KShot: Live Kernel Patching with SMM and SGX

Lei Zhou*†, Fengwei Zhang*, Jinghui Liao‡, Zhengyu Ning*, Jidong Xiao§
Kevin Leach¶, Westley Weimer¶ and Guojun Wang‖
*Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China,
{zhoul2019,zhangfw,ningzy2019}@sustech.edu.cn
†School of Computer Science and Engineering, Central South University, Changsha, China
‡Department of Computer Science, Wayne State University, Detroit, USA, jinghui@wayne.edu
§Department of Computer Science, Boise State University, Boise, USA, jidongxiao@boisestate.edu
¶Department of Computer Science and Engineering, University of Michigan, Ann Arbor, USA, {kjleach,weimerw}@umich.edu
‖School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou, China, csgjwang@gzhu.edu.cn

***Abstract*—Live kernel patching is an increasingly common trend in operating system distributions, enabling dynamic updates to include new features or to fix vulnerabilities without having to reboot the system. Patching the kernel at runtime lowers downtime and reduces the loss of useful state from running applications. However, existing kernel live patching techniques (1) rely on specific support from the target operating system, and (2) admit patch failures resulting from kernel faults. We present KSHOT, a kernel live patching mechanism based on x86 SMM and Intel SGX that focuses on patching Linux kernel security vulnerabilities. Our patching processes are protected by hardware-assisted Trusted Execution Environments. We demonstrate that our technique can successfully patch vulnerable kernel functions at the binary-level without support from the underlying OS and regardless of whether the kernel patching mechanism is compromised. We demonstrate the applicability of KSHOT by successfully patching 30 critical indicative kernel vulnerabilities.**

I. INTRODUCTION

The growing complexity and heterogeneity of software has led to a concomitant increase in the pressure to apply patches and updates, including to the operating system itself [1]. Frequently, users that choose to patch their kernels may incur downtime when the patch requires restarting the system. This unavoidable disruption impacts both enterprise and end users. For example, in systems that are performing complex scientific computations or financial transactions, users are unlikely to reboot a system [2], [3]. According to Gartner [4], the average cost of IT downtime is $\$5,600$ per minute. Businesses downtime can reach $\$300,000$ per hour, on average. Even for general end users, unplanned downtime interrupts running applications risks the loss of unsaved data. As a result, enterprises and users often delay applying patches to their operating systems, leading to increased risks to their computing resources [1].

Since patches are important to fixing vulnerabilities and adding software features, many prior approaches propose live patching mechanisms that reduce or avoid system reboots or the loss of application or OS state. Early mechanisms focused on live updating applications (e.g., POLUS [5]), but kernel vulnerabilities also merit patching. Organizations often use rolling upgrades [3], [6], in which patches are designed to affect small subsystems that minimize unplanned whole-system downtime, to update and patch whole server systems. However, rolling upgrades do not altogether obviate the need to restart software or reboot systems; instead, dynamic hot patching (live patching) approaches [7]–[9] aim to apply patches to running software without having to restart it.

Several kernel-level live patching tools have been designed previously, including kpatch [10], kGraft [11], Ksplice [12], and the Canonical Livepatch Service [13]. For example, kpatch leverages OS-provided infrastructures such as `ftrace` to trace a target function, `clone` and `fork` to hook the entry instruction in that target function, and then trampolines to a patched version of that target function. Moreover, it can use `procfs` and `ptrace` system calls to checkpoint and restore the state of running applications. In addition, all those approaches need to modify the existing kernel code and trust the operating system. In a similar vein, KUP [8] replaces the whole kernel at runtime while retaining state from running applications. However, KUP incurs significant runtime and resource overhead (e.g., more than 30GB of memory space) to support application checkpointing [14], even for very small kernel patches.

Existing patching techniques must trust the OS kernel or cooperative patching applications to deploy patches. However, patching implementations can suffer from numerous bugs [15], which may cause patching failures or interruptions. Moreover, a patch may become compromised if the OS or patching mechanism becomes compromised. For example, an internal OS update can be hijacked [16]–[18] to download and install malicious patches. Such attacks download additional malicious applications while retaining kernel functionality. Further, even after live patching applies kernel patches, kernel attacks may be able to revert the software to a vulnerable version [19]. Such situations are more likely to happen in remote or cloud computing environments [20], [21], where users have less control over a remote computer's patching operations.

Thus, there is a need to improve the dependability of live patching techniques.

The work was done while Lei Zhou visiting at COMPASS lab.
Fengwei Zhang is the corresponding author.

To summarize, live kernel patching faces three challenges:

1) **Downtime**. Traditional kernel patching methods require downtime, either from unplanned reboots or from stopping applications to checkpoint states.
2) **Overhead**. Live kernel patching techniques often incur non-trivial CPU and memory overhead to apply patches and restore previously-checkpointed state.
3) **Trust**. Live patching software depends on the correctness of the underlying OS, which may suffer from bugs [22] or security vulnerabilities. If the OS-level patching mechanism becomes compromised, then patches applied by that mechanism cannot be trusted.

In this paper, we present KSHOT, a live kernel patching technique that uses Intel Software Guard eXtensions (SGX) and System Management Mode (SMM) to effectively, efficiently, and reliably patch running, untrusted kernels. We summarize our contributions as follows:

- We develop a reliable architecture for live kernel patching. We leverage Trusted Execution Environments (TEEs) implemented with SGX and SMM to prepare and deploy kernel patches that do not require trusting the kernel patching mechanism.
- We use SMM (i.e., hardware support) to naturally store the runtime state of the target host, which reduces external storage overhead and improves live patching performance. Employing this hardware-assisted mechanism supports faster restoration without requiring external checkpoint and restore solutions (e.g., Criu [14]). Moreover, we adopt an SMM-based kernel protection approach for secure live patching.
- We use SGX as a trusted environment for patch preparation to provide adequate runtime patch performance. Furthermore, patching in an SGX enclave precludes adversarial tampering, improving patching reliability.
- We evaluate the effectiveness and efficiency of KSHOT by providing an in-depth analysis on a suite of indicative kernel vulnerabilities. We demonstrate that our approach incurs little overhead while providing trustworthy live kernel patches that mitigate known kernel exploits.

II. BACKGROUND

In this section, we first introduce existing live patching techniques. We then provide an overview of x86 System Management Mode and Intel Secure Guard eXtensions, which we use as a trusted base to implement our approach.

A. Kernel Live Patching

Live patching (also known as *hot patching*) is a method for dynamically updating software, effectively reducing the downtime and inconvenience often associated with software upgrades [7], [10], [23]–[26]. We focus on the particular domain of Kernel Live Patching (KLP), which updates the operating system to address vulnerabilities or bugs without having to restart. Figure 1 illustrates common KLP methods, which can update kernel software at three levels of abstraction: function replacement, instruction hooking and jumping, and kernel switching. In general, these methods can replace single instructions, vulnerable functions, or even the whole kernel with a patched one to repair bugs or eliminate vulnerabilities. Solutions in this area include industry-deployed mechanisms like Ksplice [12] and kpatch [10] as well as academia-proposed solutions like KUP [8] and KARMA [9].

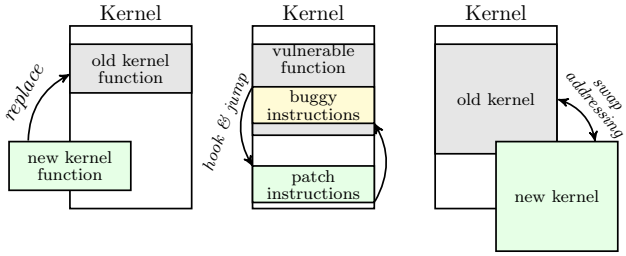

Fig. 1: Overview of live patching approaches—function-, instruction-, and kernel-level. In function-level, entire kernel functions are replaced with new ones by copying bytes into memory. In instruction-level, single buggy instructions are replaced with trampolines to new instructions. In kernel-level, the entire kernel image is replaced with a new binary image by switching page table entries so that kernel addresses correspond to a new location in physical memory that contain the revised image.

