead on average to benchmarks and real-world applications, respectively.

In summary, we make the following contributions:

- We present FortifyPatch, a tamper-resistant live patching system for hypervisors in the Arm cloud environment. To the best of our knowledge, FortifyPatch is the first live patching system capable of defending against post-deployment attacks. The source code of FortifyPatch is available at https://github.com/HammerSecurity-Lab/FortifyPatch.
- We utilize an improved GPT switching scheme to isolate the patch from the hypervisor with low-performance impact. We also employ TZC and PMU to prevent patch bypassing, improving the security of deployed patches.
- We propose well-designed traps to eliminate the memory layout changes caused by patches involving changes in global variables, which helps to deploy most patches without practical issues.
- We implement FortifyPatch and evaluate it with real-world CVE patches. The results illustrate that FortifyPatch successfully deploys 81.5% of CVE patches. Performance evaluated by benchmarks and real-world applications shows that FortifyPatch protects deployed patches with an average overhead of less than 3.1%. We also show that FortifyPatch can work with industry-level CCA-enabled hypervisors and secure VMs.

2 BACKGROUND

2.1 Live Patching

Live patching allows vulnerabilities to be patched at runtime without a reboot. Existing live patching solutions can be classified into three categories: instruction-level patching [15, 34, 57], function-level patching [14, 33, 41, 60, 74, 91], and full replacement [38]. The instruction-level patching and function-level patching replace the vulnerable instructions or functions, respectively. Full replacement, also known as live updating, replaces the entire kernel at runtime.

The differences among these categories are shown in Figure 1. The left part of the figure describes a subset of the patch [77] for CVE-2020-13974 [18], while the right part demonstrates different live patching schemes. A full replacement mechanism updates the entire kernel image to adopt the change in function `k_ascii`. Instruction-level patching places a trampoline at the start of the changed instructions to redirect execution to the patch instructions. Function-level patching works similarly, except that the trampoline is placed at the start of function `k_ascii`.

Note that the patch inserts two global variables in the `.bss` section, which results in the address change of the following variables

```
--- a/drivers/tty/vt/keyboard.c
+++ b/drivers/tty/vt/keyboard.c
@@ -127,7 +127,11 @@ static DEFINE_SPINLOCK(func_buf_lock); /*
       guard 'func_buf' and friends */
    static bool dead_key_next;
-   static int npadch = -1;
+
+   /* Handles a number being assembled on the number pad */
+   static bool npadch_active;
+   static unsigned int npadch_value;
+
    static unsigned int diacr;

@@ -882,10 +886,12 @@ static void k_ascii(struct vc_data *vc,
       unsigned char value, char up_flag)
      base = 16;
    }

-   if (npadch == -1)
-     npadch = value;
-   else
-     npadch = npadch * base + value;
+   if (!npadch_active) {
+     npadch_value = 0;
+     npadch_active = true;
+   }
+
    ...
  }
```

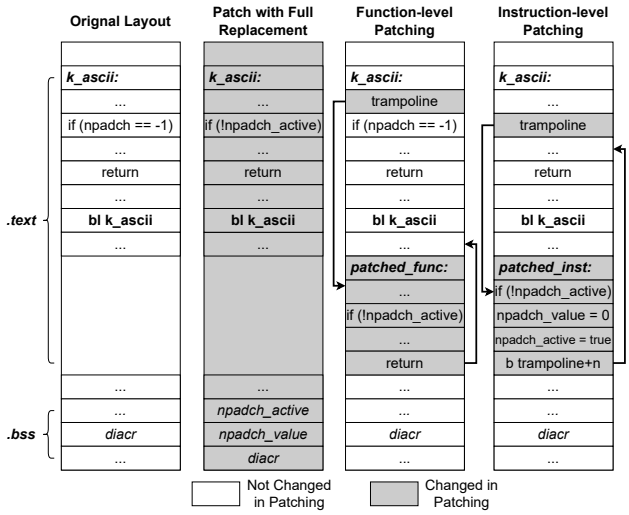


Figure 1: A Subset of Patch for CVE-2020-13974.

(e.g, `diacr`), as shown in the *Patch with Full Replacement* of Figure 1. Therefore, any instruction referring to them need to be updated, which presents practical challenges to existing instruction- and function-level patching.

2.2 TrustZone Address Space Controller and Granule Protection Table

TrustZone Address Space Controller (TZC) [4] enforces access restrictions between processors or peripherals and specific physical memory regions. It assigns access permissions to a memory region and restricts memory access from processors in different secure states according to the permission. For instance, a memory region can be configured to permit access only from a processor running in the secure state. However, each memory region needs to be continuous and TZC only supports up to eight regions, which limits its flexibility.

Arm Confidential Compute Architecture (CCA) [5] introduces the Realm Management Extension (RME) [9] in the Armv9.2 architecture and divides the execution environment into four worlds. The EL3 defined by TrustZone is now independent of Secure World and forms Root World, which owns the highest privilege over the system. Normal World and other parts of Secure World remain the same as those defined in TrustZone. A new world named Realm World is introduced to hold Realm VMs for confidential computing.

To provide memory isolation among the worlds, Arm CCA introduces GPT and assigns the physical memory pages to various Physical Address Spaces (PAS). The Granule Protection Check (GPC) is performed after the address translation to validate whether access to a memory address matches the restriction enforced by the GPT. If the check fails, a Granule Protection Fault (GPF) is raised to block the illegal access. To prevent manipulation, the control registers related to GPT and GPC can only be accessed in Root World, and the GPT itself should be in the Root PAS. **According to Arm, the TZC is expected to work with GPT instead of being replaced by GPT in Arm CCA [85].**

3 THREAT MODEL AND ASSUMPTIONS

Major Assumptions. We assume a benign but vulnerable Linux-based hypervisor is deployed in the cloud, equipped with Arm CCA support to provide secure VMs for cloud users. The vulnerable hypervisor may be exploited by attackers, but it cannot access the code and data inside the secure VMs due to the protection enforced by CCA. However, the hypervisor itself maintains critical data that may be targeted by attackers. The attacker can modify the hypervisor-level memory or execute payloads to gain information from the hypervisor. She can also interfere with the live patching process with her high privilege to prevent the exploited vulnerabilities from being fixed. Alternatively, the attacker can periodically check whether the vulnerabilities have been repaired by live patches and manipulate the patches if necessary.

We assume the cloud provider is trustworthy and the hypervisor code is securely loaded, while the correctness of the patch can be guaranteed by existing work [47, 48, 76, 81, 83, 84, 87]. The hardware architecture is considered trusted. We also assume a secure communication channel has been established between the cloud server and a trusted machine using a dedicated Ethernet interface or serial port, which is typically included in server machines for maintenance purposes.

Control-Flow Integrity. Enforcing the Control-Flow Integrity (CFI) of the hypervisor is an orthogonal research question that is independent of the primary focus of this work. Existing mechanisms [17, 29, 80, 89] normally rely on code instrumentation, which only works for the original kernel code or verified kernel modules. However, dynamically loaded code introduced by approaches like eBPF [24], kprobes [40] and ftrace [64] poses challenges for these mechanisms. Attackers might exploit the legitimate usage of these approaches to manipulate the live patches. We assume such imperfect CFI mechanisms are deployed in the hypervisor. It ensures that the attacker cannot use unauthorized function pointers to mislead the control flow, as long as the original kernel code is executing and its code integrity is maintained.

Code Integrity. Similar to the issue in CFI, current code integrity protection approaches [11, 12, 19, 30, 32] are vulnerable to the dynamically loaded code. Typically, these mechanisms monitor and intercept updates to the kernel page table to guarantee kernel code integrity. However, the dynamically loaded code can introduce code that is not monitored by these mechanisms, compromising the guarantee of code integrity. FortifyPatch assumes that attackers may attempt to manipulate the kernel code and kernel page table to bypass the deployed kernel patch.

