# Hardware-assisted Live Kernel Function Updating on Intel Platforms

Lei Zhou, Fengwei Zhang*, Kevin Leach, Xuhua Ding, Zhenyu Ning, Guojun Wang, Jidong Xiao

**Abstract**—Traditional kernel updates such as perfective maintenance and vulnerability patching requires shutting the system down, disrupting continuous execution of applications. Enterprises and researchers have proposed various live updating techniques to patch the kernel with lower downtime to reduce the loss of useful uptime. However, existing kernel live update techniques either rely on specific support from the target OS, or are deployed in virtualized environments (i.e., systems running in virtual machines). In this paper we present KSHOT, a hardware-assisted live and secure kernel function update mechanism for native operating systems. By leveraging x86 SMM and Intel SGX, KSHOT runs in hardware-assisted Trusted Execution Environments and updates kernel functions at the binary-level without relying on the underlying OS support. We demonstrate the applicability of KSHOT by successfully patching critical kernel vulnerabilities, upgrading base kernel functions and drivers nearly instantly and transparently. Our experimental results show that KSHOT incurs merely 70 microseconds downtime to update a one kilobyte binary and 18 MB memory overhead.

**Index Terms**—Kernel Function Updating, System Management Mode, Trusted Execution Environment, Consistency, Transparency.

## 1 Introduction

The growing complexity and heterogeneity of software has led to a concomitant increase in the pressure to apply **patches** and **updates** on operating systems [1]. For example, updating the Linux kernel functions from version 4.15 to 4.16 entailed 1309 commits over 18 revisions in two months. Frequently, users who choose to update their kernels may experience nontrivial downtime when the update requires restarting the system, even for a small change. However, high availability is an important requirement for many current services, especially in cloud services, which ideally must be accessible anytime without detectable downtime. To minimize the downtime, enterprises and users often delay applying updates to their operating systems, leading to increased risks to their computing resources [1], and postponing new functionality and features until a planned period of downtime to restart the system.

Since kernel functions updates are important to fixing vulnerabilities and adding software features, organizations often use rolling upgrades [2], [3], in which updates are designed to affect small subsystems that minimize unplanned whole-system downtime. However, rolling upgrades do not altogether obviate the need to restart software or reboot systems; instead, dynamic hot patching (live patching) approaches [4], [5], [6], [7], [8] aim to apply patches to running software without having to restart it.

Mainstream OSes like Windows and Linux have proposed several live updating technologies. For example, in Azure, hot patching is available in recent cloud server systems. Various Linux-based kernel live patching tools including kpatch [9], kGraft [10], Ksplice [11], and the Canonical Livepatch Service [12] have also been developed. These approaches are designed to redirect the execution flow from instructions within vulnerable kernel function to benign instructions by tracing, hooking, and trampolining the target function execution. However, all of these approaches require modifying the existing kernel code and trusting the underlying operating system. In a similar vein, some approaches like KUP [5] replace the whole kernel at runtime while retaining state from running applications. However, KUP incurs significant runtime and resource overheads (e.g., more than 30 GB of memory space) to support application checkpointing [13], even for very small kernel changes.

Existing patching implementations are known to be buggy [14] and may cause patching failures or interruptions. In the worst scenario, a patch may become a means of attack if the OS or the patching mechanism is compromised. For example, an internal OS update can be hijacked [15], [16], [17] to download and install malicious patches. Such attacks download additional malicious applications while retaining kernel functionality. It is thus imperative to harden live updating techniques. A secure live kernel function update faces three challenges:

1) **Downtime**. Traditional kernel updating methods require downtime, either from unplanned reboots or from stopping applications to checkpoint states.
2) **Overhead**. Live kernel function updating techniques often incur non-trivial CPU and memory overhead to apply patches and restore the previously-checkpointed state.

- *L. Zhou is with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen; and College of Computer at National University of Defense Technology, China.*
  *F. Zhang is with Department of Computer Science and Engineering, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China.*
  *K. Leach is with University of Vanderbilt University, USA.*
  *X. Ding is with Singapore Management University, Singapore.*
  *Z. Ning is with Hunan university, China.*
  *G. Wang is with Guangzhou University, China.*
  *J. Xiao is with Boise State University, USA.*

**The corresponding author; Email: zhangfw@sustech.edu.cn*

3) **Trust**. Live updating software depends on the correctness of the underlying OS, which may suffer from bugs [18] or security vulnerabilities. If the OS-level updating mechanism becomes compromised, then patches applied by that mechanism cannot be trusted.

In this paper, we present an enhanced hardware-assisted live kernel function updating that uses the Intel Software Guard eXtensions (SGX) and System Management Mode (SMM). We apply our prototype to effectively, efficiently, and reliably update running, untrusted kernels in native OSes. This paper is an extended version of our published conference work [1]. The original work verified the live patching framework in a virtual machine environment. The current work focuses on deploying the live patching framework in native systems for the addition of upgrading base kernel function and dynamic kernel modules like drivers (which remains named KSHOT), which addresses the new challenges associated with developing the system on bare machines while the trusted compute base— for example, the SMM handler is stored in firmware, which is a restricted or closed-source resource for commercial devices. We summarize our contributions as follows:

- We develop a reliable architecture for live updates to kernel functions. We leverage Trusted Execution Environments (TEEs) implemented with SGX and SMM features for kernel updating rather than deployed in a simulation environment which is designed in our previous work.
- We analyze patches, new functions, and dynamic kernel module updating requirements, and design classified updating strategies for different goals.
- We extend SMM to execute updating functions concurrently, which greatly improves live updating performance. We use SGX as a trusted environment for patch preparation to provide adequate runtime update performance. Furthermore, updating task information in an SGX enclave precludes adversarial tampering, and improves updating reliability.
- We evaluate the effectiveness and efficiency of KSHOT by providing an in-depth analysis of a suite of representative kernel updates. We demonstrate that our approach incurs little overhead while providing trustworthy live kernel function updates that improve the security and applicability of the live system.

## 2 BACKGROUND

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

### 2.1 Live Kernel Function Update

Live update enables OS upgrades without disrupting business-critical workloads, especially for real-time service platforms. It helps system administrators enforce live production systems and helps users achieve uninterrupted services, thereby increasing overall business productivity. Therefore, cloud service providers like Windows Azure [20] and IBM AIX [21] are rapidly developing live update services for their customers.

[1]Kshot [19], which has been published in IEEE/IFIP DSN 2020.