However, current KLP techniques extend trust to the kernel itself to correctly deploy patches. If the kernel becomes compromised, then any subsequent patches deployed by that kernel are not trustworthy, potentially leading to additional malicious activities [1]. In our work, we implement a trustworthy KLP mechanism by leveraging TEEs that enable live kernel patching even when the underlying kernel patching mechanism is compromised.

B. System Management Mode

System Management Mode (SMM) is a highly-privileged CPU execution mode present in all current x86 machines since the 80386. It is used to handle system-wide functionality such as power management, system hardware control, or OEM-specific code. SMM is used by the system firmware but not by applications or normal system software. The code and data used in SMM are stored in a hardware-protected memory region named System Management RAM (SMRAM), which is inaccessible from the normal OS (i.e., can only be accessed by SMM). SMM code is executed by the CPU upon receiving a *System Management Interrupt* (SMI), causing the CPU to switch modes from (typically) Protected Mode to SMM. The hardware automatically saves the CPU state in a dedicated region in SMRAM. Upon completing the execution of SMM code by the `RSM` instruction, the CPU's state is restored, resuming execution in Protected Mode. Moreover, SMM is able to access physical memory with a higher privilege, allowing it to read or modify kernel code and data structures in kernel memory segments.

C. Software Guard eXtensions

Software Guard eXtensions (SGX) [27] is a TEE technology proposed by Intel which allows a trusted application to run in

userspace, even if the OS kernel is compromised. SGX protects selected code and data from disclosure or modification by the OS. Developers can partition applications into processor-hardened *enclaves*, or protected areas of execution in memory, which increase security without having to extend trust beyond those enclaves. Enclaves are trusted execution environments provided by SGX. The enclave code and data reside in a region of protected physical memory called the Enclave Page Cache (EPC). The EPC is guarded by CPU access controls: non-enclave code cannot access enclave memory.

III. Threat Model and Assumptions

We propose an approach to provide reliable live patches to kernels with untrusted patching mechanisms.

We assume that kernel-based patching mechanisms can become compromised by internal weaknesses [15], [22] or external attacks [18]. For example, the vulnerability CVE-2016-5195, which exploits a race condition for privilege escalation within the kernel, can be used by attackers to install rootkits. Attackers can design such rootkits to interfere with the patching process and prevent memory-level bug repairs (e.g., by undoing changes to memory introduced by a live-patching system). Thus, we instead assume that the target OS supports SGX [28], [29] hardware. We further assume that the system is trusted during the boot process, and that System Management RAM (Section II-B) is locked by the system firmware so that an attacker cannot modify it (i.e., that the hardware is trusted to enforce access control). While SGX and SMM are potentially vulnerable to side-channel attacks like Spectre [30], Meltdown [31], Foreshadow-NG [32], SMBR [33], and SMM Reload [34], such vulnerabilities can be addressed by hardware vendors, and are not the subject of this paper. In brief, we trust the hardware and firmware, but not the software or operating system's patching mechanism. In addition, we assume that the source code of the patch is trusted.

Our proposed approach focuses on live patching vulnerabilities in existing kernel code, but we note that this capability is independent of the new kernel's correct handling of previously-tainted data. The detection and handling of tainted data left behind in memory or on the disk by an OS-compromising attacker is an orthogonal issue that may be handled by other techniques, such as cross-host taint tracking [35], remote witness servers that validate update effects [36], or SGX-based solutions to the state continuity problem [37], among others. Alternatively, the OS patch or update system might employ a mechanism such as type wrapping or transformation [38] to clean or migrate critical data. KSHOT is agnostic to the underlying patch being applied (see Section V-A) and thus supports such approaches. We also note that denial of service (DOS) attacks could prevent live patching systems from executing. However, this is not specific to our work (and, indeed, KSHOT can detect when DOS attacks occur). If DOS attacks occur, we assume that a system operator in the loop would elect to take a victim system offline for subsequent manual patching.

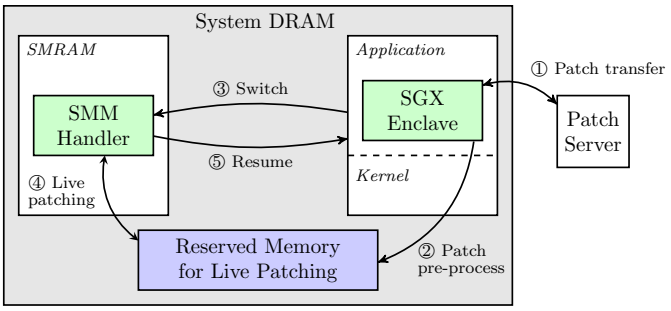

Fig. 2: High-level architecture of KSHOT. Our approach uses three secure entities: the Remote Patch Server, the SGX enclave in a helper application, and the SMM-based kernel patching environment. The annotations 1–5 trace the life cycle of trusted live patching. In (1), we transfer the patch; in (2), we pre-process the binary patch; in (3), we switch to SMM; in (4), we apply the patch at the binary level; and in (5), we resume the updated OS.

IV. System Architecture

KSHOT aims to provide a reliable and low-overhead live patching framework for untrusted kernels. KSHOT achieves these goals by using a novel combination of an SGX enclave within a helper application that securely downloads patched source code which is built and written to kernel memory by a custom SMM Handler. By construction, our approach benefits from very low storage overhead associated with application checkpointing, rapid deployment of patches with low latency, and the trustworthy application of patches even when the kernel's patching mechanism has been compromised.

First, we assume that an operator wants to update a vulnerable, buggy, or compromised kernel on a system (which we call the Target Machine). Next, we assume that developers have created an updated, fixed, or otherwise patched version of the kernel that the operator wants to apply to the Target Machine. Briefly, our approach is to leverage an SGX enclave in a helper application to download an updated binary kernel patch, then use the SMM Handler to pause the Target Machine's execution and apply the patch. This novel combination of system features allows us to deploy patches with low runtime overhead, low latency, and without having to trust the underlying OS to deploy the patch.

Figure 2 summarizes our approach. First, the Target OS information which is required for compiling compatible binary patches is gathered and sent to the remote Patch Server. Second, an SGX-based application fetches the binary patch from the remote Patch Server and collects required patching information (e.g., patch location addresses). The information is loaded into the reserved memory region to be processed by the SMM Handler code. Third, we remotely trigger [39] a patching command, and forced switch current host to SMM to execute the SMM Handler, which modifies the Target Machine's memory. Through a combination of hooking, adding redirection instructions in target functions, and locating the binary patch in a reserved memory location (see Section V),

the patch is applied so that the updated code will be executed on the next invocation once the SMM Handler completes.

A. KSHOT *Components*

There are three main components in our KSHOT architecture: the remote Patch Server, system-specific patch preprocessing in the SGX enclave, and SMM-based kernel patching.

Remote patch server: The remote Patch Server is an independent, trusted system that constructs and supplies trusted binary patches. That is, we assume that developers have already provided a fixed or updated binary kernel image that we seek to apply to the Target Machine. The Patch Server communicates with the target machine to obtain OS information, which is used to build a compatible binary kernel image, allowing for the creation of consistent binary patches.

SGX-based patch preparation: This component includes kernel information collection and binary patch preprocessing. These processes take place in an SGX enclave. The data transmitted between SGX and the Patch Server, as well as between SGX and SMM, are encrypted to protect patch code from malicious changes. Leveraging SGX for patch preprocessing provides several benefits: First, it reduces the SMM workload and thus the time during which the OS is paused to execute the SMM Handler. Second, it reduces the amount of software that must be developed in SMM (e.g., bespoke network drivers must be implemented to transfer data if all processing is handled in SMM). Finally, because of the large semantic gap between SMM and the host environment [40], it is more natural to gather kernel information from the software layer within an SGX-enabled helper application.