Out of Scope. The focus of this work is on protecting the live patching process and deployed patches to prevent the exploitation of known vulnerabilities. Implementing supplementary mechanisms (e.g., Control Flow Integrity, Data Execution Prevention, Supervisor Mode Access Prevention, and so on) which protect the kernel from unknown vulnerabilities used to circumvent patches, falls under a separate topic. Additionally, the denial-of-service attack may prevent the live patching process, but this would be detected by FortifyPatch due to patch failure. In such a scenario, the cloud provider may resort to a traditional patching process to prevent further damage to the system.

4 SYSTEM DESIGN

Figure 2 shows the overview of FortifyPatch. We employ the Patch Generation Module (§4.1) on a local machine to generate a binary patch from the corresponding code patch. Next, the binary patch is directly transmitted to the Root World of the cloud server via a secure communication channel. The Patch Deployment Module (§4.2) receives the patch and applies it to the running hypervisor. Meantime, the Patch Protection Module (§4.3) ensures the deployed patch would not be manipulated or bypassed with the assistance of GPT, TZC, and PMU. Specifically, a multiple-GPT scheme similar to Shelter [90] is applied to make sure the patch can be executed at the hypervisor level without manipulation. Note that simply reusing Shelter's scheme leads to heavy performance overhead, and we design specific policies to improve the performance (§4.5). Moreover, to prevent bypassing of the deployed patch, we utilize TZC to protect the integrity of the kernel code and verify the page table for the kernel code timely via PMU interrupts. To facilitate data access across the hypervisor and deployed patches, the Data Proxy Module (§4.4) is also introduced.

4.1 Patch Generation Module

To avoid complicated context, FortifyPatch chooses function-level patching following state-of-the-art solutions [14, 33, 41, 60, 74, 91]. The Patch Generation Module, situated on a local machine, is responsible for generating a binary patch from the corresponding source code patch. To reduce the amount of human effort required, we developed a patch generation tool that automates the process. This tool automatically merges a source code patch to the hypervisor source tree and compiles it to create a patched hypervisor image. Subsequently, the binary representation of the patched global variables and functions is extracted to generate the patch. Furthermore, the generated patch includes markers indicating the removal of any global variables and functions. It is worth noting that binary comparison is not required in this scenario.

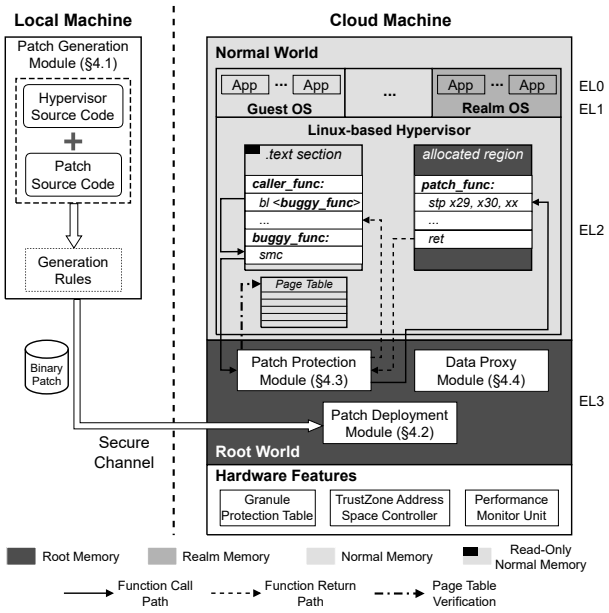


Figure 2: Overview of FortifyPatch.

4.1.1 Patch for Instructions. Once the source code patch contains modifications to code inside a function, the binary patch generated by FortifyPatch includes all instructions within the function. However, since the address of the patched function is not identical to that of the vulnerable function after deployment, the relative addressing inside the function is affected. For example, the function might include a reference to certain variables, typically obtained by calculating the offset between the instruction and the data. Unfortunately, the offset is unknown during patch generation since the memory address to place the patch has not yet been determined. To overcome this challenge, FortifyPatch replaces these instructions with placeholders during patch generation and completes them before deployment, when the offset is fixed.

4.1.2 Patch for Global Variables. The source code patch may also modify global variables, which can lead to practical issues, as mentioned in **C2**. kpatch [33] and ksplice [60] leverage shadow variable [42] to add fields to existing data structures without changing the data layout. However, a shadow variable is bound to an existing object, and using it for adding a global variable would introduce complexity in the patch code. Moreover, the shadow variable mechanism is not able to handle the situation that the size of a global variable has been changed.

To address this challenge, we carefully classify the potential global variable modifications and handle them accordingly.

Change in Data Value. This category only modifies the value of the global variable, and FortifyPatch just records the address and new value of the variable in the patch.

Change in Data Size. When the size of a variable changes, it can affect the memory layout and cause the addresses of the following data to change. To avoid updating an overwhelming number of references, FortifyPatch does not directly modify the original data. Instead, we place the changed data in a separate location in the patch generation stage.

Inserting New Data. Similar to the process for changing data size, newly inserted data is placed in a separate location instead of being directly inserted into the original location. When new data is inserted, there are typically corresponding changes to instructions within functions referring to the inserted data. To handle these changes, FortifyPatch replaces the references with placeholders.
Deleting Old Data. FortifyPatch simulates the variable deletion via zeroing the corresponding memory in the hypervisor. As for the changes to instructions related to the deleted variable, FortifyPatch requires no special handling since they are included in the instruction patch.

4.2 Patch Deployment Module

The Patch Deployment Module keeps itself executing in Root World for a high-security guarantee. It receives the patch generated by the Patch Generation Module via a secure communication channel that is only available to Root World and deploys the patch to the running hypervisor.

4.2.1 Memory Allocation and Placeholders Substitution. When a patch request is received, FortifyPatch leverages the vulnerable kernel to allocate memory for the patch. The allocated physical memory is then assigned to Root World via GPT. Next, the patch, along with its metadata, is stored in the allocated memory. The metadata includes information such as the address of the target function or data and its corresponding physical address. The virtual address assigned by the kernel is also stored as part of the metadata. With this virtual address, the Patch Deployment Module replaces the placeholders for data accesses that were inserted at the patch generation stage with runtime addresses.

Note that the vulnerable kernel may manipulate the memory allocation and return a used memory region (e.g., region for kernel code), FortifyPatch verifies the physical address of the allocated memory before using it.

4.2.2 Patch Deployment. To redirect the control flow from the vulnerable function to the patched version, FortifyPatch places a trampoline at the start of the vulnerable function. For the patches that change the value of global variables, FortifyPatch chooses the in-place replacement strategy for simplicity and isolates it from the hypervisor using the Patch Protection Module. Regarding the variable changes involving memory layout change, FortifyPatch stores the new version of the variable in the memory region allocated for the patch, and the Patch Protection Module ensures that access to the variable is redirected to the new version.

4.3 Patch Protection Module

As described in **C1**, the attacker owns the hypervisor-level privilege and could compromise or bypass the applied patch. To maintain the integrity, FortifyPatch needs to make the patch `executable` but **NOT** `writable` to the potential malicious kernel. Bypassing the patch can be achieved by replacing the trampoline or manipulating the page table of kernel `.text` section, thus FortifyPatch also focuses on protecting the kernel code and its page table.

