SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization

Zhe Wang[1,2], Chenggang Wu[1,2*], Yinqian Zhang[3], Bowen Tang[1,2], Pen-Chung Yew[4], Mengyao Xie[1,2], Yuanming Lai[1,2], Yan Kang[1,2], Yueqiang Cheng[5], and Zhiping Shi[6]

[1]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences,*
[2]*University of Chinese Academy of Sciences,* [3]*The Ohio State University,*
[4]*University of Minnesota at Twin-Cities,* [5]*Baidu USA,* [6]*The Capital Normal University*

Abstract

Information hiding (IH) is an important building block for many defenses against code reuse attacks, such as code-pointer integrity (CPI), control-flow integrity (CFI) and fine-grained code (re-)randomization, because of its effectiveness and performance. It employs randomization to probabilistically "hide" sensitive memory areas, called safe areas, from attackers and ensures their addresses are not leaked by any pointers directly. These defenses used safe areas to protect their critical data, such as jump targets and randomization secrets. However, recent works have shown that IH is vulnerable to various attacks.

In this paper, we propose a new IH technique called SafeHidden. It continuously re-randomizes the locations of safe areas and thus prevents the attackers from probing and inferring the memory layout to find its location. A new thread-private memory mechanism is proposed to isolate the thread-local safe areas and prevent adversaries from reducing the randomization entropy. It also randomizes the safe areas after the TLB misses to prevent attackers from inferring the address of safe areas using cache side-channels. Existing IH-based defenses can utilize SafeHidden directly without any change. Our experiments show that SafeHidden not only prevents existing attacks effectively but also incurs low performance overhead.

1 Introduction

Information hiding (IH) is a software-based security technique, which hides a memory block (called "safe area") by randomly placing it into a very large virtual address space, so that memory hijacking attacks relying on the data inside the safe area cannot be performed. As all memory pointers pointing to this area are ensured to be concealed, attackers could not reuse existing pointers to access the safe area. Moreover, because the virtual address space is huge and mostly inaccessible by attackers, the high randomization entropy makes brute-force probing attacks [45, 47] very difficult to succeed without crashing the program. Due to its effectiveness and efficiency, IH technique has become an important building block for many defenses against code reuse attacks. Many prominent defense methods, such as code-pointer integrity (CPI), control-flow integrity (CFI) and fine-grained code (re-)randomization, rely on IH to protect their critical data. For example, O-CFI [40] uses IH to protect all targets of indirect control transfer instructions; CPI [30] uses IH to protect all sensitive pointers; ReRanz [57], Shuffler [59], Oxymoron [4], Isomeron [15] and ALSR-Guard [36] use IH to protect the randomization secrets.

For a long time, IH was considered very effective. However, recent advances of software attacks [20, 35, 19, 43, 22] have made it vulnerable again. Some of these attacks use special system features to avoid system crashes when scanning the memory space [19, 35]; some propose new techniques to gauge the unmapped regions and infer the location of a safe area [43]; some exploit the thread-local implementation of safe areas, and propose to duplicate safe areas by using a thread spraying technique to increase the probability of successful probes [20]; others suggest that cache-based side-channel attacks can be used to infer the location of safe areas [22]. These attacks have fundamentally questioned the security promises offered by IH, and severely threatened the security defenses that rely on IH techniques.

To counter these attacks, this paper proposes a new information hiding technique, which we call SafeHidden. Our key observation is as follows: The security of IH techniques relies on (1) a high entropy of the location of the safe areas, and (2) the assumption that no attacks can reduce the entropy without being detected. Prior IH techniques have failed because they solely rely on the program crashes to detect attacks, but recent attacks have devised novel methods to reduce entropy without crashing the programs.