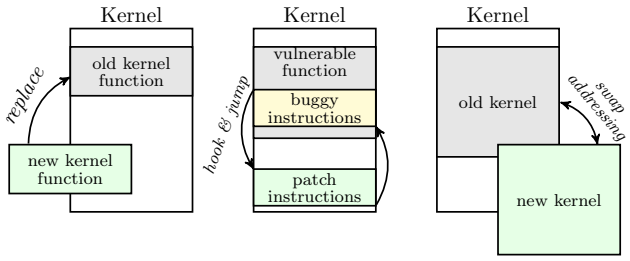



Fig. 1. Overview of live patching approaches—function-, instruction-, and kernel-level. In function-level, entire kernel functions are replaced with new ones by copying bytes into memory. In instruction-level, single buggy instructions are replaced with trampolines to new instructions. In kernel-level, 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.

Live update approaches update kernel code at three levels of abstraction: function replacement, instruction hooking and jumping, and kernel switching. Figure 1 illustrates common live kernel update methods across these three levels of abstraction. In general, live kernel patching apply changes by replacing single instructions, portions of vulnerable functions, or the entire kernel image with a patched version to repair bugs or eliminate vulnerabilities. For example, to live patch the kernel with a completely new version, KUP [5] can replace the old kernel image in memory. In more common cases, tools can update small amounts of code in the kernel memory, adding new functions or patching vulnerable ones. Solutions here include industry-deployed mechanisms like Ksplice [11] and kpatch [9], as well as academic tools like KUP [5] and KARMA [6].

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

### 2.2 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 functionalities such as power management, system hardware control, or OEM-specific code. SMM is typically 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) — the memory controller will only allow references to SMRAM when the CPU is in 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 code accesses the physical memory with a higher privilege, allowing it to read or modify kernel code and data structures in kernel memory segments. An SMI is triggered when the CPU executing the *out 0xb2, sid* instruction, where *0xb2* is an SMI triggering port on the hardware device (which may vary with the specific chipset), and *sid* represents a specific SMI handler identity in SMM. Alternatively, we can program the Interrupt Command Register in the Advanced Programmable Interrupt Controller (APIC) to generate an SMI. An SMI traps all cores in Protected Mode to SMM and passes the control to the SMI handler, which is well-suited for concurrently and securely handling updating operations.

### 2.3 Software Guard eXtensions

*Software Guard eXtensions* (SGX) [22] 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 increases 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.

## 3 SYNOPSIS

We consider the following service setting. A platform provider initializes the built-in SMM code for kernel update (denoted by the Trusted Update Agent, *TUA*), which also leverages SGX features for preparing patches (denoted by the Trusted Preparation Agent, *TPA*). A platform user (denoted by the *owner*) proposes the trusted kernel update within a potentially-compromised system with the help of the *TPA* and *TUA*.

### 3.1 Threat Model

Kernel-based updating mechanisms can become compromised by internal weaknesses [14], [18] or external attacks [17]. An adversary can obtain kernel privilege and manipulate or subvert the kernel update mechanism. For example, the 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). Note that, memory-level bug-fixing is the regular mechanism used in current Kernel Live Patching (KLP) approaches. The rootkits can be designed against the patching process and prevent physical memory-level bug repairing. In addition, the compromised system maintains an unstable runtime state which cannot ensure reliable update service and may cause the kernel to panic. Moreover, reactionary or insufficiently-verified updates may cause ongoing security risks. In the worst case, a stealthy adversary can install "legal" patches along with injecting malicious code for data leakage or as a secret backdoor. In that way, the user will continue to suffer security problems even after a kernel update.

### 3.2 Trust Model

First, we deploy our kernel live update approach on SGX- and SMM-enabled devices. We assume that the system is trusted during the boot stage, and that System Management RAM (Section 2.2) is locked by the system firmware so that an attacker cannot modify it (i.e., the hardware is trusted to enforce access control). Although SGX and SMM are potentially vulnerable to side-channel attacks like Crosstalk [23], and SMM Reload [24], 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 the operating system's patching mechanism. In addition, we assume that the source code of the patch is trusted. We note that KSHOT can be prevented by the compromised kernel since the update sources are from the user-level SGX enclave. However, this is not specific to our work (and, indeed, KSHOT can detect when Denial-of-Service (DOS) attacks occur with the SMM-based checking). If DOS attacks occur, we assume that a system operator in the loop would elect to take a victim system offline for subsequent manual updating.

### 3.3 Problem Statement and Our Approach

Our research problem is how to securely update the functions in a compromised OS under the aforementioned adversary model. The target functions include two types: base functions in monolithic kernel and dynamic loaded functions (e.g., loadable kernel modules, LKM). Functions in the base kernel are generally stored at the fixed physical addresses once the kernel is loaded at the boot stage. However, the LKMs are loaded into dynamically-allocated memory within the kernel.

A reliable and low-overhead kernel function live update is expected to meet the following requirements. (a) The update mechanism is not tampered with or faked by corrupted code inside the kernel (**trustworthiness**)). (b) The function update and revocation do not cause a kernel panic or false application execution (**consistency**). (c) The scheme should avoid high computational overheads induced by system interruption and context switch (**efficiency**).

KSHOT meets these requirements by using a novel combination of an SGX enclave within a helper application that securely downloads updated source code which is built and written to the kernel memory by a customized SMM Handler. In this way, the updating mechanism remains trusted since it does not rely on the untrusted kernel — the kernel cannot tamper with the update mechanism. In addition, the host system's kernel and applications are temporarily paused when trapping into the SMI handler, and this allows us to apply patches without causing the system to get into an inconsistent state. Moreover, when switching between the protected mode and the SMM mode, system states are automatically saved and restored by the hardware, and

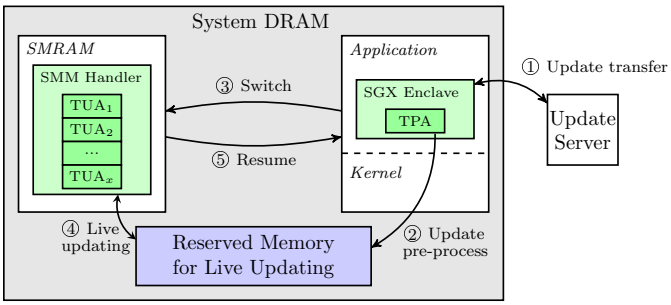



Fig. 2. High-level architecture of KSHOT. Our approach uses three secure entities: the Remote Update Server, the SGX enclave in a helper application with trust preparing agent (TPA), and the SMM-based trusted handler with multi-update agents (TUAs). The annotations 1–5 trace the life cycle of trusted kernel update. In (1), we transfer to the update module; in (2), we pre-process the binary code; in (3), we switch to SMM; in (4), we apply the update at the binary level; and in (5), we resume the updated OS.

such support from the hardware substantially reduces the overhead of runtime state saving, and thus enables rapid update deployment.