4.3.1 Patch Integrity Protection. A recent work [90], *Shelter*, presents a multi-GPT scheme to protect user-level applications. In light of this design, we leverage a similar scheme to simulate the permission-based protection required by FortifyPatch. Figure 3 demonstrates the multi-GPT scheme for the patch in Figure 1. We prepare two GPTs for processors in the system, and each processor dynamically switches between the two GPTs on the fly. The Hypervisor GPT (H.GPT) assigns the memory of the hypervisor to Normal World, leaving the memory of the patches to Root World. This GPT is configured for hypervisor execution and guarantees the security of the patch. The Patch GPT (P.GPT) assigns the patch memory to Normal World while keeping hypervisor instructions in `No-Access` state. This enables secure execution of patches without manipulation by a compromised hypervisor. The *Function Call Path* and *Function Return Path* in Figure 2 illustrate the execution path of a live patch. At boot time, the Patch Protection Module makes all processors use the H.GPT to ensure normal execution. When the vulnerable function is called, the trampoline (i.e., the `smc` instruction) takes the execution to the Patch Protection Module. The Patch Protection Module switches the active GPT from the H.GPT to the P.GPT and redirects the execution to the patched function. The return of the patched function or calls to the hypervisor functions from the patched function leads to a GPF since the target to return or call is not accessible under the P.GPT. The Patch Protection Module handles the GPF and switches the active GPT back to H.GPT. By dynamic switching between two GPTs, FortifyPatch enables the execution of a patch at the hypervisor level while ensuring its integrity. Note that Processor B in Figure 3 represents a core that is not executing the patched function and it keeps using H.GPT until its execution falls to the patched function.

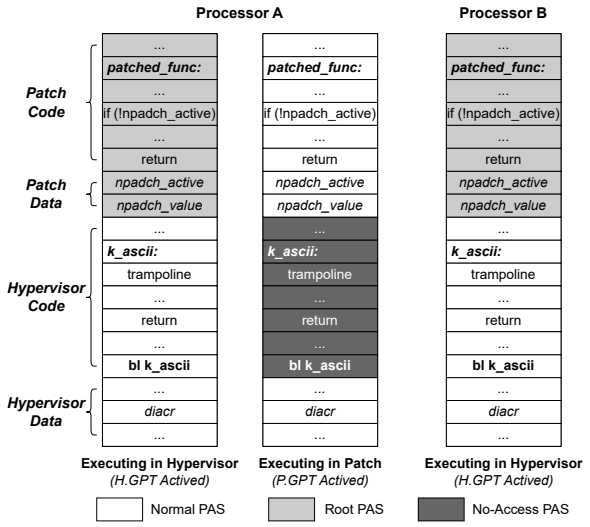


Figure 3: Multi-GPT Scheme for the Patch in Figure 1.

In a multi-processor system, each processor's GPT can be unidentical. FortifyPatch ensures that only processors executing a patch use the P.GPT, while others work with the H.GPT. To address the security issue caused by TLB sharing [90], the Patch Protection Module invalidates TLB entries on GPT switches and disables the TLB sharing once running a patch.

Note that simply reusing this multi-GPT scheme of *Shelter* does not completely satisfy the live patching scenario. This scheme places the deployed patch and the hypervisor in different

PAS, despite the fact that the patch is actually a part of the hypervisor. Frequent cross-PAS data access and function calls would trigger a large number of GPT switches and cause significant performance overhead. For instance, executing a C program with `getpid()` in Shelter involves only 4 GPT switches, while running it with 10 patches in FortifyPatch would trigger more than 4 thousand GPT switches in naive implementation. FortifyPatch solves this issue by carefully designed performance improvement strategies, which are discussed in §4.5.

4.3.2 Kernel Integrity and Page Table Verification. Once the kernel code integrity is compromised, the attacker can directly manipulate the trampoline to bypass the patch. As mentioned in §3, existing kernel code integrity protection approaches [11, 12, 30, 32] aim to monitor and intercept updates to the kernel code's page table to guarantee code integrity. These approaches would be compromised by dynamically loaded code which is out of monitoring, and also may introduce considerable overhead due to frequent interception.

Observing that the kernel `.text` section is loaded to a continuous physical memory region at boot time, FortifyPatch utilizes a single slot in TZC to mark this region as read-only to non-secure access. This protection remains intact regardless of whether attackers attempt to write the `.text` section directly or launch double-mapping attacks to bypass the page table protection. Legitimate modifications to the kernel code, such as kprobes, are achieved through requests sent via an `smc` instruction to the Patch Protection Module, which further helps to insert the probe. To prevent malicious usage, a whitelist is employed to verify registered probe handlers based on code signature. The verified probe handler is then protected by GPT, similar to how patches are protected. Other kernel trace functionalities can adopt similar protection measures even if they do not require kernel modifications. To summarize, we permit the legitimate modification to the kernel code via traping and whitelisting.

As the kernel page table can be manipulated by the attacker, she can remap the virtual address of the kernel to a physical address outside the protected `.text` region. To mitigate this issue, FortifyPatch adopts a lightweight verification mechanism to ensure the virtual address of the `.text` section is not remapped. We reserve a counter in each processor's PMU to monitor the write to the `TTBR` register. The `TTBR` register indicates the base address of the page table and is written during every process switching. Once the event occurs a predefined number of times, PMU raises an overflow interrupt, which is handled in Root World to verify the related page table entries for `.text` section. The predefined number helps to strike a balance between achieving stronger security guarantees through more frequent verification and maintaining better performance by minimizing the verification process. Moreover, the `VBAR` register is also verified upon the interrupt to ensure the entrance of the hypervisor is not manipulated.

4.3.3 Global Variables Protection. If a patch changes the value of a global variable, simply replacing the value of the variable would leave it susceptible to further manipulation. Thus, FortifyPatch assigns the memory page containing the variable to Root World to prevent hypervisor-level manipulation. It ensures that any hypervisor access to the variable results in a GPF, which is further handled by the Data Proxy Module.

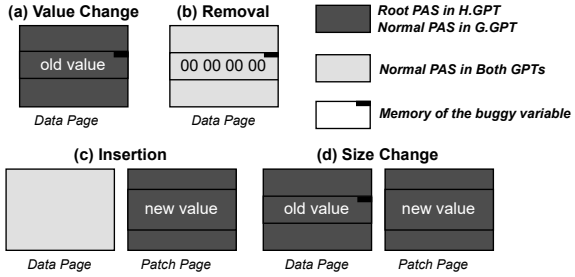


Figure 4: Data Proxy Overview of FortifyPatch.

As mentioned in **C2**, changes to a variable affecting the memory layout may cause practical issues. To address this issue, FortifyPatch adopts different mechanisms for layout changes caused by various reasons as shown in Figure 4. To remove a variable, FortifyPatch clears the corresponding memory while preserving the addresses of other data. For variable insertion, FortifyPatch places the variable into the memory region allocated for the patch instead of directly inserting it into the kernel data pages. Regarding variable size changes, FortifyPatch marks the data page containing the original variable as Root World and redirects access to the new one in the patch page. Instructions referring to these variables are updated accordingly.

4.4 Data Proxy Module

FortifyPatch leverages the GPT to protect the patched global variable and redirect its access to the Root World. However, because GPT has a minimum granularity of 4KB memory pages, the scope of protection and redirection expands to the entire memory page. As a result, hypervisor access to other data on the same page also leads to a GPF, which disrupts the normal execution.

To address this issue, the Data Proxy Module carefully handles the GPF caused by access to the protected memory page and acts as a proxy to facilitate hypervisor-level data access. Specifically, the Data Proxy Module directly operates the corresponding memory from Root World according to the instruction that caused the fault. However, additional verification applies if the hypervisor attempts to write a variable in the patch. For variables that are not expected to be modified, the Data Proxy Module denies access and alerts the system of potential threats. For other types of global variables in the patch, the Data Proxy Module checks the address of the instruction launching the memory access. Since the kernel code integrity is guaranteed, FortifyPatch considers access from the hypervisor `.text` section to be valid.

4.5 Performance Improvement

The deployment of the patch typically does not affect the system performance, but the switching of the execution between the hypervisor and patch can sometimes slow down the system. To address this challenge (**C3**) and reduce the overhead, we focus on narrowing the performance impact from the following perspectives.