SafeHidden avoids these design pitfalls. It mediates all types of probes that may leak the locations of the safe areas, triggers a re-randomization of the safe areas upon detecting

*To whom correspondence should be addressed.

legal but suspicious probes, isolates the *thread-local* safe areas to maintain the high entropy, and raises security alarms when illegal probes are detected. To differentiate accidental accesses to unmapped memory areas and illegal probing of safe areas, SafeHidden converts safe areas into *trap areas* after each re-randomization, creating a number of trap areas after a sequence of re-randomization operations. Accesses to any of these trap areas are captured and flagged by SafeHidden. SafeHidden is secure because it guarantees that *any attempt to reduce the entropy of the safe areas' locations either lead to a re-randomization (restoring the randomness) or a security alarm (detecting the attack).*

SafeHidden is designed as a loadable kernel module, which is self-contained and can be transparently integrated with existing software defense methods (e.g., CPI and CFI). The design and implementation of SafeHidden entail several unconventional techniques: First, to mediate all system events that may potentially lead to the disclosure of safe area locations, SafeHidden needs to intercept all system call interfaces, memory access instructions, and TLB miss events that may be exploited by attackers to learn the virtual addresses of the safe areas. Particularly interesting is how SafeHidden traps TLB miss events: It sets the reserved bits of the page table entries (PTE) of the safe area so that all relevant TLB miss events are trapped into the page fault handler. However, because randomizing safe areas also invalidates the corresponding TLB entries, subsequent benign safe area accesses will incur TLB misses, which may trigger another randomization. To address this challenge, after re-randomizing the safe areas, SafeHidden utilizes hardware transactional memory (i.e., Intel TSX [2]) to determine which TLB entries were loaded before re-randomization and preload these entries to avoid future TLB misses.

Detecting TLB misses is further complicated by a new kernel feature called kernel page table isolation (KPTI) [1]. Because KPTI separates kernel page tables from user-space page tables, TLB entries preloaded in the kernel cannot be used by the user-space code. To address this challenge, SafeHidden proposes a novel method to temporarily use user-mode PCIDs in the kernel mode. To prevent the Meltdown attack (the reason that KPTI is used), it also flushes all kernel mappings of newly introduced pages from TLBs.

Second, SafeHidden proposes to isolate the *thread-local* safe area (by placing it in the *thread-private* memory) to prevent the attackers from reducing its randomization entropy. Unlike conventional approaches to achieve *thread-private* memory, SafeHidden leverages hardware-assisted *extended page table* (EPT) [2]. It assigns an EPT to each thread; the physical pages in other threads' *thread-local* safe area are configured not accessible in current thread's EPT. Compared to existing methods, this method does not need any modification of kernel source code, thus facilitating adoption.

To summarize, this paper makes the following contributions to software security:

- It proposes the re-randomization based IH technique to protect the safe areas against all known attacks.
- It introduces the use of *thread-private* memory to isolate *thread-local* safe areas. The construction of *thread-private* memory using hardware-assisted *extended page tables* is also proposed for the first time.
- It devises a new technique to detect TLB misses, which is the key trait of cache side-channel attacks against the locations of the safe areas.
- It develops a novel technique to integrate SafeHidden with KPTI, which may be of independent interest to system researchers.
- It implements and evaluates a prototype of SafeHidden, and demonstrates its effectiveness and efficiency through extensive experiments.

The rest of the paper is organized as follows. Section 2 reviews information hiding techniques and existing attacks. Section 3 explains the threat model. Section 4 presents the core design of SafeHidden. Section 5 details the implementation of SafeHidden. Section 6 provides the security and the performance evaluation. Discussion, related work, and conclusion are provided in Section 7, 8 and 9.

2 Background and Motivation

2.1 Information Hiding

Information hiding (IH) technique is a simple and efficient isolation defense to protect the data stored in a safe area. It places the safe area at a random location in a very large virtual address space. It makes sure that no pointer pointing to the safe area exists in the regular memory space, hence, making it unlikely for attackers to find the locations of the safe areas through pointers. Instead, normal accesses to the safe area are all done through an offset from a dedicated register.