SMM-based kernel patching: This component includes patch decryption, patch function integrity checking, and binary patching. KSHOT promises consistency of kernel execution since the hardware automatically saves and restores architectural state (e.g., registers) while switching to SMM. This saves substantial time and resource overhead compared to software-based system state saving and restoration (i.e., checkpointing) in previous live patching approaches. In addition, if a kernel error occurs after patching [22], this component can undo the patch and rollback the system. While the patch operations are processed in SMM, the target OS is halted (which precludes simultaneous state changes). Because this activity is carried out with SMM support, even kernel-level attacks cannot compromise patching operations. In addition to a patching module, KSHOT can leverage a kernel introspection module for kernel protection.

B. *Qualitative Analysis of* KSHOT

We design a system that enables reliable and efficient patching. Current live patching systems, like kpatch and Ksplice, depend on the correct execution of kernel functions, and thus implicitly trust the kernel and patching mechanism. As a result, a compromised, buggy, or vulnerable kernel may lead to failed deployments. To address this issue, we leverage SMM to process patches, which has two advantages. First, SMM is an isolated execution environment which cannot be accessed by host applications, including kernel rootkits or malware: the SMM Handler cannot be disrupted by such activities. Second, switching to SMM pauses the host system and restores the architectural state once the SMM Handler completes. We thus avoid implementing expensive checkpointing mechanisms (as in kpatch or KUP), considerably reducing storage overhead. This represents a tradeoff between two conflicting non-functional quality properties (space and time); we evaluate this tradeoff empirically in Section VI.

Since SMM effectively pauses the OS's execution, we must carefully choose which aspects of our system execute in the SMM Handler. We propose to implement only required functionalities in SMM (i.e., memory read/write capabilities) to quickly deploy patches once they are made available to the SMM Handler. Separately, we use an SGX enclave in userspace to securely download the patch and marshal the patch data into the SMM Handler. This SGX enclave allows the patch to be downloaded securely using the system's existing networking stack. Together, the SGX enclave and SMM Handler provide a low overhead, high efficiency, secure mechanism for applying kernel patches at runtime.

V. KSHOT DESIGN AND IMPLEMENTATION

The goal for KSHOT is to live patch an OS kernel with (1) minimal downtime, (2) minimal overhead, (3) support for compromised kernels, and (4) support for consistency without being kernel-specific. We implemented a prototype of KSHOT based on Intel SGX and x86 SMM. The SGX-based TEE supports receiving and preprocessing patches, providing security without the full overhead of SMM. Encrypted patches are processed in SMM and placed in an executable memory space. Via SMM, the system stores the state of runtime processes, restoring that state after applying the patch when SMM completes. This allows for the deployment of a trusted binary patch via a possibly-compromised target system.

A. *Binary Patch Preparation*

We leverage a trusted remote server to prepare binary kernel patches. First, basic information about the OS, including the kernel version, configuration, and compilation flags sufficient to rebuild the binary image, are all transferred to the remote server. The remote server then builds pre-patch and post-patch versions of the kernel binary using that same compilation information. A binary diff is sent back to the SGX enclave on the Target Machine.

In KSHOT, kernel vulnerabilities are patched via function-level changes. We thus patch the code for the affected functions, rather than replacing either the entire kernel image or just a few vulnerable instructions (see Section II). Replacing an entire compromised kernel with a patched one (as in KUP) is a powerful solution, but it incurs a significant storage overhead. By contrast, fixing individual vulnerable instructions is more flexible and has been demonstrated in previous work (e.g., KARMA [9] and kGraft [11]). However, instruction-level approaches also have significant drawbacks. First, current instruction-level patching relies on the OS kernel to monitor

the state of runtime functions and to decide if the instructions can be patched without introducing inconsistencies. Second, challenges such as identifying instructions in target functions [41] in the face of compiler optimizations are difficult, frequently leading to patching failures [42].

Identifying Target Functions: Given the pre- and post-patch binary kernel image, we extract all corresponding patched functions. While this process is complicated by compiler optimizations [42], we do not claim any novelty in our identification of the functions that must be updated, instead making use of a combination of existing algorithms and techniques. Our prototype builds a *source-level call graph* [43], [44] of the kernel by using the codeviz tool [45]. We also make use of IDA Pro [46] to create a *binary-level call graph* of the kernel binary image. Differences between the source- and binary-level call graphs illuminate certain compiler optimizations [42], including inlining, which is particularly common in OS kernels. Because functions may be transitively inlined, we employ a worklist algorithm that iteratively identifies implicated functions until no new implicated functions can added. KSHOT makes use of existing binary signature matching methods such as iBinHunt [47] and FIBER [42] to align and identify relevant sections of the binary kernel image.

For the purposes of discussion and evaluation, we group implicated functions into three broad categories (of increasing difficulty to support via kernel live patching). *Type 1* functions do not involve inlining. *Type 2* functions do involve inlining. *Type 3* functions modify global or shared variables.

For Type 3 analyses, we consider global or shared variables changed in the patch. Such a variable might be deleted, added, or modified. If the variable's size is not modified, the patch code is unaffected. However, if storage space for a variable is inserted or deleted, care must be taken to avoid inconsistent handling of that data between pre- and post-patch code. To handle such variable modifications, we change the corresponding variable and type in kernel memory (i.e., in *data* and *text* segments). In general, significant changes to storage layouts (e.g., adding or removing a field in a widely-used data structure) may result in patch application failures; we evaluate this empirically in Section VI.

Patching Target Functions: After we identify and analyze all relevant target functions, we must make the memory containing the (binary) newly-patched instructions accessible to the running kernel. In general, we cannot directly replace vulnerable function instruction memory with a patched function without compromising consistency. To solve this problem, we use *trampolines* (cf. [24]): We store the patched functions in a reserved memory space and link old code to the new functions by replacing the first instruction in the target function with a *jmp* instruction. The configurations of reserved memory, including memory size, location and page attributes, are all saved in SMM code in advance via the patch server. A basic trampoline approach addresses calls to the beginning of a function but does not address internal jumps and branches to intermediate labels. This is because the offset for each jump and branch in the post-patch binary may have changed. Thus, we must change these offsets to retain required functionality via the standard approach of calculating label differences.

KSHOT is a system for kernel-space patches that need not trust the operating system: our focus is on deploying a compiled binary patch in a compromised system (e.g., via hardware support) and we are agnostic to the underlying standard binary patching mechanism.

Supporting Kernel Tracing: Recent versions of the Linux kernel include a special form of tracing support [48] that is relevant to kernel live patching. When the trace attribute is enabled, more than half of functions ($23,000$ of $32,000$ in Linux 3.14) are compiled with a special 5-byte trace instruction sequence which can be dynamically changed at runtime by the kernel itself (not by our live patching). KSHOT must be aware of such tracing instructions to avoid conflicts. Naively patching an entire function containing such a tracing sequence will result in incorrect execution or other memory errors at runtime. Since the tracing instructions are located at a fixed offset from the entry of the function, our solution is to identify such 5-byte trace instruction signatures and patch the instructions after them, leaving the tracing itself untouched.

B. SGX-based Patch Preparation

KSHOT uses Intel SGX hardware support to safeguard trusted live patch preprocessing. The preparation of executable binary patches proceeds in a trusted environment before the processed patch is made available to the SMM-based live patching module. In this subsection, we describe our SGX enclave behavior. We assume that collecting information about the current OS kernel can be done safely at boot time, and that such information can be passed to the remote patch server that produces the binary patch. In addition, we encrypt communication when obtaining the binary patch from the remote server. This is also particularly relevant when passing data between the SMM handler and SGX enclave. Both communications are handled by untrusted applications or network drivers—we encrypt data while in transit.