Reducing the number of traps required. In an ideal design, only the patch functions are assigned to Normal World in the P.GPT to prevent any potential manipulation. However, since the patch function is actually part of the hypervisor, it involves access to the hypervisor data, which results in frequent traps and slows down

the system. Through serious security analysis (Detailed in §6), we consider assigning hypervisor data to Normal World would not weaken the overall threat model while significantly reducing the number of traps. As a result, FortifyPatch marks the hypervisor data as Normal World in the P.GPT. To prevent security issues, the Patch Protection Module ensures the address mapping of the patch is not manipulated before switching to the P.GPT.

Reducing the operations in the GPT switching. To minimize the number of operations required in the GPT switching process, we avoid the allocation and update of GPTs at switch time. Instead, the H.GPT and P.GPT are created in the patching process. Once a patch is deployed, the Patch Deployment Module notifies the Patch Protection Module to mark the hypervisor data sections and the patch as Normal World in the P.GPT, while keeping hypervisor code sections as `No-Access`. During GPT switching, FortifyPatch simply changes the value of *GPTBR_EL3* register and flush TLB.

Reducing the overhead of context save/restore. As explained in §4.3.1, the switch of the execution between the hypervisor and patch requires the switch between Normal World and Root World, which involves context save and restore operations. In a standard design, all 31 general-purpose registers are saved to the stack during the context save operation, and their values are restored in context restore. However, not all of them are required for a simple operation like switching the GPT. Our GPF handler only uses 4 registers and thus only these 4 registers are saved and restored during the switch.

In our experiment using a simple C program with `getpid()`, these policies result in a reduction of approximately 66% in GPT switches. Additionally, there is a decrease of over 70% in performance overhead for each switch.

5 IMPLEMENTATION

To validate FortifyPatch's functionality, we implement a prototype on the official Arm simulator Fixed Virtual Platform (FVP) [8]. The offline patch generator is developed with C while the online software stack is based on Arm Trusted Firmware-A (TF-A) arm_cca_v0.3 [10] which supports the functionality of CCA.

Patch Generation Module. The patch is generated from the binary compiled from the patched hypervisor code. With `system.map`, we extract the patched global variable and binary representation of the patched function. As the patch is not placed into the original address, the `bl` instructions inside the patched function are replaced with `blr` instructions to overcome the branch offset limitation. To work with the `blr` instruction, four additional instructions are used to load the address of the target function into the related register and two more instructions are used to resume the register. A similar procedure is applied to `b` instructions targeting an address outside of the patch. For `adrp` instruction, the maximum encoded offset can be 4GB, which is sufficient for the patch. Thus, FortifyPatch just replaces the offset encoded in the instruction with placeholders during patch generation.

Patch Deployment Module. FortifyPatch redirect the execution to the `__vmalloc` function to allocate page-aligned memory for the patched function and data. The Patch Deployment Module retrieves the physical address of the allocated memory via `AT S1E2R` instructions and ensures it does not overlap with kernel code and other patches. Next, the Patch Deployment Module replaces the placeholders in the patch based on the virtual address allocated by the kernel. Moreover, FortifyPatch employs the `smc` instruction as the trampoline since the address mapping of this virtual address may be manipulated.

Patch Protection Module. During boot time, we set the *SCR_EL3-.GPF* bit is to trap all GPF to Root World. While handling a GPF, we examine the *ESR_EL3* to determine whether it is caused by an instruction access. If this is the case, we verify the virtual address encoded in *ELR_EL3* with the `AT S1E2R` instruction to ensure that the mapping for the patch is valid. Then, the GPT is switched by placing the corresponding GPT base address to *GPTBR_EL3*.

The kernel `.text` section in our prototype resides in 0x80081000-0x80a00000. We define this region in TZC by filling these addresses to the corresponding `REGION_BASE` and `REGION_TOP` register, respectively. Next, we set the `REGION_ID_ACCESS.nsaid_wr_en` bit to 0 to ensure the region is read-only for Normal World.

The counter PMEVCNTR0_EL0 in each processor's PMU is reserved for FortifyPatch. By specifying the evtCount bit of the register PMEVTYPER0_EL0 to 0x1c, the reserved counters monitors the write to `TTBR` register. The counter's initial value is set to 0xFFFFFFFFFFFFFFFF minus the predefined event count to ensure it overflows after the event occurs a specific number of times. We enable the PMU overflow interrupt by setting PMINTENSET_EL1.P<0> bit and route the interrupt to Root World by configuring corresponding GICD_IGROUPR1 registers in Arm Generic Interrupt Controller (GIC). The page table verification is achieved by confirming the value of 387 page table entries related to kernel `.text` section is not modified. After the verification, FortifyPatch resets the counter to the initial value.

Data Proxy Module. For a GPF caused by data access, we use *ELR_EL3* and *FAR_EL3* to determine the address of the instruction triggering the GPF and the address of the data being accessed, respectively. Once we have validated the instruction address, the Data Proxy Module simulates the data access operation. At present, FortifyPatch has implemented the proxy for 54 variants of 12 widely used data access instructions, such as `STR`, `LDR`, and so on.

6 SECURITY ANALYSIS

The patch deployment in FortifyPatch is carried out in Root World, which means that a hypervisor-level adversary cannot directly prevent the deployment. However, the attacker with the hypervisor-level privilege may attempt to manipulate or bypass the patch.

Attack on Trampoline Deployment. The trampoline instruction is deployed to the start of a vulnerable function, but the attacker may want to prevent the trampoline deployment by providing a fake virtual address of the function. However, FortifyPatch locates the vulnerable function by recording the hypervisor image load address at boot time and directly modifies the function via the recorded physical address with an offset. Alternatively, the attacker may attempt to directly modify the deployed trampoline or use the double mapping technique to map the hypervisor to a writable virtual address. Since the hypervisor code is protected by TZC with its physical address, these attempts would fail.

Attack on Patch Memory Allocation. The memory of patches is allocated by the hypervisor. The attacker may interfere with the memory allocation function and make it allocate an invalid or duplicated virtual address. However, an invalid virtual address would

indicate a Denial-of-Service attack, which could be detected by the Patch Deployment Module and lead to a threat alert. The duplicated address can also be identified since the Patch Deployment Module verifies the address after the allocation. Moreover, as the allocated memory is then protected by GPT, the attacker can no longer access this memory allocation.

Attack on Patch Integrity. The attacker is not able to tamper with the patch content, which is guarded by GPT even in the scenario of a multi-processor system. For a processor executing in the hypervisor, the GPT used in the processor must be the H.GPT, which prevents hypervisor-level access to the patch. For a processor running the patch, although the patch is assigned to the Normal World by the P.GPT, it remains in Root World from the view of other processors. Moreover, the processor has switched to the H.GPT before the patch calls or returns to other hypervisor functions, preventing the patch from manipulation while executing.

Attack with Hypervisor Data Pages. As the hypervisor data pages are marked as Normal World in the P.GPT, attackers can change them to executable pages in the page table for manipulation. However, direct function calls within the patch are not vulnerable since the page table entries for the `.text` section is verified. If the patch involves function pointers, the attacker could manipulate them to redirect execution. To prevent this attack, the Patch Generation Module inserts additional instructions before indirect branch instructions to confine the branch target within the patch or the hypervisor `.text` section.

Other Bypass Techniques. One potential approach to bypass the patch is to manipulate the page table to remap the virtual address of the vulnerable function or the entire kernel to a malicious copy at a different physical memory location. However, FortifyPatch defends against this manipulation by performing timely page table verification. While there may exist a small time window for TOCTTOU tricks, the chances of success are low. A single failed attempt triggers a threat alert by FortifyPatch, prompting a recommendation for a full update. Similarly, the attacker can remap the virtual address of the patch itself, but the Patch Protection Module can detect this before the execution enters the patch. Furthermore, the attacker could try to redirect the control flow by using well-crafted callback functions for kernel trace functionalities such as kprobe and ftrace. However, FortifyPatch restricts the usage of these functionalities with a whitelist and protects the callback function from further modification after signature verification. In summary, all of these attempts would fail under the protection of FortifyPatch.