## 4 System Architecture

KSHOT is designed to achieve two goals while updating functions in the kernel, including (1) upgrading the base kernel function and loadable kernel modules to a designated version, (2) fixing vulnerable functions in both base kernel and loadable kernel modules. KSHOT implements a trustworthy download mechanism to acquire an update block with the help of an SGX enclave, then applies the update to the designated kernel memory, pausing the target system's execution. This novel combination of system features allows us to deploy updates with low runtime overhead, low latency, and without having to trust the underlying OS.

Figure 2 summarizes our approach. First, target system information required for compiling a compatible update binary is gathered and sent to the remote update server. Second, an SGX-based application fetches the update-required binary code from the remote update server and collects required updating information (e.g., patch function location addresses). The information is loaded into a reserved memory region to be processed by the SMM handler code. Third, we remotely trigger [25] an updating command, and 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 code in a reserved memory location (see Section 5), the update is applied so that the updated code will be executed on the next invocation once the SMM handler completes.

### 4.1 KSHOT Components

There are three main components in our KSHOT architecture: the remote update server, system-specific update preprocessing in the SGX enclave, and SMM-based kernel updating.

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

**SGX-based update preparation:** This component includes kernel information collection and binary update preprocessing. These processes take place in an SGX enclave as the trust preparing agent (*TPA*). The data transmitted between SGX and the update server, as well as between SGX and SMM, is authenticated to protect the update code from malicious changes. Leveraging SGX for preprocessing updates provides several benefits: First, the SGX enclave, functioning as a user-level trusted execution domain, can be verified by the user through Intel-supported remote and local attestation mechanisms [26]. This ensures that the user has confidence that the update code has been processed in a trusted environment. Second, *TPA* reduces the SMM workload and thus the time during which the OS is paused to execute the SMM handler. Third, *TPA* 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). Fourth, *TPA* maintains the confidentiality of the binary update code, which reduces readily-visible knowledge for a hypothetical attacker. Finally, because of the large semantic gap between SMM and the host environment [27], it is more natural to gather kernel information from the software layer within an SGX-enabled helper application.

**SMM-based kernel updating:** This component includes system information verification, data decryption, update function integrity checking, and binary updating. 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 updating approaches. Furthermore, since each core in SMM has an independently-executed interrupt handler, KSHOT can deploy multiple trust update agent (*TUA*) to process parallel update tasks in one system pausing interval, which effectively improves the performance of kernel updating. In addition, if a kernel error occurs after updating [18], *TUA* can undo the update and rollback the system. While the update operations are processed in SMM, the target OS is paused (which precludes simultaneous state changes). Because this activity is carried out with SMM support such that even kernel-level attacks cannot compromise the updating operations. In addition to an updating module, KSHOT can leverage a kernel introspection module for kernel protection.

### 4.2 Qualitative Analysis of KSHOT

We design KSHOT to enable reliable and efficient live kernel function updating from three perspectives: how the trustworthiness of update processing is preserved under a compromised OS; how the consistency of system execution is maintained during the live updating; and how the effectiveness of update handling is achieved via update preparation

and memory addressing. For the above perspectives, we leverage Intel SGX and SMM to build a trusted execution environment for update code pre-processing and update implementation. Meanwhile, the data (i.e., the update binary and configuration) transferred between the remote update server and the SGX enclave, and between the SGX enclave and the SMM handler, are signed with a secret signature and encrypted under an untrusted data transmission channel.

**Security of Updating Environment.** 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 updates. SMM is an isolated execution environment that cannot be accessed by host applications. In addition, no other interrupts are valid once all cores trap into SMM after an SMI is triggered. With the above hardware-assisted functional design, the SMM handler cannot be disrupted by kernel rootkits or malware. Since updating is implemented at the memory level with SMM handler, the update binary passed to the SMM handler from the user-level application will maintain confidentiality. With the help of a cryptography channel and an SGX-based execution environment, the ciphertext of the update binary is transferred to SMM handler via shared memory.

**Consistency of Runtime Execution.** The challenge of live updating is to ensure the execution status of the runtime system is consistent after updating. Unlike existing updating approaches, which require restarting the system and discarding the execution status data the traditional updating approaches, or implementing expensive process tracing and checkpointing mechanisms to restore execution status data, KSHOT temporarily halts the entire host system via switching to SMM. During this temporary system-wide halt, all kernel and user processes are suspended with no new status produced, while the SMM handlers can access the physical memory and reserved host register values for updating. By precisely locating the memory of a target kernel function to update, KSHOT creates a trampoline to the updated instance of that function without breaking the execution flow of other host processes. However, some complex updates (less than 2% in statistical cases of Linux kernel updates) may change the semantics of target functions, which might affect other non-patched functions. For example, the update might change the order in which locks are acquired in multiple functions at the same time. Currently, KSHOT cannot handle those cases independently but can address this problem with help of constructing a consistency model and safely choosing update tasks [9], [12].

**Efficiency Requirement.** First, switching to SMM pauses the host system and restores the architectural state once the SMM handler completes. We thus avoid implementing expensive process tracing and checkpointing mechanisms (as in kpatch or KUP), considerably reducing time and storage overhead. Current live patching approaches generally have to stop the process during update handling, but the basic kernel functions like CPU scheduling, and memory management keeps running. However, in KSHOT, the host system including kernel and user applications is paused once switched to SMM, which manifests as a stuck, frozen state for the user. To minimize the interval of time spent paused due to SMI execution, and to make it imperceptible to users, we propose to implement only the required functionality in SMM (i.e., memory read/write capabilities) to quickly deploy updates once they are made available to the SMM handler. Separately, we use an SGX enclave in user space to securely download the patch and marshal the update data into the SMM handler. This SGX enclave allows the update-required binary code 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 updates at runtime.

## 5 KSHOT DESIGN AND IMPLEMENTATION

The goal for KSHOT is to live update 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 update code, providing security without the full overhead of SMM. Encrypted update codes are processed in SMM and placed in an executable memory space. The system stores the state of runtime processes, restoring that state after applying the update when SMM completes. This allows for the deployment of a trusted binary via a possibly-compromised target system.