Due to the isolation properties of Intel SGX enclaves and SMRAM, there is no direct channel for data transmission between them. To exchange data between these two entities, we use shared memory for encrypted data transmission. In general, unless care is taken, there may not be a spare kernel memory region available. In addition, if we live patch an existing kernel function, it may change the function size and cause a kernel consistency issue. We address these issues by reserving a physical memory space for KSHOT at boot time.

Memory Protection and Isolation: We first configure the boot loader (e.g., *grub*) to reserve a suitable kernel memory allocation space (18MB for our prototype implementation). We also add page attribute operation code to the *paging_init* function to provide the appropriate access limitations for that memory. The reserved memory includes three logical parts: *mem_RW*, *mem_W*, and *mem_X*. The small *mem_RW* is a read/write area used for key exchange. Our prototype uses the *Diffie-Hellman* key exchange algorithm [49]. The larger *mem_W* region is write-only and is used for storing the

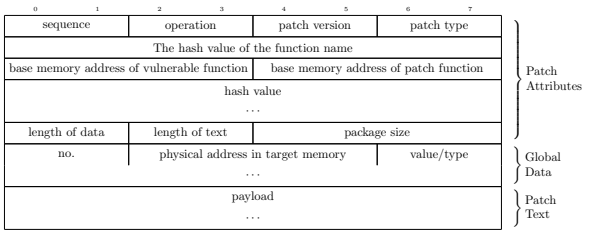

Fig. 3: The structure of patch package transmitted from SGX enclave to SMM.

encrypted patch text. The untrusted application writes data from the SGX enclave into *mem_W*. However, the untrusted application cannot decrypt this output data. Finally, the much larger *mem_X* region is executable-only and is used to store decrypted patched instructions as the kernel text. Read and write access to those instructions is prohibited (as is standard with kernel function memory) to maintain integrity. Moreover, we can use existing SMM-based runtime checking systems [39], [50] to further ensure the integrity of this region.

These access control mechanisms only limit the OS kernel. By contrast, the hardware-supported SMM handler can read and write any reserved memory. The SGX enclave receives the post-compilation binary patch. KSHOT formats the instruction text, adds external message fields to ensure that the SMM handler can process the text correctly, and places the text in right memory position and alignment.

Patch Preprocessing: The SGX enclave receives a patch set from the remote patch server $P = \{p_1, \ldots, p_n\}$, with edits to n functions. An individual patch p_i has the form $\{sequence, opt, type, \ldots, payload\}$. The details are shown in Figure 3. The patch preparation workflow follows a standard sequence of steps. First, we verify the integrity of the received patch to guard against network transmission errors. Next, the modified binary patch will be written out as an executable memory block. We package this memory block with external header information (Figure 3). We encrypt this data in the SGX enclave. The outside untrusted application then passes the encrypted data to the mem_W segment. After that, an SMI is triggered to transfer control to the SMM-based live patching component.

C. SMM-based Live Patching

The CPU changes to System Management Mode when it receives a triggering instruction. The SMM hardware ensures that the latest runtime state and register values are saved to the protected SMRAM region of memory. Before the patching, a *Diffie-Hellman (DH)* key generation module is executed in SMM to create the private key, which is used to encrypt/decrypt the patch related data in SMM. This cryptographic key is dynamically changed before each kernel patch to guard against replay attacks between data transmissions. While a Man-in-The-Middle (MITM) attack could still intercept the communication between the SGX enclave and SMM, KSHOT can verify the enclave's identity via the trusted patch server and thus mitigate the MITM attack. We implement the live patching process in the SMM handler, including integrity checking and the patching module itself. The following is the workflow of patching operations performed in SMM.

First, the data fetching function in SMM obtains the binary patch packages from the mem_W segment. We compare a cryptographic hash of the payload to the hash stored in the package header to ensure patch integrity.

Second, we check if any global variable needs to be changed in the kernel *data* or *bss* segment. To ensure data consistency, we locate the addresses of global data in either segment using the kernel symbol table and change the value/type of each.

Third, we check the *operation* field in the package. If the value is *patch*, then we add a jump instruction as the redirection at the entry of that vulnerable function. We define the location address of the patch function *paddr* at *mem_X*. The location address of the first patching function $p_1.paddr$ is the base address of *mem_X*. Then, the location address of the ith patching function is $p_i.paddr = p_{(i-1)}.paddr + p_{(i-1)}.size$, where $size$ denotes the size of a binary patch. The binary patch p_i is then placed between the memory of $p_i.paddr$ and $p_i.paddr + p_i.size$. The trampoline instruction at $p_i.taddr$, where $taddr$ is the physical memory base address of the vulnerable function, is replaced with a jmp instruction with the offset value of $p_i.paddr - p_i.taddr + 5$, which ensures that process will be redirected to the patch function once the vulnerable function is called (and respects the 5-byte kernel tracing setup).

Once the redirection instruction is set up, the system switches back to Protected Mode and resumes the OS.

Patch Rollback/Update. After patching the kernel, the system or its applications may not run correctly for many different reasons [8]. For example, the patch may introduce a new bug or cause a new vulnerability. Indeed, a software engineering study of commercial and open source operating systems by Yin *et al.* found that 15–24% of human-written OS patches were incorrect and resulted in end-user-visible impacts such as crashes or security problems [22]. Supporting rollback is thus critical for a realistic deployment. In such situations, we can send a rollback instruction from remote sever. The SMM handler rolls back the patch function to the original function. We keep the patch information in SMM and store the original instruction in mem_W. As a result, if a rollback operation is triggered, we can fetch out the original instruction and replace the jump instruction in the vulnerable function. In KSHOT, the last patching operation can always be rolled back in this manner.

D. Patching Protection

In this subsection, we discuss several techniques we employ to address potential malicious interference with our live patching process.

Malicious Patch Reversion. Some latent attacks in a compromised OS might revert the patch with an original (i.e., vulnerable) version of the kernel or function. However, KSHOT can mitigate such attacks by leveraging SMM-based introspection. Specifically, we use SMM-based kernel protection

mechanisms [39], [50] to prevent the Target OS from reversion or modification by rootkits after applying the patching.

We can similarly use the SMM handler to introspect regions of memory overwritten with trampoline instructions to ensure that the patched version of code persists after deploying a patch with our approach. Because SMM has higher privilege than the kernel, and because it can transparently introspect the Target OS, it can detect changes to the kernel text and data.

Denial-of-service attacks. DOS attacks may preclude the patch preparation operation from running, leading to a live patching failure. DOS attacks are generally difficult to defend [51], [52], however we can detect DOS attacks using SMM-based introspection techniques. After the Remote Server sends the Patch source to the SGX Enclave in the Target OS, the Enclave and the Remote Server can communicate the state of the Patch Preparation. Once the patch binary is written in to the Reserved Memory, the Remote Server can verify with the SMM Handler that the patch binary was written to memory (i.e., via introspection in the SMM Handler). This approach cannot prevent DOS attacks but can detect them.

VI. Evaluation

We evaluate the applicability, performance, and security of KSHOT when live patching Linux kernels. Our prototype machine uses an Intel Core i7 CPU (supporting SGX and SMM) with 16GB memory. We use a combination of Coreboot [53] with a SeaBIOS [54] payload as the system BIOS. We experiment with Ubuntu 14.04 and 16.04 using kernel versions 3.14 and 4.4.

We consider three research questions:

- RQ1. Can KSHOT correctly apply kernel patches?
- RQ2. What is KSHOT's performance overhead?
- RQ3. How does KSHOT compare to existing approaches?

A. Benchmark Selection

We evaluate KSHOT's ability to patch critical kernel vulnerabilities by using a suite of real-world patches from the Common Vulnerabilities and Exposures (CVE) database [55]. We analyzed 267 such vulnerabilities for Linux kernels 3.14 and 4.4. Of these 267, we found that 214 of them were reproducible and applicable for our x86 architecture. The remaining cases were excluded for one of two reasons: either the vulnerability applied to a non-x86 platform (e.g., Android or embedded devices), or the patch involved complex data structure changes beyond the scope of our patching framework (discussed further in Section VIII).