7 EVALUATION

To evaluate the functionality of FortifyPatch, we implement a prototype on FVP Base RevC-2xAEMvA platform [8] and integrate FortifyPatch to TF-A version `arm_cca_v0.3` [10]. However, as FVP is not cycle-accurate [7], we evaluate the performance of FortifyPatch using a dedicated prototype discussed in §7.3.1. Our evaluation focused on answering the following research questions:

RQ1: What is the size of the Trusted Computing Base (TCB) introduced by FortifyPatch? (§7.1)

RQ2: Is FortifyPatch effective in patching CVEs? (§7.2)

RQ3: How is the performance of FortifyPatch? (§7.3)

RQ4: Would FortifyPatch affect the secure VMs supported by CCA? (§7.4)

Table 1: Patch trigger count while executing benchmarks. Each benchmark is executed 30 times with each patch and group, and the table shows the total trigger count of 30 runs.

(a) With a Single Patch.

Index	CVE	sysbench	Unixbench	lmbench
1	CVE-2014-0196	1.50×10^2	5.70×10^2	7.20×10^2
2	CVE-2016-0728	0	18	2
3	CVE-2016-7916	0	2	0
4	CVE-2017-17052	1.03×10^3	4.91×10^7	1.03×10^5
5	CVE-2018-1095	98	2	0
6	CVE-2018-10087	1.62×10^3	3.97×10^7	1.28×10^5
7	CVE-2018-13405	6.87×10^2	6.68×10^6	3.48×10^6
8	CVE-2019-9213	8.28×10^2	2.23×10^7	2.97×10^4
9	CVE-2020-13974	2.41×10^5	2.52×10^6	6.07×10^5
10	CVE-2022-2978	1.27×10^4	1.17×10^7	3.02×10^6

(b) With Multiple Patches.

Group	Patch Indices	sysbench	Unixbench	lmbench
1	2, 8	8.46×10^2	2.23×10^7	2.97×10^4
2	2, 4, 8, 10	1.38×10^4	8.16×10^7	3.30×10^6
3	1, 2, 4, 7, 8, 10	1.61×10^4	8.68×10^7	5.23×10^6
4	1, 2, 3, 4, 7, 8, 9, 10	2.58×10^5	8.93×10^7	5.79×10^6
5	all	2.59×10^5	1.18×10^8	5.99×10^6

RQ5: How does FortifyPatch compare to state-of-the-art kernel live patching systems? (§7.5)

7.1 RQ1: TCB Introduced by FortifyPatch

The Patch Generation Module is deployed in a local machine and consists of 661 Line-of-Code (LoC). Since FortifyPatch works with the hypervisor, we do not consider the Patch Generation Module to be part of the TCB of FortifyPatch. The other components of FortifyPatch are located inside the TF-A, we only increased about 1.3K LoC to implement FortifyPatch while the code size of TF-A used in our prototype includes approximately 310K LoC. Specifically, the Patch Deployment Module, Patch Protection Module, and Data Proxy Module include 312, 350, and 696 LoC, respectively.

7.2 RQ2: Effectiveness of FortifyPatch

To verify that FortifyPatch is capable of live patching Linux-based hypervisor, we randomly select 2000 CVEs submitted from 2013 to 2023 in the mainline Linux kernel. The corresponding patches are obtained from the National Vulnerability Database (NVD) website [46], resulting in a total of 1,385 identified patches. However, most of these patches are not available for our prototype since FVP is based on 64-bit Arm architecture and only simulates limited hardware. Specifically, 122 patches are not designed for 64-bit Arm architecture, and 862 patches are related to the device drivers or functionalities that are not supported in FVP and TF-A. Therefore, only 401 patches can potentially work with our prototype. We manually verify that 327 of them can be deployed by FortifyPatch. In summary, if the developed patch matches the deployment platform, FortifyPatch can deploy approximately 81.5% of the CVE patches. The reasons why other patches could not be successfully patched are summarized below.

Table 2: Patch Deployment Time.

CVE	Patch Size (in Byte)	Downtime (in μs)			
		Copy	GPT	Trampoline	Total
CVE-2014-0196	1,656	275.03	0.36	1.25	276.64
CVE-2016-0728	668	119.20	0.26	1.22	120.68
CVE-2016-7916	744	126.87	0.26	1.06	128.19
CVE-2017-17052	488	90.88	0.34	1.21	92.43
CVE-2018-1095	332	57.08	0.30	1.07	58.45
CVE-2018-10087	528	92.67	0.29	1.13	94.08
CVE-2018-13405	280	55.34	0.29	1.15	56.78
CVE-2019-9213	960	165.39	0.31	1.22	166.92
CVE-2020-13974	552	104.92	0.42	1.51	106.85
CVE-2022-2978	576	103.68	0.31	1.26	105.25

Compile-time Expanding Semantics. Macros, inline functions, and data structures are expanded at compile time, and changes to them affect all functions that refer to them. In theory, updating all these functions could achieve live patching. However, the naive approach could lead to practical issues. We identified 18 patches involving changes to macros and inline functions, while 44 patches are related to changes in data structures.

Makefile and Kconfig Files. Changes to Makefile and Kconfig files typically come with a large-scale code change, which makes the impact of the patch too large for a live patching system. We identify 9 patches in this category.

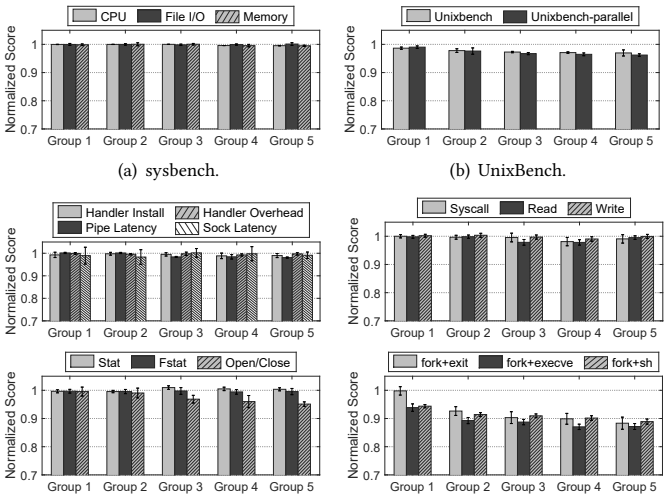
`init` Functions. The `init` functions in the kernel are placed in the `.init` section. These functions are invoked only once during boot time and then released in kernel memory free up. We identify 3 patches in this category.

In summary, if the patch matches the deployment platform, FortifyPatch can deploy approximately 81.5% of the CVE patches.

7.3 RQ3: Performance of FortifyPatch

7.3.1 Experiment Setup. As no real-world device with Arm CCA support is available at this moment, we developed a performance prototype using Raspberry PI 3B+ [61] with CPU frequency fixed at 1.4GHz to simulate FortifyPatch on the cycle-accurate platform. Since the GPT works similarly with an additional address translation layer in MMU, the Stage-2 translation is utilized to simulate the functionality and performance impact of GPCs. Specifically, we assume a Linux kernel running in EL1 is a hypervisor and transplant FortifyPatch from Root World to EL2. The Stage-2 translation table is used to substitute the GPT in the Patch Protection Module and also provide kernel code protection since the Raspberry PI 3B+ lacks support for TZC. To emulate the PMU overflow interrupt, we instrument the kernel to execute a `hvc` instruction upon receiving a `svc` exception or writing to the `TTBR` register. Other modules are implemented similarly to that in the FVP-based prototype. Although the Raspberry PI does not fully support the hardware features required by FortifyPatch, we consider it does not affect the performance evaluation heavily since most performance overhead in FortifyPatch is introduced by the switching of exception level and executing instructions not related to unsupported features.