Table 1 lists some of the defenses using the IH technique. The column "TL" shows whether the safe area is used only by its own thread or by all threads. The column "AF" shows how frequent the code accesses the safe area. Because most accesses to the safe area are through indirect/direct control transfer instructions, their frequencies are usually quite high. The column "Content in protected objects" shows the critical data tried to protect in safe areas. The column "Reg" shows the designated register used to store the (original) base address of the safe area. Some of them use the x86 segmentation register `%fs`/`%gs`. Others use the stack pointer register on X86_64, `%rsp`, that originally points to the top of the stack. They access a safe area via an offset from those registers. For the `%gs` register, they often use the following formats: *%gs:0x10*, *%gs:(%rax)*, *%gs:0x10(%rax)*, etc. For the `%rsp` register, they often use the following formats: *0x10(%rsp)*, *(%rsp, %rax, 0x8)*, *pushq %rax* [1], etc.

[1]It still conforms to the access model. The designated register is %rsp,

Defense	**Protected Objects**	**Reg**	**Content in Protected Objects**	**AF**	**TL**
O-CFI [40]	Bounds Lookup Table	%gs	The address boundaries of basic blocks targeted by an indirect branch instruction.	High	No
ReRanz [57]	Real Return Address Table	%gs	The table that contains the return addresses pushed by call instructions.	High	Yes
Isomeron [15]	Execution diversifier data	%gs	The mapping from the randomized code to the original code.	Hign	No
ASLR-Guard[36]	AG-Stack	*%rsp*	Dynamic code locators stored on the stack, such as return addresses.	High	Yes
	Safe-stack	%gs	ELF section remapping information and the key of code locator encryption.	High	No
Oxymoron [4]	Randomization-agnostic translation table	%fs	The translation table that contains the assigned indexes that are used to replace all references to code and data.	High	No
Shuffler [59]	Code pointer table	%gs	The table that contains all indexes that are transformed from all function pointers at their initialization points.	High	No
CFCI [61]	Protected Memory	%gs	File name and descriptors, and the mapping between file names and file descriptors.	Low	No
CPI [30]	Safe Stack	*%rsp*	Return address, spilled register, and objects accessed within the function through the stack pointer register with a constant offset.	High	Yes
	Safe Pointer Store	%gs	Sensitive pointers and the bounds of target objects pointed by these pointers.	High	No

Table 1: The list of defenses using information hiding (IH) techniques. **AF** is short for *Access Frequency*. **TL** is short for *Thread Local*.

A safe area is usually designed to be very small. For example, the size of a safe area shown in Table 1 is usually limited to be within 8 MB in practice. On today's mainstream X86_64 CPUs, the randomization entropy of an 8 MB safe area is 2^{24}. Such a high randomization entropy makes brute force probing attacks [45, 47] hard to guess its location successfully. A failed guess will result in a crash and detected by administrators.

2.2 Attacks against Information Hiding

Recent researches have shown that the IH technique is vulnerable to attacks. To locate a safe area, attackers may either improve the memory scanning technique to avoid crashes, or trigger the defense's legal access to the safe area and infer its virtual address using side-channels.

2.2.1 Memory Scanning

The attackers could avoid crashes during their brute-force probing. For example, some adversaries have discovered that some daemon web servers have such features. The daemon servers can fork worker processes that inherit the memory layout. If a worker process crashes, a new worker process will be forked. This enables the so-called *clone-probing* attacks where an adversary repeatedly probes different clones in order to scan the target memory regions [35]. CROP [19] chooses to use the exception handling mechanism to avoid crashes. During the probing, an access violation will occur when an inadmissible address is accessed. But, it can be captured by an exception handler instead of crashing the system.

Attackers could also use memory management APIs to infer the memory allocation information, and then locate the safe area. In [43], it leverages the *allocation oracles* to obtain the location of a safe area. In a user's memory space, there are many unmapped areas that are separated by code and data areas. To gauge the size of the largest unmapped area, it uses a binary search method to find the exact size by allocating and freeing a memory region repeatedly. After getting the exact size, it will allocate the memory in this area through the *persistent allocation oracle*. It then uses the same method to gauge the second largest unmapped area. Because a safe area is mostly placed in an unmapped area, an attacker can probe its surrounding areas to find its location without causing exceptions or crashes.

All probing attacks need to use such covert techniques to probe the memory many times without causing crashes because the size of a user's memory space is very large. In [20], it finds the safe area in many defenses is thread local (see Table 1). So, it proposes to leverage the thread "spraying" technique to "spray" a large number of safe areas to reduce the number of probings. After spraying, the attackers only need very few probes to locate the safe area.