### 5.1 Update Preparation

We leverage a trusted remote server to prepare binary kernel update code. 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-update and post-update versions of the kernel binary using that same compilation information. A binary diff is sent back to the SGX enclave on the target machine. Note that, the information sent to the remote server may be compromised by OS. For example, the memory address of the pre-updated kernel module is tampered with by the adversary. To defend against such an attack, we incorporate a checking module within SMM to verify the encrypted information received from the enclave since the SMM handler can access the host memory.

We update the kernel by assessing the requirements of each kernel-level function that must be updated. For small updates in kernel functions, KSHOT patches the code by overwriting instructions within the function. For driver upgrade and downgrade, KSHOT updates the kernel function via memory-block interchange. KSHOT prepares the executable updated function binary and kernel module binary according to the target kernel configuration. For example, we compile the same kernel module $x.ko$ with functions changed and mark the library function in any call instructions which must be replaced with the real address once loaded into memory. These marked instructions are passed to the enclave in the target system together for further binary assembly.

#### 5.1.1 Identifying Target Functions

Given the pre- and post-update binary kernel image, we extract all corresponding updated functions. While this process is complicated by compiler optimizations [28], we do not claim any novelty in our identification of the functions that must be updated, instead we make use of a combination of existing algorithms and techniques. Our prototype builds a *source-level call graph* [29], [30] of the kernel by using the codeviz tool [31]. We also make use of IDA Pro [32] 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 [28], 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 be added. KSHOT makes use of existing binary signature matching methods such as iBinHunt [33] and FIBER [28] to align and identify relevant sections of the binary kernel image.

For the purposes of discussion and evaluation, we group implicated code into two categories: individual functions and modules. Individual functions have three broad types (of increasing difficulty to support via kernel live updating), including *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 updated function. Such a variable might be deleted, added, or modified. If the variable's size is not modified, the update 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-update 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 update application failures; we evaluate this empirically in Section 6. Individual modules are loosely coupled with other kernel functions, and they all export the their base addresses to a file under the /proc directory, and this allows us to easily address the modules.

#### 5.1.2 Updating Target Functions

After we identify and analyze all relevant target functions and modules, we must make the memory containing the (binary) newly-updated instructions accessible to the running kernel. In general, we cannot directly replace vulnerable function instruction memory with an updated function without compromising consistency. To solve this problem, we use *trampolines* (cf. [34]): We store the updated 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 update server. A basic trampoline approach addresses the call to the beginning of a function but does not address internal jumps or branches to intermediate labels. This is because the offset for each jump and branch in the post-update binary may have changed. Thus, we must change these offsets to retain the required functionality via the standard approach of calculating label differences.

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

#### 5.1.3 Supporting Kernel Tracing

Recent versions of the Linux kernel include a special form of tracing support [35] that is relevant to kernel live updating. When the trace attribute is enabled, more than half of the 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 updating). KSHOT must be aware of such tracing instructions to avoid conflicts. Naively updating 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 replace the instructions after them, leaving the tracing itself untouched.

### 5.2 SGX-based Update Preparation

KSHOT uses Intel SGX hardware support to safeguard trusted live update preprocessing. The preparation of executable binary update code proceeds in a trusted environment before the processed update is made available to the SMM-based live updating module. In this subsection, we describe our SGX enclave behavior. We encrypt communication when obtaining the binary update code from the remote server. This is also particularly relevant when passing data between the SMM handler and the 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 update 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.

#### 5.2.1 Memory Protection and Isolation

We first configure the boot loader (e.g., *grub*) to reserve a suitable kernel memory allocation space ($18$MB for our prototype implementation). We also add page attribute operation code to the *paging_init* function to provide the appropriate access permission control for that memory region. The reserved memory region includes three logical parts: *mem_RW*, *mem_W*, and *mem_X*. The small *mem_RW* is a read/write area used for identity exchange.

KSHOT uses a AES share key to encrypt the identity in our prototype. If improved to against the relay or MITM attack, we can leverage the SMM-based hardware feature to execute designed SGX enclave by refer the SMILE work [36].

The larger *mem_W* region is write-only and is used for storing the encrypted update 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 updated 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 [25], [37] to further ensure the integrity of this region.

KSHOT uses a shared AES key to encrypt the identity and update code following an authenticated *Diffie-Hellman (DH)* key exchange between the SMM handler and the enclave. The handler and the enclave have their public/private key pairs and share the public keys through the *mem_RW* mechanism. As a result of key exchange, both parties generate the same 128-bit AES key which is securely stored in SMRAM and EPC for exclusive accesses from the handler and the enclave, respectively. To defend against MITM attacks, we can leverage the SMM-based confined host environment to execute designed SGX enclave as in SMILE [36].

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 update code. 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 the correct memory location and alignment.

#### 5.2.2 Update Preprocessing

The SGX enclave receives an update code set from the remote update server $P = \{p_1, \ldots, p_n\}$, with edits to $n$ functions. An individual update $p_i$ has the form $\{sequence, opt, type, \ldots, payload\}$. The update preparation workflow follows a standard sequence of steps. First, we verify the integrity of the received update code to guard against network transmission errors. Next, the modified binary update code will be written out as an executable memory block. We package this memory block with external header information. 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 updating component.

### 5.3 SMM-based Live Updating

The CPU changes to System Management Mode when a System Management Interrupt is triggered. The SMM hardware ensures that the latest runtime state and register values are saved to the protected SMRAM region of memory. Before the updating, a *DH* key generation module is executed in SMM to create the private key, which is used to encrypt/decrypt the update code related data in SMM. This cryptographic key is dynamically changed before each kernel update 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 update server and thus mitigate the MITM attack. We implement the live updating process in the SMM handler, including integrity checking and the updating module itself.

#### 5.3.1 Update Mechanism Deployment

Current commercial Intel devices disable the SMM firmware for secure protection. Thus, accessing the SMM execution environment is a critical problem. We leverage the Intel debug connection interface technologies and reverse engineering to locate and modify the runtime SMI handler memory.

The SMRAM layout various with the motherboard type. OEMs often modify or extend the SMI handler to meet their own requirements. Therefore, the context and layout of the SMI handler differ from device to device. This means that the code and functionality implemented within the SMI handler can vary significantly. It is important for programmers and system designers to be aware of these variations in order to ensure compatibility and proper functioning of software across different devices and motherboard configurations.