We select *sysbench* [45], *UnixBench* [51], and *lmbench* [52] as the performance benchmark, as they are widely used to measure the performance of CPU computation and intensive operations [28, 43, 68, 89]. The configurations of these benchmarks are left as default. Since the performance overhead of FortifyPatch is highly related to the patch trigger count during the execution, we choose 10 different CVE patches and make the trigger count cover different orders of magnitude. The selected CVEs are listed in the `CVE` column of Table 1(a), and 1 patch involves data-related changes.



(a) sysbench. (b) UnixBench.

(c) lmbench.

Figure 5: Performance Evaluated with Multiple Patches.

7.3.2 Downtime for Patch Deployment. Since the downtime of a server is an important metric for a live patching system, we measure the downtime required to deploy the selected CVE patches. Table 2 shows that it takes only 56.78μs for FortifyPatch to deploy a 280-byte patch, which involves changes for a function of 18 LoC. For a patch as large as 1,656 bytes that changes a function of 88 LoC, the downtime is as small as 276.64μs. The table also shows that the downtime is primarily due to copying the patch to the reserved memory region, while the time consumption for GPT configuration and trampoline deployment is negligible. Therefore, a larger patch would result in a longer downtime. In summary, FortifyPatch only introduces negligible microseconds of interruption, making it well-suited for live patching scenarios.

7.3.3 Evaluation with Benchmarks. We use FortifyPatch to deploy the patches discussed in §7.3.1 and execute the benchmarks with the patches. Each experiment is repeated for 30 times. Moreover, all these benchmarks provide scores for various evaluation metrics. We normalize the score of each metric without any patches as 1 and show the relative score after patches are deployed. For simplicity, we only present the overhead while deploying multiple patches, and the overhead caused by a single patch follows similar trends.

To learn the performance of FortifyPatch when multiple patches are deployed in the hypervisor, we randomly group the patches in §7.3.1 and measure the performance in our experiments. The group information and corresponding patch trigger times incurred in the experiments are listed in Table 1(b). Figure 5 summarizes the evaluation result with these patch groups. The `lmbench` contains a group of metrics and we only show the metrics with the

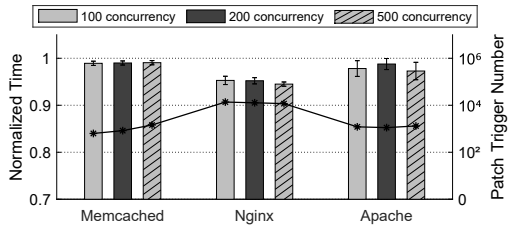


Figure 6: Performance Evaluated with Real-World Projects.

most significant downgrade in Figure 5(c). According to the figure, the performance impact of FortifyPatch is negligible in most metrics. However, since patches for both CVE-2017-17052 and CVE-2019-9213 patch the functions involved in the `fork` process, the `fork`-related experiments in `lmbench` introduces an overhead at approximately 12.9%. These patches do not affect other metrics, and the average performance slowdown caused by FortifyPatch is 0.98%. Note that the patch trigger count in Table 1 does not directly reflect the frequency of patch triggering since the time consumption of the benchmarks is different. Specifically, a single run of `sysbench`, `UnixBench`, and `lmbench` costs about 5.5, 56.2, and 14.5 minutes, respectively.

7.3.4 Evaluation with Real-World Applications. To learn the impact on real-world workloads, we use `Memcached` [21], `Nginx` [75], and `Apache` [26] to simulate the memory-intensive and I/O-intensive operations. With all patches in Table 2 deployed in the hypervisor, we run each application with various concurrency levels, and each experiment is repeated 30 times. Figure 6 shows the performance overhead introduced by FortifyPatch while working with these applications. We find that the concurrency level does not significantly affect the performance, likely because the execution switch introduced by FortifyPatch account for only a small percentage of all operations of the application. The performance overhead is approximately 1.1%, 5.5%, and 2.7% for `Memcached`, `Nginx`, and `Apache`, respectively. FortifyPatch introduces heavier overhead to `Nginx` due to more frequent patch triggering. The curve in Figure 6 illustrates that the patch is triggered more than 10,000 times in the 30-second execution of `Nginx`. Overall, we consider the performance impact on real-world applications to be small.

7.4 RQ4: Compatibility of FortifyPatch

To verify that FortifyPatch does not affect the ability of hypervisors to maintain secure VMs, we integrate FortifyPatch into an industry-level confidential computing prototype, Samsung Islet [67]. Islet is built on FVP with CCA support to provide confidential computing in the Realm VMs. For our experiment, we use the Islet versioned with commit `bdde32c`, which includes TF-A-based firmware, Linux-based lightweight hypervisor, Rust-based RMM, and scripts to launch Linux-based realms. We integrate FortifyPatch into the firmware of Islet and launch a realm with the provided scripts. To simulate the workload inside the realms, we make the realm run the provided sample payload named `sdk-example` endlessly. Next, we deploy the patches listed in Table 2 into the hypervisor and manually verify whether the patch and the workload inside the realm work as expected. The result illustrates that FortifyPatch is able to work with Islet without interfering with each other.

7.5 RQ5: Comparing with Other Systems

We compare FortifyPatch with indicative live patching systems including KUP [38], RapidPatch [34], kpatch [33], and KShot [91]. The result is summarized in Table 3. "Secure Patching" refers to a manipulation-resistant patching process immune to kernel-level attacks. Regarding "Patch Level", kernel-level patching addresses most vulnerabilities, while instruction-level patching minimizes patch count. Function-level patching balances downtime and patching ability. "Patch Scope" describes a live patching system capable of effectively patching instructions and global data changes in a practical way. The results in the table demonstrate that FortifyPatch possesses a broader patching capability while maintaining a comparable level of performance overhead.

Functionality Comparison. While all live patching systems offer basic patching functionality, only FortifyPatch is capable of patching global variables without practical issues. In addition, both FortifyPatch and KShot [91] provide secure patching, as the patching process cannot be controlled by privileged attackers. To the best of our knowledge, FortifyPatch is the only live patching system that can prevent attackers from tampering with deployed patches.

Performance Comparison. It should be noted that the systems used for comparison are designed for different architectures and evaluated on various hardware platforms, which makes it difficult to make a fair comparison. Nevertheless, based on the results, it can be observed that deploying a 1KB patch using FortifyPatch requires moderate memory and system downtime, making it a practical solution for patch deployment. Moreover, FortifyPatch introduces a low overhead to protect the patches, which puts it in a competitive position compared to state-of-the-art systems.

8 RELATED WORK

Kernel Live Patching. kpatch [33], Ksplice [60], kGraft [74], and Linux Livepatch [41] works in a similar way and uses function-level live patching to redirect the execution of vulnerable functions to updated ones. KARMA [15] focuses on the Linux kernel in Android and adaptively patches the kernel at multiple levels. RapidPatch [34] relies on eBPF virtual machines to patch the firmware of resource-constrained embedded devices. Although these solutions are capable of deploying patches at runtime, they fail to guarantee the trustworthiness of the patching procedure since they rely on kernel functionalities that may have been manipulated due to the exposed vulnerability. KShot [91] aims to protect the patching process using Trusted Execution Environments (TEE), such as the x86 System Management Model (SMM) [3] and Intel Software Guard eXtension (SGX) [16]. However, KShot only protects the patch process and lacks post-deployment protection.

Live Updating. KUP [38] utilizes a checkpoint-and-restart mechanism to replace the Linux kernel at runtime and restore application states after the update. VM-PHU [65] records the memory and device state of all running guest VMs before a live update and uses kernel soft reboot to activate the updated version of the hypervisor. However, this category of live update requires additional time and resources to save and restore the runtime states.

Automatic Program Repair. VulRepair [27] and TransRepair [49] are automated vulnerability repair techniques based on artificial intelligence. RAP-Gen [79] utilizes an external codebase to generate

Table 3: Comparison FortifyPatch with Other Systems. The size of the patch used is about 1KB.