2.2.2 Cache-based Side-Channel Attacks

To translate a virtual address to a physical address, the MMU initiates a page table (PT) walk that visits each level of the page table sequentially in the memory. To reduce the latency, most-recently accessed page table entries are stored in a special hardware cache, called *translation lookaside buffer* (TLB). Because of the large virtual address space in 64-bit architectures, a hierarchy of cache memories has been used to support different levels of page-table lookup. They are called the page table caches, or *paging-structure cache*s by Intel [2]. In addition, the accessed PT entries are also fetched into the last level cache (LLC) during the page-table walk.

It has been demonstrated that cache-based side-channels can break coarse-grained address space layout randomization [22]. The location of the safe area can be determined through the following attack method: First, the attacker triggers the defense system's access to the safe area. To ensure this memory access invokes a PT walk, the attacker cleanses the corresponding TLB entries for the safe area's virtual address beforehand. Second, the attacker conducts a `Prime+Probe` or `Evict+Time` cache side-channel

and the offset is equal to 0. The only difference is that it will change the value of the designated register.

attack [44] to monitor which cache sets are used during the PT walk. As only certain virtual addresses map to a specific cache set, the virtual address of the safe area can be inferred using cache side-channel analysis.

However, it is worth mentioning that *to successfully determine the virtual address of one memory area, hundreds of such Prime+Probe or Evict+Time tests are needed. It is also imperative that the addresses of the PTEs corresponding to this memory area are not changed during these tests. That is, the cache entries mapped by these PTEs are not changed.* Our defense effectively invalidates such assumptions.

3 Threat Model

We consider an IH-based defense that protects a vulnerable application against code reuse attacks. This application either stands as a server that accepts and handles remote requests (e.g., through a web interface), or executes a sandboxed scripting code such as JavaScript as done in a modern web browsers. Accordingly, we assume the attacker has the permission to send malicious remote requests to the web servers or lure the web browsers to visit attacker-controlled websites and download malicious JavaScript code.

This IH-based defense has a safe area hidden in the victim process's memory space. We assume the *design* of the defense is not flawed: That is, before launching code reuse attacks, the attacker must circumvent the defense by revealing the locations of the safe areas (e.g., using one of many available techniques discussed in Section 2.2). We also assume the implementation of defense system itself is not vulnerable, and it uses IH correctly. We assume the underlying operating system is trusted and secured.

We assume the existence of some vulnerabilities in the application that allows the attacker to (a) read and write arbitrary memory locations; (b) allocate or free arbitrary memory areas (e.g., by interacting with the application's web interface or executing script directly); (c) create any number of threads (e.g., as a JavaScript program). These capabilities already represent the strongest possible adversary given in the application scenarios. Given these capabilities, all known attacks against IH can be performed.

3.1 Attack Vectors

Particularly, we consider the following attack vectors. All known attacks employ one of the four vectors listed below to disclose the locations of the safe areas.

- **Vector-1:** Gathering memory layout information to help to locate safe areas, by probing memory regions to infer if they are mapped (or allocated);
- **Vector-2:** Creating opportunities to probe safe areas without crashing the system, e.g., by leveraging resumable exceptions;

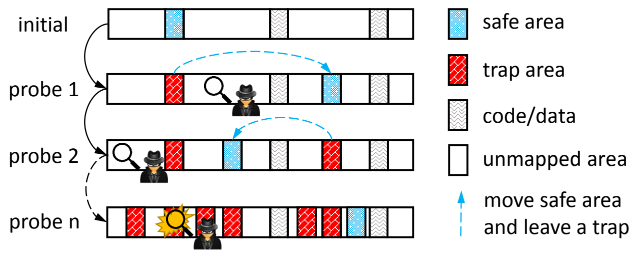

Figure 1: The high-level overview of the proposed re-randomization with the dispersed trap areas.

- **Vector-3:** Reducing the entropy of the randomized safe area locations to increase the success probability of probes, by decreasing the size of unmapped areas or increasing the size of safe areas;
- **Vector-4:** Monitoring page-table access patterns using cache side-channels to infer the addresses of safe areas, while triggering legal accesses to safe areas.