We use the IDA and UEFI tools to decode the control flow of the SMI handler and locate suitable space to insert a new function in runtime SMRAM. In order to update a 64-bit function, we have to design the update handler in 64-bit. However, in SMM, the SMI handler does not execute the 64-bit instruction at first, it has to execute instructions in 16-bit mode first, then configure the control register and memory space, so as to switch to 64-bit mode and execute instructions in 64-bit mode. In that way, we have to address the 64-bit executable space in the original handler if we can not configure the mode due to lack of specification.

To ensure binary code updating, we design three main components to add within the SMI handler, including key generation, binary decryption, and verification. Key generation is built upon the Diffie-Hellman key exchange algorithm, which can exchange the 128-bit encrypted key without leaking each private key. Binary decryption is developed with the AES-NI instruction set that is also supported within SMM; this module is roughly 40 instructions. Binary verification is developed with a simple hash algorithm (SDBM). Since in a short session of exchanging binary data between the SGX enclave and SMM handler, current verification is secure enough to attest to the integrity of the updated binary code. At the end of the verification, those readily updated binary code is moved to the designated shared memory space.

In total, the update mechanism code is 2894 bytes in size, which can be inserted into the SMRAM. After that, we insert a trampoline instruction in the individual SMM execution code of each CPU core, which allows the updating threads to execute concurrently on all CPU cores.

#### 5.3.2 Vulnerable Kernel Function Patching

We define the location address of the patch function *paddr* at *mem_X*. The location address of the first updating function $p_1.paddr$ is the base address of *mem_X*. Then, the location address of the $i$th updating function is $p_i.paddr = p_{(i-1)}.paddr + p_{(i-1)}.size$, where $size$ denotes

the size of a binary update code. The binary update code $p_i$ 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 updating function once the vulnerable function is called (and respects the 5-byte kernel tracing setup).

#### 5.3.3 Base Kernel Function Upgrading

Besides fixing vulnerabilities, kernel patches oftentimes add or modify the code in the kernel that changes functionality. For example, the locking protocols in *md* should assume the device never be removed once it in *resync*, *recovery*, and *reshape* state, however, some cases like *state_store()* with such locking (by calling function $remove\_and\_add\_spares$ and $md\_check\_recovery$) is not needed, which only cause extra time overhead. Therefore, in the new kernel version, the kernel adds new conditional statements to abort the redundant execution. For such cases, KSHOT updates the function with the same operations as patching vulnerable functions discussed above. First, we add the execution binary of the updated function in $mem\_X$, then add the trampoline introduction at the first address of the old function.

However, there are more factors introduced in upgrade cases that increase the difficulty of live updating. For example, a kernel patch may export symbols and define new structures (similar to type 3 patches mentioned in Section 5). To apply such a patch in the memory, KSHOT changes the kernel *data* and *bss* sections, and such changes may lead to changes of the kernel's layout in the memory. A more flexible solution is to reserve a memory block for updating during the kernel initialization.

#### 5.3.4 Kernel Module Updating

Dynamically loadable kernel modules are another mechanism for function modification at the kernel level. Most device drivers are implemented as a loadable kernel module. The Linux system supports dynamic driver installation, and some other technologies like Dynamic Kernel Module Support (DKMS) to support the management of the driver update without recompilation. However, updating drivers inevitably pause user applications and, in the worst case, requires rebooting the system.

Unlike small changes to the base kernel, updating a kernel module typically involves much larger changes. For example, in *kvdo*, each update affects more than 20 files with hundreds of changed lines. To avoid the creation of an overly-bloated update binary, we apply the following steps to update kernel modules: First, we sign the kernel module information via an SGX enclave with a key passed from SMM. The key is used to encrypt messages — both the key and the messages are stored in the shared memory. Second, we install the new kernel module with the help of the untrusted kernel, which places the new code in a new address in memory. Third, we check the signed message in SMM and verify the kernel module in host memory. Fourth, we modify the device tree structure and point the old device to the new address. In brief, by using the existing untrusted update mechanism and verifying its work, we can save substantial effort and space when live updating kernel modules.

#### 5.3.5 Multi-tasks Updating

Recall that each core can execute independent SMM handlers in parallel. Meanwhile, each update task for different kernel updates is independent. Thus, KSHOT applies live updating operations across several cores to improve performance. As Figure 3 shows, we sort the update binary in shared memory. We marked each SMI handler in order and pick the corresponding package for updating. After the update operation, all cores return to protected mode simultaneously. Thus, the total time that the host system spends suspended depends on the longest update, but not the entire update operation. This significantly shortens the total time the system must spend paused in SMM.

### 5.4 Update Rollback/Update

After updating the kernel, the system or its applications may not run correctly for many different reasons [5]. For example, the update 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 [18]. Supporting rollback is thus critical for a realistic deployment. In such situations, we can send a rollback instruction from the remote server. The SMM handler rolls back the update function to the original function. We keep the updated 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 updating operation can always be rolled back in this manner.

### 5.5 Updating Protection

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

*Malicious Update Reversion*. Some latent attacks in a compromised OS might revert the update to 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 [25], [37] to prevent the target OS from reversion or modification by rootkits after applying

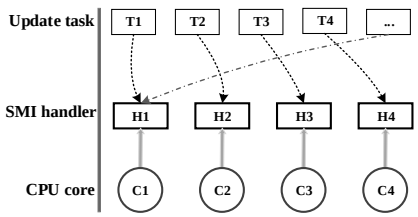



Fig. 3. The SMI handler parallel executes the update tasks.

the updating. We can similarly use the SMM handler to introspect regions of memory overwritten with trampoline instructions to ensure that the updated version of code persists after deploying a update 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 (DOS) attacks*. DOS attacks may preclude the update preparation operation from running, leading to a live updating failure. DOS attacks are generally difficult to defend [38], [39], however we can detect DOS attacks using SMM-based introspection techniques. After the remote server sends the update code source to the SGX Enclave in the target OS, the enclave and the remote server can communicate the state of the update preparation. Once the update binary is written in to the reserved memory, the remote server can verify with the SMM Handler that the update binary was written to memory (i.e., via introspection in the SMM Handler). This approach cannot prevent DOS attacks but can detect them.

## 6 Evaluation

We evaluate the applicability, performance, and security of KSHOT when live updating Linux kernel functions. Our prototype machine uses an Intel Core i7 CPU (supporting SGX and SMM) with 16GB of memory. We use the x86 GA-Q170M-D3H motherboard with F21 BIOS. We experiment with Ubuntu 14.04 to 20.04 using kernel versions 3.14 to 5.4.1.