System Name	Patch Level	Secure Patching	Tamper Resistance	Patch Scope		Downtime	Memory	Overhead
				Instruction	Global Data			
KUP [38]	kernel	✗	✗	✓	✗	2.4s/kernel	56GB	/
RapidPatch [34]	instruction	✗	✗	✓	✗	7.5us/23LoC	18KB	2.2%∼9.1%
kpatch [33]	function	✗	✗	✓	✗	45.6ms/patch	20MB	/
KShot [91]	function	✓	✗	✓	✗	50μs/patch	18MB	3%
FortifyPatch	function	✓	✓	✓	✓	166.92μs/patch	16MB	0.1%∼6.4%

reliable patches, while TransplantFix [86] employs graph differencing for patch generation. Shariffdeen *et al.* [71] propose a patch backporting tool to automatically transfer patches from the mainline Linux into older versions. However, these solutions only focus on generating source code-level patches and fail to consider the protection of patches.

CCA-based Systems. Shelter [90] extends Arm CCA to provide a user-level Trusted Execution Environment (TEE). By assigning the application a dedicated GPT that prevents access from the OS kernel, Shelter is able to protect the application from OS-level manipulation. However, the scheme of Shelter cannot be directly used in live patching due to high-performance overhead. ACAI [73] enables secure communication between confidential VMs and accelerators via the security guarantee provided by Arm CCA.

9 LIMITATION AND FUTURE WORK

Patch Scope. Similar to other difference-based live patching mechanisms [33, 60, 74, 91], handling the changes in macros, inline functions, and data structures would lead to practical issues. If a large number of functions are affected by the changed objects, we suggest using the live updating approach as a supplement to refresh the hypervisor. Moreover, shadow variables [42] are presented to be a potential solution for adding variables to data structures.

Reducing Data Access Traps. The data access trap in FortifyPatch helps to solve the practical issue caused by patching global variables. However, unnecessary traps are triggered when the hypervisor accesses other variables on the same memory page. This is due to the minimum granularity of a PAS controlled by GPT being a 4K memory page. The unnecessary trap may be reduced by a more fine-grained protection mechanism (e.g., Arm Memory Tagging Extension [6]), which we leave as future work.

Consistency Issue. FortifyPatch may cause consistency issues if other tasks are using the vulnerable function or data during the patching process. The Linux Livepatch [41] mechanism addresses this by employing a per-task consistency model. While the design of FortifyPatch is compatible with this consistency model, integrating it would require additional engineering efforts. As the primary focus of this work is patch tamper resistance, addressing the consistency issue is left for future work.

TOCTTOU Problem. FortifyPatch employs PMU interrupt to initiate the verification of the page table entries for kernel `.text` section. This leaves a small time window for TOCTTOU attacks. However, we consider the success rate of the attack to be low and a single failed attempt would lead to threat alert.

CCA Security. As a newly designed solution for confidential computing, there might be undisclosed vulnerabilities within the CCA design. Existing attacks [13, 50, 58, 62, 70, 78] to Arm TEEs may also be used to compromise the security of CCA. However, we consider this to be another research topic and out of the scope of this work.

Performance Impact with LLVM KCFI. While LLVM KCFI is implemented in the kernel, we consider the overhead of FortifyPatch might slightly increase. Since LLVM KCFI requires loading a function identifier from `.text` section before branching to the function, it would cause one additional GPT switch for each call to the patch function. However, our experiments show that on average 11.7 GPT switches are required for a patch function call without LLVM KCFI, the additional one switch would only affect the performance slightly.

Generalizability. FortifyPatch relies on a fine-grained memory management mechanism in a high-privilege context. Once such a mechanism exists in other ISAs, FortifyPatch is applicable. For example, the ongoing efforts in the RISC-V community to implement such a mechanism [66] offer the potential to deploy FortifyPatch to RISC-V architecture.

10 CONCLUSION

In this paper, we present FortifyPatch, a tamper-resistant live patching system designed to persistently patch Linux-based hypervisors at runtime. FortifyPatch leverages a group of hardware features to protect deployed patches from being manipulated or bypassed. To address the practical issue associated with patching global variables in live patching, FortifyPatch employs well-designed traps to minimize the number of affected instructions. We implement FortifyPatch and evaluate its functionality and performance on Arm FVP and Raspberry PI 3B+, respectively. The evaluation indicates that FortifyPatch is capable of deploying 81.5% of CVE patches. The performance evaluation shows that FortifyPatch protects deployed patches with 0.98% and 3.1% overhead on average across indicative benchmarks and real-world applications, respectively.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62102175, No.62372218, No.62202150, No.62302050, No.U20A20174, No.U22A2030, and Excellent Youth Fund, Distinguished Youth Fund Project of Hunan Province, China under Grant No. 2022JJ10018, Shenzhen Science and Technology Program under Grant No.SGDX20201103095408029, Fundamental Research Funds for the Central Universities, National Key R&D Projects under Grant No.2022YFB3103500, Technology Projects of Hunan Province under Grant No.2015TP1004, and Science and Technology Key Projects of Changsha City under Grant No.kh2103003.

DATA AVAILABILITY STATEMENT

We have made all the artifacts publicly available at https://doi.org/10.5281/zenodo.12657258 [88] for replication.

REFERENCES

[1] Alibaba. 2023. Alibaba cloud security white paper. https://alicloud-common.oss-ap-southeast-1.aliyuncs.com/2021/Whitepaper/Alibaba%20Cloud%20Security%20Whitepaper%20-%20International%20Edition%20V2.1%20%282021%29.pdf.

[2] Amazon. 2023. Regulation systems compliance and integrity considerations for the AWS cloud. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html.

[3] AMD. 2023. AMD64 architecture programmer's manual volume 2: system programming. https://www.amd.com/system/files/TechDocs/24593.pdf.

[4] Arm. 2015. Arm CoreLink TZC-400 TrustZone address space controller technical reference manual. https://developer.arm.com/documentation/100325/latest/.

[5] Arm. 2023. Arm confidential compute architecture. https://www.arm.com/en/architecture/security-features/arm-confidential-compute-architecture.

[6] Arm. 2023. Armv8.5-A memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[7] Arm. 2023. Fast models reference guide. https://developer.arm.com/documentation/100964/1121/About-the-models/Fast-Models-accuracy/Timing-accuracy-of-Fast-Models.

[8] Arm. 2023. Fixed virtual platforms. https://www.arm.com/products/development-tools/simulation/fixed-virtual-platforms.

[9] Arm. 2023. Realm management extension. https://developer.arm.com/documentation/den0129/latest/.

[10] Arm. 2023. Trusted firmware-a. https://github.com/ARM-software/arm-trusted-firmware.

[11] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the Arm TrustZone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*. https://doi.org/10.1145/2660267.2660350

[12] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight secure kernel-level execution environment for Arm.. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*. https://doi.org/10.14722/ndss.2016.23009

[13] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. Sok: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. https://doi.org/10.1109/SP40000.2020.00061

[14] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. Instaguard: Instantly deployable hot-patches for vulnerable system programs on Android. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23124

[15] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (Security'17)*.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[17] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*. https://doi.org/10.1109/SP.2014.26

[18] National Vulnerability Database. 2020. CVE-2020-13974 detail. https://nvd.nist.gov/vuln/detail/CVE-2020-13974.

[19] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical mitigation of data-only attacks against page tables.. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*. https://doi.org/10.14722/ndss.2017.23421

[20] Developer.com. 2017. 90% of the public cloud runs on Linux. https://www.developer.com/news/90-of-the-public-cloud-runs-on-linux/.

[21] Dormando. 2023. Memcached. https://memcached.org/.

[22] Eric Dumazet. 2016. Patch for CVE-2016-5696. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=75ff39ccc1bd5d3c455b6822ab09e533c551f758.

[23] Eric Dumazet. 2019. Patch for CVE-2019-11478. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=f070ef2ac66716357066b683fb0baf55f8191a2e.