We randomly selected 30 of those 214 patches to construct a benchmark suite similar in scale to existing work [9], [56]. The selected patches are listed in Table I. The "CVE Number" field identifies the associated kernel defect. The "Affected Functions" field lists the kernel functions changed by the patch. The "Patch Size" field lists total size, in lines of code, of all changed functions in post-patch version (this corresponds to the size of the patch that KSHOT must deploy).

TABLE I: Benchmark suite of 30 critical kernel patches.

CVE Number	Affected Functions	Size	Type⋆
CVE-2014-0196[1]	n_tty_write	86	1
CVE-2014-3687[1]	sctp_chunk_pending, ctp_assoc_lookup_asconf_ack	16	1,2
CVE-2014-3690[1]	vmx_vcpu_run, vmcs_host_cr4, vmx_set_constant_host_state	247	3
CVE-2014-4157[1]	current_thread_info	5	2
CVE-2014-5077[1]	sctp_assoc_update	98	1
CVE-2014-5206[1]	do_remount	34	2
CVE-2014-7842[1]	handle_emulation_failure	16	1
CVE-2014-8133[1]	set_tls_desc, regset_tls_set	81	1,2
CVE-2015-1333[1]	__key_link_end	21	1
CVE-2015-1421[1]	sctp_assoc_update	96	1
CVE-2015-5707[1]	sg_start_req	117	1
CVE-2015-7872[1]	key_gc_unused_keys, request_key_and_link	20	1
CVE-2015-8812[1]	iwch_l2t_send, iwch_cxgb3_ofld_send	26	1
CVE-2015-8963[1]	perf_swevent_add, swevent_hlist_get_cpu, perf_event_exit_cpu_context	72	3
CVE-2015-8964[2]	tty_set_termios_ldisc	10	2
CVE-2016-2143[2]	init_new_context, pgd_alloc, pgd_free	53	2
CVE-2016-2543[2]	snd_seq_ioctl_remove_events	25	1
CVE-2016-4578[1,2]	snd_timer_user_ccallback	24	1
CVE-2016-4580[2]	x25_negotiate_facilities	67	1
CVE-2016-5195[2]	follow_page_pte, faulti_page	229	1,3
CVE-2016-5829[2]	hiddev_ioctl_usage	119	1
CVE-2016-7914[2]	assoc_array_insert-_into_terminal_node	330	1
CVE-2016-7916[2]	environ_read	63	1
CVE-2017-6347[1,2]	ip_cmsg_recv_checksum	15	2
CVE-2017-8925[1,2]	omninet_open	9	2
CVE-2017-16994[2]	walk_page_range	27	1
CVE-2017-17053[2]	init_new_context	13	2
CVE-2017-17806[1,2]	shash_no_setkey, hmac_create, crypto_shash_alg_has_setkey	91	1,2
CVE-2017-18270[1,2]	key_alloc, install_user_keyrings, join_session_keyring	273	1,2
CVE-2018-10124[1.2]	kill_something_info, sys_kill	51	1,2

[1] affects Linux 3.14. [2] affects Linux 4.4. ⋆ indicates patch type (Section V-A).

B. RQ1 — Correct Kernel Patching

We evaluated KSHOT on Linux kernels running on live hardware. We determined that the system was in a stable state with the default Ubuntu 14.04 or 16.04 background processes running. We then instructed KSHOT to apply the appropriate patch and manually verified its correct deployment (e.g., no kernel panics, no crashed processes, no system log errors or warnings, etc.). We also conducted experiments with heavier active workloads during live patching (see Section VI-C3). Our primary result is that KSHOT *correctly applied live patches*

in all 30 cases considered, demonstrating that our system is applicable across multiple OS versions and defect types.

To provide additional insight into our successful applicability results, we detail a few patches as case studies. Recall from Section V-A that we can classify each kernel patch into one of three categories. Type 1 patches involve no inlining and thus have their own independent instruction memory (a default, simple case). Type 2 patches involve inlining. Type 3 patches require changes to kernel data structures or global variables. We discuss an example patch from each category that we considered.

Example Type 1 Patch: We consider CVE-2017-17806. This vulnerability admits a kernel stack buffer overflow when a local attacker executes a crafted sequence of system calls that encounter a missing SHA-3 initialization and eventually a stack-out-of-bounds bug. The official fix, partially shown in Listing 1, is to add the cryptographic check to the relevant kernel function (see Line 7). This is our most direct case.

Listing 1 Type 1 example: CVE-2017-17806 patch

```
1    static int hmac_create(struct crypto_template *tmpl,
         struct rtattr **tb)
2    salg = shash_attr_alg(tb[1], 0, 0);
3    if (IS_ERR(salg))
4    return PTR_ERR(salg);
5  +  alg = &salg->base;
6    err = -EINVAL;
7  +  if (crypto_shash_alg_has_setkey(salg))
8  +     goto out_put_alg;
9  +
10   ds = salg->digestsize;
11   ss = salg->statesize;
12 - alg = &salg->base;
```

Listing 2 Type 2 example: CVE-2017-17053 patch

```
1    static inline int init_new_context(struct task_struct
         *tsk,
2    ...
3    #endif
4  -  init_new_context_ldt(tsk, mm);
5  -  return 0;
6  +  return init_new_context_ldt(tsk, mm);
```

Example Type 2 Patch: Consider the use-after-free vulnerability CVE-2017-17053. In this bug, the Linux kernel does not correctly handle errors from certain table allocations when forking a new process, allowing a local attacker to achieve a use-after-free via a specially-crafted program. In the official fix for this bug, the return value in function *init_new_context* is changed (see Listing 2, Line 6). Critically for KSHOT, this patch involves inlining, so more than one function is implicated and must be updated (as listed in Table I).

Listing 3 Type 3 example: CVE-2014-3690 patch

```
1    struct vcpu_vmx {
2      int           gs_ldt_reload_needed;
3      int           fs_reload_needed;
4      u64           msr_host_bndcfgs;
5  +   unsigned long vmcs_host_cr4
6    } host_state;
```

Example Type 3 Patch: We consider CVE-2014-3690 as an example Type 3 patch involving updates to local data structures. The official patch, partially shown in Listing 3, adds a new field to local struct *vcpu_vmx*. In addition, function *vmx_set_constant_host_state* assigns a value to the new field, and function *vmx_vcpu_run* reads the field's value. Thus, both functions must be patched. KSHOT successfully applies this patch, but Type 3 cases remain difficult in general; we return to this issue in Section VIII.

C. RQ2 — Performance Evaluation

To evaluate the performance of KSHOT, we measured each stage of the live patching process. We consider overhead from two sources: SGX-based binary patch preparation and SMM-based patching. Since the SMM patching process essentially pauses the target OS but the SGX-based enclave does not, we evaluate the performance of two parts separately, including a comparison with existing methods. In our experiments, the total size of the binary patch generally ranged from 40 bytes to 4KB.

1) SGX-Based Patch Preparation Performance: The SGX enclave must (1) fetch the patch from the remote server, (2) preprocess the patch through integrity checking and branch instruction replacing, (3) pass the patch with encrypting and writing to shared memory region for consumption by the SMM side. We evaluate the time consumption in each step.

Table II shows a breakdown of the time consumed by this SGX-based patch preparation for various patch sizes, averaged over 100 trials. Consider the 4KB case as an example. The time to fetch a binary patch from our remote server is 200μs, and the time to prepare the patch is 8,034μs. In addition, 51μs is required to store the encrypted binary patch into the shared memory region. All told, we use 8,285μs to complete the preprocessing of a 4KB patch.

2) SMM-Based Patching Performance: The SMM handler pauses the target OS while carrying out key generation, data reading and decryption, patch verification, and binary patch activities. In addition, there are overheads associated with switching to and from SMM and protected mode. We evaluate these times empirically using the $rdtsc$ instruction to count the number of CPU cycles elapsed during each operation.