4 SafeHidden Design

We proposed SafeHidden, an IH technique that leverages re-randomization to prevent the attackers from locating the safe areas. It protects safe areas in both single-threaded programs and multi-threaded programs. It is designed primarily for Linux/X86_64 platform, as most of the defenses leveraging IH are developed on this platform.

At runtime, SafeHidden detects all potential memory probes. To avoid overly frequent re-randomization, it migrates the safe area to a new randomized location only after the detection of a suspicious probing. It then leaves a trap area of the same size behind. Figure 1 illustrates the high-level overview of the re-randomization method. In the figure, the memory layout is changed as the location of the safe area is being moved continuously, and the unmapped memory space becomes more fragmented by trap areas. The ever-changing memory layout could block **Vector-1**.

As the attackers continue to probe, new trap areas will be created. Gradually, it becomes more likely for probes to stumble into trap areas. If the attacker touches a trap area through any type of accesses, SafeHidden will trigger a security alarm and capture the attack. The design of trap areas mitigates the attacks from **Vector-2**, and significantly limits the attackers' ability to probe the memory persistently. While the attackers are still able to locate a safe area before accessing the trap areas, the probability is proven to be very small (see Section 4.4).

To block **Vector-3**, SafeHidden prevents unlimited shrink of unmapped areas and unrestricted growth of safe areas: (1) *Unmapped areas.* Because IH assumes that safe areas are hidden in a very large unmapped area, SafeHidden must prevent extremely large mapped areas. In our design, the

Events	**Interception Points**	**Responses in SafeHidden**			
		SA	**UA**	**TA**	**OA**
memory management syscalls	*mmap*, *munmap*, *mremap*, *mprotect*, *brk*, ...	Alarm	Rand	Alarm	–
syscalls that could return EFAULT	*read*, *write*, *access*, *send*, ...	Alarm	Rand	Alarm	–
cloning memory space	*clone*, *fork*, *vfork*	Rand	Rand	Rand	Rand
memory access instructions	*page fault exception*	–	Rand	Alarm	–

Table 2: Summary of potential stealthy probings and SafeHidden's responses. "**SA**": safe areas, "**UA**": unmapped areas, "**TA**": trap areas, "**OA**": other areas. "Alarm": triggering a security alarm. "Rand": triggering re-randomization. "–": do nothing.

maximum size of the mapped area allowed by SafeHidden is 64 TB, which is half of the entire virtual address space in the user space. Rarely do applications consume terabytes of memory; even big data applications only use gigabytes of virtual memory space; (2) *Safe areas.* Although safe areas in IH techniques are typically small and do not expand at runtime, attackers could create a large number of threads to increase the total size of the *thread-local* safe areas. To defeat such attacks, SafeHidden uses *thread-private* memory space to store *thread-local* safe areas. It maintains strict isolation among threads. When the *thread-local* safe area is protected using such a scheme, the entropy will not be reduced by thread spraying because any thread sprayed by an attacker can only access its own local safe area.

To mitigate **Vector-4**, SafeHidden also monitors legal accesses to the safe area that may be triggered by the attacker. Once such a legal access is detected, SafeHidden randomizes the location of the safe area. As the virtual address of the safe area is changed during re-randomization, the corresponding PTEs and their cache entries that are used by the attacker to make inferences no longer reflects the real virtual address of the safe area. Thus, **Vector-4** is blocked. It is worth noting that unlike the cases of detecting illegal accesses to the safe area, no trap area is created after the re-randomization.

In the following subsections, we will detail how SafeHidden recognizes and responds to the stealthy memory probes (see Section 4.1), how SafeHidden achieves the *thread-private* memory (see Section 4.2) and how SafeHidden defeats cache-based side-channel analysis (see Section 4.3).

4.1 Stealthy Memory Probes

In order to detect potential stealthy memory probes, we list all memory operations in the user space that can potentially be used as probings from the attackers (see Table 2).