We consider three research questions:

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

### 6.1 RQ1 — Correct Kernel Updating

We evaluated KSHOT's ability to live update the kernel in three scenarios: (1) patch critical kernel vulnerabilities by using a suite of real-world patches from the Common Vulnerabilities and Exposures (CVE) database [40]; (2) upgrade critical functions changed in a kernel version update; (3) replace kernel modules following user requirements. For scenario 1, we analyzed 267 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 updating framework. For scenario 2, we analyzed the patches according to changes required, including extension, addition, and deletion of functions. We chose several representative cases to show how our kernel live update approach applies. For scenario 3, we analyzed user requirements to update kernel modules including new I/O device drivers and other LKMs.

#### 6.1.1 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 randomly selected 30 of those 214 patches to construct a benchmark suite similar in scale to existing work [6], [41]. The selected patches are listed in previous work [19]. To provide additional insight into our successful applicability results, we detail a few patches as case studies. Recall from Section 5.1 that we can classify each kernel patch into one of three categories. Type 1 patches involve no inlining and thus have their 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.

**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;
```

**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 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** We 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.

**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.

#### 6.1.2 Kernel Function Upgrade

In our test, we do not implement a complete upgrade but choose parts of critical cases for evaluation. For example, previous patches for the *objtool* enable it to read the *retpoline* alternatives but leave a bug in the patched function which may cause an incorrect handling at the end of *retpoline* alternatives. The kernel version 4.16 upgrades this function, and the official update is partially shown in Listing 4. However, this does not mean we target the defective function directly, the final update function segment is up to the optimization options upon compiling, e.g., the *handle_group_alt* function will be complied with inlined operations (-O2 optimization option) and finally be inserted to the caller function *check()* as parts of the instruction segment (87 instructions), then we can select compatible binary code for the next live updating.

**Listing 4** Upgrade case: add support for objtool

```
  static int handle_group_alt(struct objtool_file *file,
    ...
-   return -1;
+   if(next_insn_same_sec(file, last_orig_insn)){
+    fake_jump = malloc(sizeof(*fake_jump));
+    if(!fake_jump){
   ...
+   *new_insn=fake_jump;
    return 0;
```

We consider another kernel function upgrade example in Linux kernel 5.4.1. The block storage function *nbd_add_socket* in the previous version of the network kernel driver has a vulnerability that can cause a memory leakage. Fortunately, the corresponding repair method in the source code is simple, and the official update is partially shown in Listing 5. In this update, the patch only changes the order of the function workflow. While in KSHOT, the updated binary does not add or delete any binary code but only changes the order of the instructions, which introduces minimal impact on the runtime system.

**Listing 5** Upgrade case: nbd-prevent memory leak