For our experimental platform, the average times for switching to, and resuming from, SMM are 12.9μs and 21.7μs, respectively. These values depend on specific hardware configuration, but are typically on the same order of magnitude in our experience. Once we switch to SMM, we spend 5.2μs to generate encryption keys. The switching operation and key generation are fixed-cost operations, regardless of patch size.

TABLE II: Breakdown of SGX operations (μs; $n = 100$).

Patch Size	Fetching	Pre-processing	Passing	Total
40B	54	150	9	213
400B	68	850	29	947
4KB	200	8,034	51	8,285
40KB	2,266	82,611	498	85,375
400KB	16,707	785,616	4,985	807,308
10MB	415,944	19,991,979	124,565	20,532,488

TABLE III: Breakdown of SMM operations (μs; $n = 100$).

Patch Size	Data Decryption	Patch Verification	Patch Application	Total[1]
40B	0.04	2.93	0.06	42.83
400B	0.31	6.32	0.72	47.15
4KB	1.27	8.52	6.92	56.51
40KB	13.84	33.85	17.22	104.71
400KB	133.30	311.15	396.45	880.70
10MB	2,832.00	5,973.00	2,619.00	11,464.00

[1] includes key generation and SMM switching time.

The SMM handler reads the encrypted patch provided by the SGX enclave, then applies it to the kernel memory. The time taken to read, decrypt, and apply the patch depends on the patch size. We tested patch sizes ranging from 40 bytes to 10MB. Table III shows the time breakdown of patching operations for various patch sizes. For example, a 4KB patch takes $1.27\mu s$ to read and decrypt, $8.52\mu s$ to verify, and $6.92\mu s$ to apply to kernel memory (e.g., to actually write the new patch to memory). Note that the majority of the patch time comes from the patch verification process, which involves computing a SHA-2 hash. We could reduce this time by employing a simpler hashing algorithm such as SDBM [57].

The overhead grows approximately linearly with the patch size. Even in the case of a large 40MB patch, the total required time is under 1 second. On average, the patches from our CVE dataset are less than 1KB. Note that we did not count the overhead imposed by communication between the Patch Server and Target Machine's untrusted helper application, which has minimal effect on the SGX enclave. Extrapolating from Table III, the average patch thus requires roughly $74\mu s$. We view this as a small and acceptable time interval to pause the system, especially given the rarity of live patching events.

3) Whole-System Performance Evaluation: We randomly selected 6 of our benchmarks for a detailed analysis of whole-system performance.[1] In addition to the patched code itself, each function requires 42 bytes of header data in the transmitted patch package (following the packaging process from Figure 3). Figure 4 shows that the time breakdown in the SGX preprocessing stage, which indicates the majority of time is spent preprocessing the patch according. Similarly, Figure 5 shows the time breakdown in SMM for each patch. Larger patches require more patching time, while the switching and key generation times are relatively constant across all patches. In these whole-system experiments, KSHOT required very little time to apply each patch. For example, for *CVE-2014-4608*, the total time required on the Target Machine was about 7,941μs for a 156-byte patch. The preparation time in SGX dominates the time cost, and the system is only paused for a brief 47.6μs during SMM activities. This including 5.2μs for key generation and 34.6μs for SMM switching. The patch completed successfully, without changing application state.

We also used *Sysbench* [58] to measure overall system overhead. We live patched the kernel while *Sysbench* executed in userspace and measured end-user-visible system overhead. Over 1,000 live patches of each of the 6 aforementioned CVE patches, we incur under 3% overhead from the combined SGX and SMM patch preparation and deployment times.

D. RQ3 — Patching System Comparison

KSHOT provides a live and reliable mechanism for kernel patching with the help of Intel SMM and SGX. We compare KSHOT with existing general-purpose live patching systems and also with live kernel patching systems.

1) General Patching Comparison: Table IV presents a comparison of KSHOT to indicative non-kernel and kernel binary patching approaches used in more general software engineering contexts. To the best of our knowledge, only KSHOT does not require trusting or depending on the OS kernel. The Dyninst [24] and EEL [10] systems can be applied to patch executable binary files. However, they do not handle runtime memory. Kernel live patching systems must traditionally handle application state in some manner — either through checkpointing and recovery (as with many previous approaches) or through hardware assistance and pausing (as in KSHOT). The Libcare [25] system uses system calls and hooks to replace buggy functions in a userspace process's memory. In a typical use, the replaced function is only used by one process; by contrast, kernel live patching faces more significant consistency issues. The Kitsune [59] and PROTEOS [26] systems are dynamic software updating approaches. They take advantage of developer annotations of safe update points. Developer-marked software locations are assumed to admit correct patching. By contrast, KSHOT infers target functions automatically and uses hardware support to create safe pauses for updates.

[1]patches for CVE-2014-4608, CVE-2015-7872, CVE-2016-2143, CVE-2016-5696, CVE-2017-16994, CVE-2017-18270; corresponding patch sizes: 198, 171, 257, 79, 174, 322 bytes.

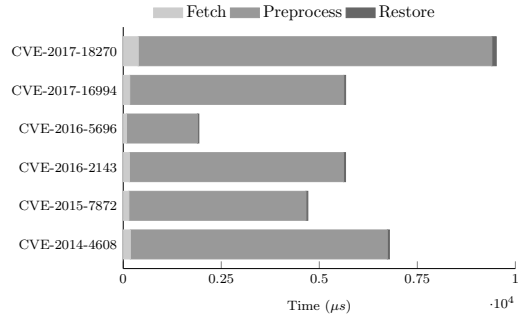

Fig. 4: SGX-based patch preparation time.

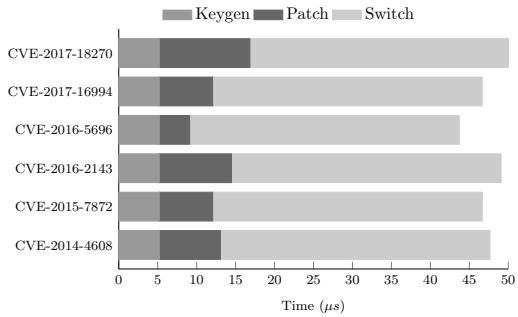

Fig. 5: SMM-based live patching time.

TABLE IV: Comparison with non-kernel binary patching.

	Kernel Dependency	Untrusted OS	Applicability
Dyninst [24]	✓	✗	userspace
EEL [10]	✓	✗	userspace
Libcare [25]	✓	✗	userspace
Kitsune [59]	✓	✗	userspace
PROTEOS [26]	✓	✗	kernel
KSHOT	✗	✓	kernel

TABLE V: Comparison with kernel patching systems.

	Type	Downtime	Untrusted OS	Memory
KUP [8]	kernel	3s/kernel	✗	>30G
KARMA [9]	instruction	5μs/patch[1]	✗	lua engine
kpatch [10]	function	45.6ms/patch[1]	✗	16G
KSHOT	function	50μs/patch[1]	✓	18M

[1] for an averaged sized patch of less than 1KB

2) Kernel Patching Comparison: Existing kernel live patching systems assume that patches are trusted when they are stored in the target OS. However, the integrity of patches can be easily compromised by attacks which have the kernel access privilege (e.g., syscall_hijacking [19]). By contrast, KSHOT leverages the SGX enclave to preprocess binary patches without having to trust the underlying OS. Additionally, data blocks transmitted between SGX and SMM through the shared memory are encrypted to protect the patch's integrity from malicious modification during preprocessing.

In addition, existing solutions rely on kernel-specific functions to implement the patching operations (e.g., *ptrace, stop_machine, kexec*). However, existing vulnerabilities [55], such as CVE-2015-7837, CVE-2014-4699, or CVE-2012-4508, can affect those particular kernel functions. For example, the CVE-2015-7837 vulnerability allows the attacker to load an unsigned kernel via *kexec*, which would compromise KUP's patching mechanism. In KSHOT, live patching operations execute in the SMM handler, which cannot be modified even if the underlying Target OS is compromised. Our use of SMM as a trusted execution environment for deploying patches prevents a compromised OS from interfering with KSHOT.