The first row of Table 2 lists system calls that are related to memory management. The attackers could directly use them to gauge the layout of the memory space by allocating/deallocating/moving the memory or changing the permission to detect whether the target memory area is mapped or not. The second row lists the system calls that could return an EFAULT (bad address) error, such as "`ssize_t write (int fd, void * buf, size_t count)`". These system calls have a parameter pointing to a memory address. If the target memory is not mapped, the system call will fail without causing a crash, and the error code will be set to EFAULT. These system calls can be used to probe the memory layout without resulting in a crash. The third row lists the system calls that can clone a memory space. The attackers could use them to reason about the memory layout of the parent process from a child process. The fourth row lists memory access instructions that can trigger a *page fault exception* when the access permission is violated. The attackers could register or reuse the signal handler to avoid a crash when probing an invalid address.

Four types of memory regions are considered separately: safe areas, unmapped areas, trap areas, and other areas. Unmapped areas are areas in the address space that are not mapped; trap areas are areas that were once safe areas; other areas store process code and data. As shown in Table 2, SafeHidden intercepts different types of memory accesses to these areas and applies different security policy accordingly:

- If the event is an access to an unmapped area, SafeHidden will randomize the location of all safe areas. The original location of a safe area become a trap area.
- If the event is a memory cloning, it will perform randomization in the parent process after creating a child process, in order to make the locations of their safe areas different.
- If the event is an access to safe areas through memory management system calls or system calls with EFAULT return value, SafeHidden will trigger a security alarm.
- If the event is an access to trap areas through memory access instructions, memory management system calls, or system calls with EFAULT return value, it will trigger a security alarm.
- SafeHidden does not react to memory accesses to other areas. Since they do not have pointers pointing to the safe areas, probing other areas do not leak the locations.

To avoid excessive use of the virtual memory space, SafeHidden sets an upper limit on the total size of all trap areas (the default is 1 TB). Once the size of trap areas reaches the upper limit, SafeHidden will remove some randomly chosen trap area in each randomization round.

The design of such a security policy is worth further discussion here. Trap areas are previous locations of safe areas, which should be protected from illegal accesses in the same way as safe areas. As normal application behaviors never access safe areas and trap areas in an illegal way, accesses to them should raise alarms. For accesses to unmapped areas, an immediate alarm may cause false positives because the application may also issue memory management system calls, system calls with an EFAULT return value, or a memory access that touches unmapped memory areas. Therefore, accesses to unmapped areas only trigger re-randomization of the safe area to restore the randomness (that could invalidate the knowledge of previous probes), but no alarm will be raised. An alternative design would be counting the number of accesses to unmapped areas and raising a security alarm when the count exceeds a threshold. However, setting a proper threshold is very difficult because different probing algorithms could have different probing times. Therefore, monitoring critical subsets of the unmapped areas—the safe areas and trap areas—appears a better design choice.

4.2 Thread-private Memory

Thread-private memory technique was usually used in multi-threaded record-and-replay techniques [25, 7, 31]. We propose to use *thread-private* memory to protect safe areas. Conventional methods to implement *thread-private* memory is to make use of *thread-private* page tables in the OS kernel. As a separate page table is maintained for each thread, a reference page table for the entire process is required to keep track of the state of each page. The modification of the kernel is too complex, which cannot be implemented as a loadable kernel module: For example, to be compatible with `kswapd`, the reference page table must be synchronized with the private page tables of each thread, which requires tracking of CPU accesses of each PTE (especially the setting of the accessed and dirty bits[2] by CPU). The need for kernel source code modification and recompilation restricts the practical deployment of this approach.

To address this limitation, we propose a new approach to implement *thread-private* memory using the hardware virtualization support. Currently, a memory access in a guest VM needs to go through two levels of address translation: a *guest virtual address* is first translated into a *guest physical address* through the guest page table (GPT), which is then translated to its *host physical address* through a hypervisor maintained table, e.g., the *extended page table* (EPT) [38] in Intel processors, or the *nested page table* (NPT) [56] in AMD processors. Using Intel's EPT as an example, multiple virtual CPUs (VCPU) within a guest VM will share the same EPT. For instance, when the two VCPUs of a guest VM run

[2]These flags are provided for use by memory-management software to manage the transfer of pages into and out of physical memory. CPU is responsible for setting these bits via the physical address directly.