```
  static int nbd_add_socket(struct nbd_device *nbd,
       unsinged long arg,
    sockfd_put(sock);
    return -ENOMEM;
   }
+
+ config->socks =socks;
+
   nsock=kzalloc(sizeof(struct nbd_sock),GFP_KERNEL);
  if(!nsock) {
    sockfd_put(sock);
    return -ENOMEM;
  }
-
- config->socks =socks;
-
   nsock->fallback_index =-1
   ...
```

#### 6.1.3 Kernel Module Update

For user requirements, we consider the functionality of a loadable driver and system call module — e.g., one of the famous loadable drivers is the open-source NVIDIA GPU driver. We can dynamically update those modules without suspending the user application. Since the loadable kernel module is designed for many goals (a partial list is shown in Table 3), we choose those use cases to show how to live update functions in the kernel module.

Updating a loadable kernel module is more flexible to implement because most modules have no influence on the base kernel functionality even when the LKM's functions change. Furthermore, we can replace the whole module function in new memory segments, and we can update the kernel module in SMM without restarting the user application.

TABLE 1
Breakdown of SGX operations ($\mu s$).

| Binary Size | Fetching | Code Verifying | Pre-processing | Total |
|---|---|---|---|---|
| 40B | 16.51 | 11.22 | 6.75 | 34.48 |
| 400B | 17.48 | 12.39 | 6.94 | 36.81 |
| 4KB | 30.57 | 21.58 | 31.69 | 83.84 |
| 1MB | 5,833.81 | 2,503.35 | 2,775.94 | 11,113.1 |
| 4MB | 23,366.51 | 9,822.55 | 11,510.43 | 44,699.49 |
| 16MB | 93,190.15 | 38,817.51 | 46,039.29 | 178,046.95 |

### 6.2 RQ2 — Performance Evaluation

To evaluate the performance of KSHOT, we measured each stage of the live updating process. We consider overhead from two sources: SGX-based binary preparation and SMM-based updating. Since the SMM updating 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 code generally ranged from 40 bytes to 4 KB.

#### 6.2.1 SGX-Based Update Preparation Performance

The SGX enclave must (1) fetch the update code from the remote server, (2) preprocess the update code through integrity checking and branch instruction replacing, and passing the code with encryption and writing to the shared memory region for consumption by the SMM side. We evaluate the time consumption in each step.

Table 1 shows a breakdown of the time consumed by this SGX-based update binary preparation for various binary sizes, averaged over 100 trials. Consider the 4 KB case as an example. The time to fetch a binary code from our remote server is 30.57 $\mu s$, and the time to verify the code is 21.58 $\mu s$. In addition, 31.69 $\mu s$ is required to store the encrypted binary code into the shared memory region. All told, we use 83.84 $\mu s$ to complete the preprocessing of a 4 KB update.

#### 6.2.2 SMM-Based Updating Performance

The SMM handler pauses the target OS while carrying out key generation, data reading and decryption, binary verification, and binary code activities. In addition, there are overheads associated with switching between the SMM mode and the protected mode. We evaluate these times

TABLE 2
Breakdown of SMM operations ($\mu s$).

| Binary Size [1] | Binary Decryption | Binary Verification | Total |
|---|---|---|---|
| 40B | 14.39 | 0.41 | 15.34 |
| 400B | 15.04 | 1.27 | 16.31 |
| 4KB | 16.11 | 9.38 | 25.49 |
| 1MB | 344.64 | 2,376.61 | 2,721.25 |
| 4MB | 1,336.93 | 9,481.82 | 10,818.75 |
| 16MB | 5,308.85 | 37,900.85 | 43,209.70 |

[1] includes key generation but not includes SMM switching time.

empirically using the $rdtsc$ instruction to count the number of CPU cycles elapsed during each operation.

Since our SMM-based updating code is inserted in the original SMM handler code, SMM switch and original handler execution take an average of 29.1 $\mu s$ in our experimental platform. These values depend on specific hardware configuration and workflow of the SMM handler but are typically on the same order of magnitude in our experience. Once we switch to SMM, we spend 14.3 $\mu s$ to generate encryption keys. The switching operation and key generation are fixed-cost operations, regardless of update binary size.

The SMM handler reads the encrypted code provided by the SGX enclave, then applies it to the kernel memory. The time taken to read, decrypt, and apply the update code depends on the binary size. We tested binary sizes ranging from 40 bytes to 10 MB. Table 2 shows the time breakdown of updating operations for various binary sizes. For example, a 4 KB binary takes 16.11 $\mu s$ to read and decrypt, and 9.38 $\mu s$ to verify and apply to kernel memory (e.g., to actually write the update function to memory).

The overhead grows approximately linearly with the binary size. Even in the case of a large 16 MB binary, the total required time is under 0.2 s. On average, the updates from our CVE dataset are less than 1 KB, and from the general kernel module are less than 10 MB. Note that we did not count the overhead imposed by communication between the update Server and target Machine's untrusted helper application, which has minimal effect on the SGX enclave. Extrapolating from Table 2, the average update thus requires roughly 35 $ms$. We view this as a small and acceptable time interval to pause the system, especially given the rarity of live updating events.

#### 6.2.3 Whole-System Performance Evaluation

First, we consider the kernel function update. We randomly selected 5 of our benchmarks for a detailed analysis of whole-system performance.[2] In addition to the update binary code itself, each function requires 42 bytes of header data in the transmitted binary package. Figure 4 shows the time breakdown in the SGX preprocessing stage, which indicates the majority of time is spent preprocessing the update according. Similarly, Figure 5 shows the time breakdown in SMM for each update. Larger updates require more updating time, while the switching and key generation times are relatively constant across all updating. In these whole-system experiments, KSHOT required very little time to apply each update. For example, for *CVE-2014-4608*, the total time required on the Target Machine was about 140 $\mu s$ for a 156-bytes binary, The system is only paused for a brief 71.33 $\mu s$ during SMM activities. This includes 14.8 $\mu s$ for key generation and 29.1 $\mu s$ for SMM switching. The update was completed successfully, without changing the application state.

Second, we consider large updates to kernel modules. We halt and replace a whole vulnerable module in the kernel at one time. For example, we choose a representative Linux kernel module application including devices, filesystems, and network drivers, listed in Table 3. While small kernel module updates (less than 1 KB takes similar overhead to updating a single kernel function, larger module updates (larger than 1 MB) take more than 10 ms.

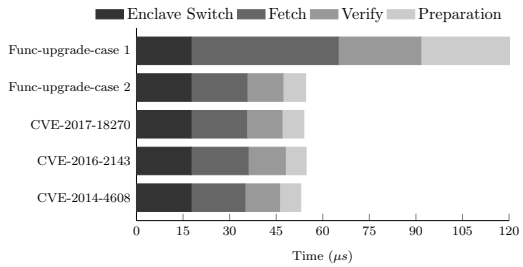



Fig. 4. SGX-based binary preparation time.

### 6.3 RQ3 — Updating System Comparison

Existing kernel live updating systems assume that update binary code are trusted when they are stored in the target OS. However, the integrity of the binary can be easily compromised by attacks which have kernel access privilege (e.g., syscall_hijacking [42]). By contrast, KSHOT leverages the SGX enclave to preprocess and apply binary updates without having to trust the underlying OS. Additionally, data blocks transmitted between SGX and SMM through the shared memory are encrypted to protect the binary's integrity from malicious modification during preprocessing.

In addition, existing solutions rely on kernel-specific functions to implement the updating operations (e.g., *ptrace, stop_machine, kexec*). However, existing vulnerabilities [40], 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

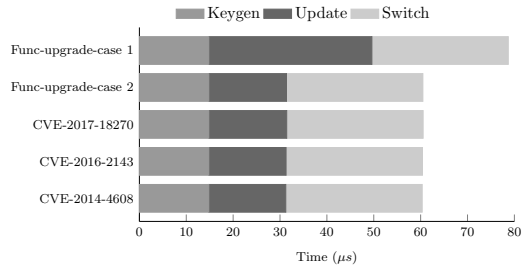



Fig. 5. SMM-based live updating time.

[2]patches for CVE-2014-4608, CVE-2016-2143, CVE-2017-18270; corresponding patch sizes: 198, 257, 322 bytes. Also, we use the function upgrade case 1 and 2 in section 6.1.2, the corresponding sizes: 9717, 384 bytes.

TABLE 3
Kernel Module Update Cases. ($\mu s$; $n = 100$).

| Functionality type | Example case | Partially Changed function | Update Binary size (bytes) | Preparation overhead | Update overhead | Update valid |
|---|---|---|---|---|---|---|
| Device driver | gpu-modules | kernel_gsp | 0x204000 | 17,650.18 | 5,458.87 | ✓ |
| Filesystem driver | tarfs | tarfs_file_read | 0xd79 | 74.67 | 38.50 | ✓ |
| System call | reveng_rtkit | unprotect_memory | 0x21e | 43.89 | 31.25 | ✓ |
| Network driver | realtek_r8152 | bottom_half | 0x2f359 | 1,649.89 | 528.30 | ✓ |
| Storage driver | kvdo | getPageNoStats | 0x40c97 | 2,227.47 | 713.42 | ✓ |
| Useful interpreter | keylogger | read_dev | 0x5db | 63.28 | 33.23 | ✓ |

TABLE 4
Comparison with kernel updating systems.

| | Type | Downtime | Untrusted OS | Memory |
|---|---|---|---|---|
| KUP [5] | kernel | 3s/kernel | ✗ | >30G |
| KARMA [6] | instruction | $5\mu s$/patch[1] | ✗ | lua engine |
| kpatch [9] | function | 45.6ms/patch[1] | ✗ | 16G |
| KSHOT | function | $70\mu s$/update[1] | ✓ | 18M |

[1] for an averaged sized update code of less than 1KB

updating mechanism. In KSHOT, live updating 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 binary prevents a compromised OS from interfering with KSHOT.