We compare KSHOT with representative kernel live patching methods (including KUP, KARMA, kpatch) in Table V in terms of patch granularity, patching time, trusted code base, and memory consumption. KUP replaces an entire vulnerable kernel in around 3 seconds. Additionally, KUP can handle patches with complex data structure changes. KARMA requires less than 5μs for small patches and uses very little memory. kpatch takes longer, but it can be deployed and integrated in the Linux kernel. However, these existing methods all rely on the OS kernel (and thus their TCB includes the whole kernel). By contrast, in KSHOT, the TCB extends only to SMM and the SGX enclave. Moreover, KSHOT needs no checkpointing of running applications, and uses only 18MB extra memory space for patch analysis and management. Moreover, KSHOT requires only about 50μs to deploy most patches, which is faster than all existing non-instruction-level methods. Our approach provides an efficient and secure live patching mechanism.

E. Evaluation Summary

We find that KSHOT is a general, performant, secure approach to live patching vulnerable Linux kernels. Across an indicative benchmark of 30 critical kernel security vulnerability patches, we correctly applied *all* of them successfully with our approach. Based on our combination of SGX and SMM patch preparation and deployment, KSHOT incurs under 3% total system overhead over $1,000$ live patches. Finally, this approach requires a substantially smaller TCB compared to previous techniques.

To put these results in context, we discuss two of our kernel patches with respect to time from vulnerability discovery to patch to adoption. First, CVE-2014-8133 was first discovered 10 October 2014, but a patch was not created until 14 December 2014 in Linux 3.13. Moreover, this patch did not get merged into Ubuntu 14.04 until 26 Feb 2015. Second, CVE-2017-17806 was discovered 17 October 2017, with a corresponding patch built 29 November 2017 for Linux 4.4, and merged into Ubuntu 16.04 on 4 April 2018. These timelines match industry reports that critical CVEs take an average of over a month to get patched [60]. However, even when a patch is created, it may take additional time for end users to adopt the new patch [61]—many successful exploits rely on old, previously-patched vulnerabilities [62]. Live patching techniques are intended in part to reduce the cost associated with applying an update, and techniques like KSHOT show promise in furthering that cost reduction while extending kernel live patching capabilities.

VII. Related Work

In this section, we survey related work from the areas of trusted execution environments, patch analysis, and live patching methods.

A. Trusted Execution Environment

Trusted execution environments (TEE) are intended to provide a safe haven for programs to execute sensitive tasks. Being able to run programs in a trusted execution environment is crucial to guarantee the program's confidentiality and integrity. Hardware-based TEEs include x86 SMM [63], Intel SGX [64], [65], AMD memory encryption technology [66], and ARM TrustZone [67]. HyperCheck [39] leverages SMM to build a trusted execution environment and monitor hypervisor integrity. VC3 [21] leverages Intel SGX to provide an isolated region for secure big data computation. Scotch [68] combines x86 SMM and Intel SGX to monitor cloud resource usage. KSHOT uses a TEE for reliable kernel live patching.

B. Patch Analysis

Traditional patching mechanisms simply apply the source-code-based patch to the kernel source, re-compile, and reboot to install the new kernel. In live patching, we directly replace binary-level code with a new version at runtime. However, both approaches must identify the target code and prepare the patch code by analyzing the source or binary code. Patch analysis methods [42], [69]–[71] can be classified into two

broad types: source-to-source and binary-to-binary. Source-to-source methods require both the original source code and the patch code. To identify the functions in source code, methods such as string [72], token [73], and parse tree [74] matching can be used. Moreover, the call graph [71] and control flow graph [75] can be constructed to identify relationships between functions. By contrast, binary-to-binary methods do not require source code. Both the patch and the target are presented in a binary format, and all comparisons are based only on binary-level features. To accurately identify relevant function in binary code, tools such as *IDA* [46] can be used to extract relevant information. Additionally, techniques such as BinHunt [76] and iBinHunt [47] use symbolic execution and theorem proving to formally verify basic block level semantic equivalence. Fiber [42] employs a precise and accurate patch code matching mechanism with the source patch code and binary vulnerable functions. KSHOT analyzes compiled kernel binary code as well as the source code patch to obtain rich information (see Section V). Our prototype evaluation uses codeviz, IDA, iBinHunt and Fiber, but our approach is agnostic and could employ any similar tool.

C. Live Patching

Existing live patching focuses on open-source operating systems, mainly Linux. For example, Ksplice [12], kpatch [10], and kGraft [11] can effectively patch security vulnerabilities without causing a significant downtime. kpatch and Ksplice both stop the running OS and ensure that none of the processes are affected by changes induced by patched functions. Specifically, kpatch replaces the whole functions with patch ones, and Ksplice patches individual instructions instead of functions. kGraft patches vulnerabilities at function level, but does not need to stop the running processes. It maintains the original and patched function simultaneously and decides which one to execute by monitoring the state of processes, potentially inducing incorrect behavior or consuming additional storage. These methods cannot address changes to data structures [8]. To address this limitation, KUP [8] replaces the whole kernel with a patched version, but uses checkpoint-and-restore to maintain application state consistency. However, it checkpoints all the user processes, leading to large CPU and memory overhead. KARMA [9] uses a kernel module to replace vulnerable instructions that it identifies from a given patch diff. In addition, several live updating methods have been integrated into operating systems, like Canonical Livepatch Service [13] in Ubuntu, and Proteos [26] on MINIX 3, which can update new components if the patch is small. However, these methods still rely on the trustworthy operation of the target OS, so potential kernel-level attacks may tamper with the live patching operation, leading to system failure. KSHOT addresses this by leveraging a TEE to reliably patch the target kernel with a smaller TCB and low total overhead.

VIII. Limitations and Future Work

In this section, we discuss potential limitations of our kernel live patching approach.

Attacks to Trusted Execution Environments: While we treat the x86 SMM and Intel SGX as a foundation to implement KSHOT, they might be compromised through vulnerabilities at the hardware or firmware level. For example, the recent Foreshadow [32] is able to leak the information from SMM or SGX. Some other attacks like SMM rootkits [33], [34] or Software Grand Exposure [77], can also compromise SMM or SGX; we consider these attacks beyond our scope.

Downtime for SMM Handler operations: Although KSHOT outperforms existing kernel patching systems, it still introduces some downtime for patching. This is because the SMM Handler suspends the OS while applying the patch. Note that KSHOT minimizes this downtime by moving the preprocessing operations from (blocking) SMM to the (non-blocking) SGX enclave.

Consistency Issues: Some complex patches may change the semantics of target functions, which might affect other non-patched functions. For example, a patch might change the order in which locks are acquired in multiple functions at the same time, or some patches might change global data used by multiple functions. Currently, KSHOT cannot handle those cases. Our empirical evaluations suggest this is rare, occurring in around 2% of kernel CVE patches. One way to address this problem is to construct a consistency model and safely choose patch tasks [10], [13], identify and patch all relevant functions, which can be applied even to unstructured programs (e.g., [78]). We leave this consideration for future work.

IX. Conclusions

In this paper, we presented KSHOT, a secure and efficient framework for kernel patching. It leverages x86 SMM and Intel SGX to patch the kernel without depending on the OS. Additionally, we use SMM to naturally store the runtime state of the target host, which reduces external overhead and improves live patching performance. Employing this hardware-assisted mechanism supports faster restoration without external checkpoint-and-restore solutions. We evaluate the effectiveness and efficiency of KSHOT by providing an in-depth analysis of the technique against a suite of indicative kernel vulnerabilities. We demonstrate that our approach incurs an average downtime of 50μs for a 1KB binary kernel patch, but consumes only 18MB of extra state for patch analysis, a substantial reduction over previous work. **In an empirical evaluation on 30 randomly-selected indicative, critical kernel CVEs, KSHOT live-patched each one successfully with low overhead and a small trusted code base.**

Acknowledgments. We would like to thank our shepherd, Miguel Correia, and the anonymous reviewers for their insightful comments that improved the paper. This work is partly supported by National Science Foundation Grant No. CCF 1763674, Air Force Grant No. FA8750-19-2-0006, National Natural Science Foundation of China Grant No. 61632009, and Guangdong Provincial Natural Science Foundation Grant No. 2017A030308006. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the agencies.