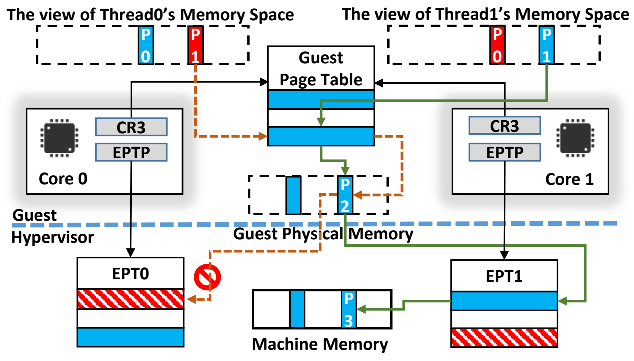

Figure 2: An example of the *thread-private* memory mechanism. P0 and P1 is *thread-private* memory page of Thread0 and Thread1, respectively.

two threads of the same program, both the virtual CR3 registers point to the page table of the program, and both EPT pointers (EPTPs) of VCPUs are pointing to a shared EPT.

To implement a *thread-private* memory, we can instead make each EPTP to point to a *separate* EPT to maintain its own *thread-private* memory. In such a scheme, each thread will have its own private EPT. The physical pages mapped in a thread's private memory in other threads' private EPTs will be made inaccessible. Figure 2 depicts an example of our *thread-private* memory scheme. When Thread1 tries to access its *thread-private* memory page P1, the hardware will walk both GPT and EPT1 to get the P3 successfully. But when Thread0 tries to access P1, it will trigger an EPT violation exception when the hardware walking EPT0 and be captured by the hypervisor.

In such a scheme, when a thread is scheduled on a VCPU, the hypervisor will set EPTP to point to its own EPT. In addition, SafeHidden synchronizes the EPTs by tracking the updates of the entries for the *thread-local* safe areas. For example, when mapping a guest physical page, SafeHidden needs to add the protection of all threads' EPTs for this page.

The *thread-private* memory defeats **Vector-3** completely. When *thread-local* safe areas are stored in such *thread-private* memory, spraying *thread-local* safe areas is no longer useful for the attackers because it will spray many prohibited areas that are similar to trap areas, called *shielded areas* (e.g., P1 is Thread0's shielded area in Figure 2), and be captured more easily.

4.3 Thwarting Cache Side-Channel Attacks

As discussed in Section 2.2.2, a key step in the cache side-channel attack by Gras et al. [22] is to force a PT walk when an access to the safe area is triggered. Therefore, a necessary condition for such an attack is to allow the attacker to induce TLB misses in a safe area. SafeHidden mitigates such attacks by intercepting TLB misses when accessing safe areas.

To only intercept the TLB miss occurred in safe areas, SafeHidden leverages a reserved bit in a PTE on X86_64 processors. When the reserved bit is set, a *page fault exception* with a specific error code will be triggered when the PTE is missing in TLB. Using this mechanism, a TLB miss can be intercepted and handled by the page fault handler. SafeHidden sets the reserved bit in all of the PTEs for the safe areas. Thus, when a TLB miss occurs, it is trapped into the page fault handler and triggers the following actions: (1) It performs one round of randomization for the safe area; (2) It clears the reserved bit in the PTE of the faulting page; (3) It loads the PTE (after re-randomization) of the faulting page into the TLB; (4) It then sets the reserved bit of the PTE again. It is worth noting that loading the TLB entry of the faulting page is a key step. Without this step, the program's subsequent accesses to the safe area will cause TLB misses again, which will trigger another randomization.

The re-randomization upon TLB miss effectively defeats cache-based side-channel analysis. As mentioned in Section 2.2.2, a successful side-channel attack requires hundreds of `Prime+Probe` or `Evict+Time` tests. However, as each test triggers a TLB miss, the safe area is re-randomized after every test. The PTEs used to translate the safe areas in each PT levels are re-randomized. Thus, the cache entries mapped by these PTEs are also re-randomized that completely defeating cache-based side-channels [22].