We compare KSHOT with representative kernel live updating methods (including KUP, KARMA, kpatch) in Table 4 in terms of patch granularity, updating time, trusted code base, and memory consumption. KUP replaces an entire vulnerable kernel in around 3 seconds. Additionally, KUP can handle update images with complex data structure changes. KARMA requires less than 5 $\mu s$ 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 18 MB extra memory space for update binary analysis and management. Also, KSHOT requires only about 70 $\mu s$ to deploy most updating, which is faster than all existing non-instruction-level methods. Overall, our approach provides an efficient and secure live updating mechanism.

### 6.4 Evaluation Summary

We find that KSHOT is a generic, performant, secure approach to live updating Linux kernel functions. Through 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 binary preparation and deployment, KSHOT incurs under 3% total system overhead over $1,000$ live updates. Finally, this approach requires a substantially smaller TCB compared to previous techniques.

To put these results in context, we discuss two of our kernel function updates with respect to time from vulnerability discovery to fix. First, CVE-2014-8133 was first discovered on 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 on 17 October 2017, with a corresponding patch built on 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 [43]. However, even when a patch is created, it may take additional time for end users to adopt the new patch [44]—many successful exploits rely on old, previously-patched vulnerabilities [45]. Live updating 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 updating capabilities.

## 7 Related Work

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

### 7.1 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 [46], Intel SGX [47], [48], AMD memory encryption technology [49], and ARM TrustZone [50]. HyperCheck [25] leverages SMM to build a trusted execution environment and monitor hypervisor integrity. VC3 [51] leverages Intel SGX to provide an isolated region for secure big data computation. Scotch [52] combines x86 SMM and Intel SGX to monitor cloud resource usage. KSHOT uses a TEE for reliable kernel live patching.

### 7.2 Live Patching

Existing live patching techniques focus on open-source operating systems, mainly Linux. For example, Ksplice [11], kpatch [9], and kGraft [10] 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 [5]. To address this limitation, KUP [5] replaces the whole kernel with an updated 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 [6] uses a kernel module to replace vulnerable instructions that it identifies from a given patch diff file. In addition, several live updating methods have been integrated into operating systems, like Canonical Livepatch Service [12] in Ubuntu, 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 update the target kernel function with a smaller TCB and low total overhead.

## 8 CONCLUSIONS

In this paper, we presented KSHOT, a secure and efficient framework for kernel updating. It leverages x86 SMM and Intel SGX to update 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 updating 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 70 $\mu$s for a 1 KB binary kernel function update, but consumes only 18 MB of extra memory space for binary analysis, a substantial reduction over previous work. In addition, for 1MB kernel module updating, it takes about 10 ms which is unnoticeable for users.

**Acknowledgments**. This work is partly supported by the Shenzhen Science and Technology Program (No.SGDX20201103095408029), the National Natural Science Foundation of China (No.62002151, No.62102175), and the National Key Research and Development Program of China (No.2020YFB1005804).

## 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] C. Niesler, S. Surminski, and L. Davi, "HERA: Hotpatching of embedded real-time applications." in *2021 Network and Distributed Systems Security (NDSS'21)*, 2021.

[8] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "RapidPatch: Firmware hotpatching for Real-Time embedded devices," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

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

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

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

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

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

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

[15] 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.

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

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

[18] 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

[19] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang, "KShot: Live kernel patching with SMM and SGX," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 1–13.

[20] Microsoft, "Hotpatch for Windows Server Azure Edition," https://learn.microsoft.com/en-us/azure/automanage/automanage-hotpatch, 2022.

[21] I. AIX, "AIX upgrade without reboot, zero downtime (AIX Live update)," https://www.ibm.com/docs/en/aix/7.2?topic=updates-live-update, 2022.

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

[23] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021, pp. 1852–1867.

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

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

[26] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[27] 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.

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

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

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

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

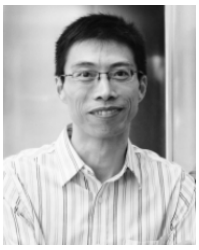
[32] H. Rays, "IDA Tools," https://www.hex-rays.com, 2018.

[33] 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.
[34] 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.
[35] S. Rostedt, "Ftrace Linux Kernel Tracing," in *Linux Conference Japan*, 2010.
[36] L. Zhou, X. Ding, and F. Zhang, "Smile: Secure memory introspection for live enclave," in *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2022, pp. 1536–1536.
[37] 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.
[38] 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.
[39] 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.
[40] MITRE CVE Team, "CVE Details: The ultimate security vulnerability datasource," https://www.cvedetails.com/, 2019.
[41] 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.
[42] GitHub, "Syscall Hijacking on Linux Kernel," https://github.com/crudbug/simple-rootkit/, 2014.
[43] Rapid7, https://blog.rapid7.com/2018/08/22/whats-going-on-in-production-application-security-2018/, August 2018.
[44] 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.
[45] 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.
[46] 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
[47] 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.
[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] 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.

**Lei Zhou** received the PhD degree in Computer Science from Central South University. He is a Research Associate in the College of Computer at National University of Defense Technology. His primary research interests are in the areas of x86 systems security, including trustworthy execution, hardware-assisted security, and memory forensics.

**Fengwei Zhang** received the PhD degree in Computer Science from the George Mason University. He is an Associate Professor in the Department of Computer Science and Engineering at Southern University of Science and Technology (SUSTech). His primary research interests are in the areas of systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.

**Kevin Leach** received the PhD in Computer Engineering at the University of Virginia. He is an Assistant Professor of Computer Science at Vanderbilt University. His work is in systems security, specifically the debugging transparency problem, though occasionally work on conversational artificial intelligence, program analysis, medical informatics, and big data applications.



**Xuhua Ding** received the PhD in Computer Science from the University of Southern California. He is an Associate Professor in the School of Computing and Information Systems at Singapore Management University. His research interests are in Network and system security, Applied cryptography,Trustworthy systems for data protection, to design the trustworthy systems in commodity x86 and ARM platforms to counter kernel space attacks.

**Zhenyu Ning** received the PhD in Computer Science from the Wayne State University. He is an Associate Professor at the College of Computer Science and Electronic Engineering at Hunan University. His primary research interests are in different areas of security and privacy, including system security, mobile security, IoT security, transportation security, trusted execution environment, and hardware-assisted security.

**Guojun Wang** received the PhD in Computer Science from Central South University. He is a Professor in the School of Computer Science and Cyber Engineering at Guangzhou University, China, His research interests include artificial intelligence, big data, cloud computing, Internet of Things, blockchain, trustworthy/dependable computing, network security, privacy preserving, recommendation systems, smart cities, and medical information systems.

**Jidong Xiao** received the PhD degree in Computer Science from the College of William and Mary. He is an Assistant Professor in the Department of Computer Science at Boise State University. His research interests are mainly in cyber security, with a particular emphasis on operating system security and virtualization/cloud security. He also has approximately 6 years industry experience, including various roles at Intel, Symantec, Nokia, and Juniper.