[24] The eBPF community. 2023. eBPF. https://ebpf.io/.

[25] RISC-V Foundation. 2017. The RISC-V instruction set manual volume II: Privileged architecture. https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[26] The Apache Software Foundation. 2023. Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[27] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. https://doi.org/10.1145/3540250.3549098

[28] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*.

[29] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*. https://doi.org/10.1109/EuroSP.2016.24

[30] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST'14)*. https://doi.org/10.48550/arXiv.1410.7747

[31] Google. 2022. Google security overview. https://cloud.google.com/docs/security/overview/whitepaper.

[32] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. Trustshadow: Secure execution of unmodified applications with Arm TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*. https://doi.org/10.1145/3081333.3081349

[33] Red Hat. 2023. kpatch: dynamic kernel patching. https://github.com/dynup/kpatch.

[34] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware hotpatching for real-time embedded devices. In *Proceedings of the 31st USENIX Security Symposium (Security'22)*.

[35] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. 2021. ACHyb: a hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. https://doi.org/10.1145/3468264.3468627

[36] Yuri Ilyin. 2016. Hothacking: an obscure Windows feature as an APT weapon. https://www.kaspersky.com/blog/hotpatching-apt/15142/.

[37] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting concise bug-fixing patches from human-written patches in version control systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. https://doi.org/10.1109/ICSE43902.2021.00069

[38] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. 2016. Instant OS updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC'16)*.

[39] Kaspersky. 2023. What is an advanced persistent threat? https://www.kaspersky.com/resource-center/definitions/advanced-persistent-threats.

[40] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. 2023. Kernel Probes. https://docs.kernel.org/trace/kprobes.html.

[41] The kernel development community. 2023. Livepatch. https://docs.kernel.org/livepatch/livepatch.html.

[42] The kernel development community. 2023. Shadow Variables. https://docs.kernel.org/livepatch/shadow-vars.html.

[43] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. 2021. Z-Journal: Scalable per-core journaling.. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC'21)*.

[44] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid fuzzing on the Linux kernel. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*. https://doi.org/10.14722/ndss.2020.24018

[45] Alexey Kopytov. 2023. sysbench. https://github.com/akopytov/sysbench.

[46] Information Technology Laboratory. 2023. National vulnerability database. https://nvd.nist.gov/.

[47] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. https://doi.org/10.1109/ICSE.2019.00064

[48] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 25th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. https://doi.org/10.1145/3106237.3106309

[49] Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022. Transrepair: Context-aware program repair for compilation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. https://doi.org/10.1145/3551349.3560422

[50] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. Armageddon: Cache attacks on mobile devices.. In *Proceedings of the 25th USENIX Security Symposium (Security'16)*.

[51] Kelly Lucas. 2023. byte-unixbench. https://github.com/kdlucas/byte-unixbench.

[52] Larry McVoy. 2023. lmbench. https://lmbench.sourceforge.net/.

[53] Microsoft. 2022. Microsoft operational security practices. https://www.microsoft.com/en-us/securityengineering/osa/practices.

[54] MITRE. 2021. Impair defenses: Downgrade attack. https://attack.mitre.org/techniques/T1562/010/.

[55] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. LXDs: Towards isolation of kernel subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*.

[56] Liam O Murchu Nicolas Falliere and Eric Chienh. 2010. Symantec Security Response. https://www.wired.com/images_blogs/threatlevel/2010/11/w32_stuxnet_dossier.pdf.

[57] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of embedded real-time applications. In *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS'21)*. https://doi.org/10.14722/ndss.2021.24159

[58] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the security of arm debugging features. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. https://doi.org/10.1109/SP.2019.00061

[59] Team Nuggets. 2018. Why Linux runs 90 percent of the public cloud workload? https://www.cbtnuggets.com/blog/certifications/open-source/why-linux-runs-90-percent-of-the-public-cloud-workload.

[60] Oracle. 2023. Ksplice. https://ksplice.oracle.com/.

[61] Raspberry Pi. 2023. Raspberry Pi 3 model b+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/.

[62] Sandro Pinto and Nuno Santos. 2019. Demystifying arm TrustZone: A comprehensive survey. *ACM Computing Surveys (CSUR'19)* (2019). https://doi.org/10.1145/3291047

[63] Gartner Research. 2019. Ensure cost balances with risk in high-availability data centers. https://www.gartner.com/en/documents/3906266.

[64] Steven Rostedt. 2017. Function tracer. https://docs.kernel.org/trace/ftrace.html.

[65] Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. 2021. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. https://doi.org/10.1145/3447786.3456232

[66] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. 2023. CoVE: Towards Confidential Computing on RISC-V Platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. https://doi.org/10.1145/3587135.3592168

[67] Samsung. 2023. Islet. https://github.com/Samsung/islet.

[68] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing syscalls for PKU-based memory isolation systems. In *Proceedings of the 31st USENIX Security Symposium (Security'22)*.

[69] Microsoft Security. 2020. System management mode deep dive: How SMM isolation hardens the platform. https://www.microsoft.com/en-us/security/blog/2020/11/12/system-management-mode-deep-dive-how-smm-isolation-hardens-the-platform/.

[70] Alon Shakevsky, Eyal Ronen, and Avishai Wool. 2022. Trust dies in darkness: Shedding light on Samsung's TrustZone keymaster design. In *Proceedings of the 31st USENIX Security Symposium (Security'22)*.

[71] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. https://doi.org/10.1145/3460319.3464821

[72] Manjeet Singh. 2022. Role of Linux in cloud computing. https://www.linkedin.com/pulse/role-linux-cloud-computing-manjeet-singh.

[73] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. 2024. ACAI: Extending Arm confidential computing architecture protection from CPUs to accelerators. *To Appear in the 32nd USENIX Security Symposium (Security'23)* (2024).

[74] SUSE. 2023. Live patching the Linux kernel using kGraft. https://documentation.suse.com/sles/12-SP4/html/SLES-kgraft/index.html.

[75] Igor Sysoev. 2023. Nginx. https://github.com/nginx/nginx.

[76] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. https://doi.org/10.1145/3324884.3416532

[77] Dmitry Torokhov. 2020. Patch for CVE-2020-13974. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b86dab054059b970111b5516ae548efaae5b3aae.

[78] Jie Wang, Kun Sun, Lingguang Lei, Shengye Wan, Yuewu Wang, and Jiwu Jing. 2020. Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*. https://doi.org/10.1145/3372297.3417886

[79] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. https://doi.org/10.1145/3611643.3616256

[80] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10)*. https://doi.org/10.1109/SP.2010.30

[81] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. https://doi.org/10.1145/3180155.3180233

[82] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. Kepler: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security'19)*.

[83] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)*. https://doi.org/10.1109/ICSE48619.2023.00129

[84] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 36th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'27)*. https://doi.org/10.1145/3092703.3092718

[85] Zenon Xiu. 2023. Does TZASC still work in Armv9 when RME is implemented? https://community.arm.com/support-forums/f/architectures-and-processors-forum/52069/does-tzasc-still-work-in-armv9-when-rme-is-implemented.

[86] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. https://doi.org/10.1145/3551349.3556893

[87] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2023. A Large-Scale Empirical Review of Patch Correctness Checking Approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. https://doi.org/10.1145/3611643.3616331

[88] Zhenyu Ye. 2024. Artifact. https://doi.org/10.5281/zenodo.12657258.

[89] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-kernel control-flow integrity on commodity OSe using Arm pointer authentication. In *Proceedings of the 31st USENIX Security Symposium (Security'22)*.

[90] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. Shelter: Extending Arm CCA with isolation in user space. In *Proceedings of the 32nd USENIX Security Symposium (Security'23)*.

[91] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. 2020. KShot: Live kernel patching with SMM and SGX. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'20)*. https://doi.org/10.1109/DSN48063.2020.00021

Received 16-DEC-2023; accepted 2024-03-02