REFERENCES

[1] S. Farhang, J. Weidman, M. M. Kamani, J. Grossklags, and P. Liu, "Take It or Leave It: A Survey Study on Operating System Upgrade Practices," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[2] F. Vitale, J. Mcgrenere, A. Tabard, M. Beaudouin-Lafon, and W. E. Mackay, "High Costs and Small Benefits: A Field Study of How Users Experience Operating System Upgrades," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.

[3] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[4] Gartner, "Ensure Cost Balances With Risk in High-Availability Data Centers," https://www.gartner.com/en/documents/3906266/ensure-cost-balances-with-risk-in-high-availability-data, 2019.

[5] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 271–281.

[6] M. Nabi, M. Toeroe, and F. Khendek, "Rolling upgrade with dynamic batch size for iaas cloud," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016.

[7] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, "Katana: A hot patching framework for elf executables," in *2010 International Conference on Availability, Reliability and Security*. IEEE, 2010, pp. 507–512.

[8] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov, "Instant OS updates via userspace checkpoint-and-restart." in *USENIX Annual Technical Conference*, 2016.

[9] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive Android kernel live patching," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[10] J. Poimboeuf and S. Jennings, "Introducing kpatch: dynamic kernel patching," *Red Hat Enterprise Linux Blog*, vol. 26, 2014.

[11] SUSE, "Live Patching the Linux Kernel Using kGraft," https://www.suse.com/documentation/sles-15/book_sle_admin/data/cha_kgraft.html, 2018.

[12] ORACLE, "Ksplice," http://www.ksplice.com/, 2018.

[13] Ubuntu, "Canonical Livepatch Service," https://www.ubuntu.com/livepatch, 2018.

[14] Checkpoint, "Restore in Userspace," https://criu.org/Main_Page, 2018.

[15] Github, "Kpatch bugs," https://github.com/dynup/kpatch/issues, 2019.

[16] Windows Defender ATP, "Software supply chain cyberattack," https://www.microsoft.com/security/blog/2017/05/04/windows-defender-atp-thwarts-operation-wilysupply-software-supply-chain-cyberattack/?source=mmpc, 2017.

[17] GitHub, "APT/APT-GET RCE vulnerability," https://github.com/freedomofpress/securedrop/issues/4058, 2019.

[18] Kaspersky, "Operation ShadowHammer," https://securelist.com/operation-shadowhammer/89992/, 2019.

[19] GitHub, "Syscall Hijacking on Linux Kernel," https://github.com/crudbug/simple-rootkit/, 2014.

[20] I. Khalil, A. Khreishah, and M. Azeem, "Cloud computing security: A survey," *Computers*, vol. 3, no. 1, pp. 1–35, 2014.

[21] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 38–54.

[22] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *Foundations of Software Engineering*, 2011, pp. 26–36. [Online]. Available: https://doi.org/10.1145/2025113.2025121

[23] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 4, p. 13, 2014.

[24] W. R. Williams, X. Meng, B. Welton, and B. P. Miller, "Dyninst and mrnet: Foundational infrastructure for parallel tools," in *Tools for High Performance Computing 2015*. Springer, 2016, pp. 1–16.

[25] Github, "Libcare – patch userspace code on live processes," https://github.com/cloudlinux/libcare, 2019.

[26] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 279–292.

[27] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, 2016.

[28] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[29] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang, "Establishing trusted i/o paths for sgx client systems with aurora," *IEEE Transactions on Information Forensics and Security*, 2019.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[32] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," Technical report, Tech. Rep., 2018.

[33] S. Embleton, S. Sparks, and C. C. Zou, "SMM rootkit: a new breed of OS independent malware," *Security and Communication Networks*, 2013.

[34] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," *CanSecWest, Vancouver, Canada*, 2009.

[35] A. Zavou, G. Portokalidis, and A. D. Keromytis, "Taint-exchange: A generic system for cross-process and cross-host taint tracking," in *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*, 2011, pp. 113–128. [Online]. Available: https://doi.org/10.1007/978-3-642-25141-2_8

[36] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 1271–1287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

[37] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 875–892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx

[38] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 72–83. [Online]. Available: http://doi.acm.org/10.1145/1133981.1133991

[39] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assistedintegrity monitor," 2014.

[40] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A Dependable Introspection Framework via System Management Mode," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[41] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23th USENIX Security Symposium*, 2014.

[42] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *Proceedings of the 27th USENIX Security Symposium*, 2017.

[43] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, 1982.

[44] S. Poznyakoff, "GNU cflow," http://www.gnu.org/software/cflow/, 2005.

[45] K. Mgebrova, "CodeViz: a callgraph visualizer," http://www.csn.ul.ie/~mel/projects/codeviz, 2012.

[46] H. Rays, "IDA Tools," https://www.hex-rays.com, 2018.

[47] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[48] S. Rostedt, "Ftrace Linux Kernel Tracing," in *Linux Conference Japan*, 2010.

[49] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater, "Provably authenticated group Diffie-Hellman key exchange," in *Proceedings of the 8th ACM conference on Computer and Communications Security*, 2001.

[50] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[51] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured routines: Language-based construction of trusted execution environments," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 571–586.

[52] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1723–1740.

[53] Coreboot, "Open-Source BIOS," http://www.coreboot.org/, 2018.

[54] SeaBIOS, http://www.coreboot.org/SeaBIOS, 2018.

[55] MITRE CVE Team, "CVE Details: The ultimate security vulnerability datasource," https://www.cvedetails.com/, 2019.

[56] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[57] A. Partow, "General Purpose Hash Function Algorithms," http://www.partow.net/programming/hashfunctions, 2018.

[58] GitHub, "Sysbench," https://github.com/akopytov/sysbench, 2016.

[59] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *International Conference on Verified Software: Tools, Theories, Experiments*. Springer, 2012, pp. 278–293.

[60] Rapid7, https://blog.rapid7.com/2018/08/22/whats-going-on-in-production-application-security-2018/, August 2018.

[61] P. Kotzias, L. Bilge, P.-A. Vervier, and J. Caballero, "Mind your own business: A longitudinal study of threats and vulnerabilities in enterprises." in *NDSS*, 2019.

[62] R. A. Grimes, "Zero-days aren't the problem – patches are," https://www.csoonline.com/article/3075830/zero-days-arent-the-problem-patches-are.html, June 2016.

[63] Intel, "64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2018. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[64] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[65] Y. Wang, Y. Shen, C. Su, K. Cheng, Y. Yang, A. Faree, and Y. Liu, "Cfhider: Control flow obfuscation with intel sgx," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, 2019.

[66] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption, White Paper," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, April 2016.

[67] ARM Ltd., "ARM Security Technology - Building a Secure System using TrustZone Technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[68] K. Leach, F. Zhang, and W. Weimer, "Scotch: Combining Software Guard Extensions and system management mode to monitor cloud resource usage," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[69] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[70] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 462–472.

[71] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for Java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

[72] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," in *SIAM Journal on Computing*, 1997.

[73] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[74] B. A. Galitsky, "Generalization of parse trees for iterative taxonomy learning," *Information Sciences*, vol. 329, pp. 125–143, 2016.

[75] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115.

[76] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[77] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," 2017.

[78] M. Harman, A. Lakhotia, and D. W. Binkley, "Theory and algorithms for slicing unstructured programs," *Information & Software Technology*, vol. 48, no. 7, pp. 549–565, 2006.