Nevertheless, two issues may arise: First, the PTEs of a safe area could be updated by OS (e.g., during a page migration or a reclamation), and thus clearing the reserved bits. To avoid these unintended changes to the safe areas' PTEs, SafeHidden traps all updates to the corresponding PTEs to maintain the correct values of the reserved bits. Second, as the location of a safe area is changed after a randomization, it will cause many TLB misses when the safe area is accessed at the new location, which may trigger many false alarms and re-randomizations. To address this problem, SafeHidden reloads the safe area's PTEs that were already loaded in the TLB back to the TLB after re-randomization. This, however, requires SafeHidden to know which PTEs were loaded in the TLB before the re-randomization. To do so, SafeHidden exploits an additional feature in Intel transactional synchronization extensions (TSX), which is Intel's implementation of hardware transactional memory [2]. During a re-randomization, SafeHidden touches each page in the safe area from inside of a TSX transaction. If there is a TLB miss, a *page fault exception* will occur because the reserved bit of its PTE is set. But this exception will be suppressed by a TSX transaction and handled by its abort handler. Therefore, SafeHidden can quickly find out all loaded PTEs before the re-randomization and reload them for the new location in the TLB without triggering any *page fault exception*.

Integrating SafeHidden with kernel page table isolation (KPTI) [1] introduces additional challenges. KPTI is a default feature used in the most recent Linux kernels. It separates the kernel page tables from user-space page tables, which renders the pre-loaded TLB entries of the safe areas in kernel unusable by the user-space application. We will detail our solution in section 5.

4.4 Security Analysis

SafeHidden by design completely blocks attacks through **Vector-1**, **Vector-3**, and **Vector-4**. However, it only probabilistically prevents attacks through **Vector-2**. As such, in this section, we outline an analysis of SafeHidden's security guarantee. Specifically, we consider a defense system with only one safe area hidden in the unmapped memory space. We abstract the attackers' behavior as a sequence of memory probes, each of which triggers one re-randomization of the safe area and creates a new trap area.

$$P_{c\_ith} = \begin{cases} (i \cdot P_t) \cdot \prod_{j=1}^{i-1} (1 - P_h - j \cdot P_t) & if\ i \le M \\ (M \cdot P_t) \cdot (\prod_{j=1}^{M} (1 - P_h - j \cdot P_t)) \cdot (1 - P_h - M \cdot P_t)^{i-1-M} & \\ & if\ i > M \end{cases} \tag{1}$$

The probability of detecting probes. Let the probability of detecting the attacks within N probes be P_{c_n}. Then the cumulative probability $P_{c\_n} = \sum_{i=1}^{n} P_{c\_ith}$, where P_{c_ith} represents the probability that an attacker *escapes* all $i-1$ probes, but is captured in the ith probe when it hits a trap area. An *escape* means that the attacker's probe is unsuccessful but remains undetected. P_{c_ith} is calculated in Equation (1), where i denotes the number of probes, j denotes the number of existing trap areas, P_h denotes the probability that the attacker hits the safe area in a probe, P_t represents the probability that the attacker hits one of the trap areas in a probe, M denotes the maximum number of trap areas. As an *escape* results in one re-randomization and the creation of a trap area, we approximate the number of existing trap areas with the number of escapes. But the number only increases up to M. So we consider if i reaches M separately. In the equation, $(i \cdot P_t)$ or $(M \cdot P_t)$ represents the probability that the probes are detected in the ith probe and $(1 - P_h - j \cdot P_t)$ or $(1 - P_h - M \cdot P_t)$ represents the probability of *escaping* the ith probe.

The attacker's success probability. We denote the probability of the attacker's successfully locating the safe area within N probes as P_{s_n}. $P_{s\_n} = \sum_{i=1}^{n} P_{s\_ith}$, where P_{s_ith} represents the probability that the attacker escapes in the first $i-1$ probes, but succeed in the ith probe. P_{s_ith} is provided in Equation (2).

$$P_{s\_ith} = \begin{cases} P_h \cdot \prod_{j=1}^{i-1} (1 - P_h - j \cdot P_t) & if\ i \le M \\ P_h \cdot (\prod_{j=1}^{M} (1 - P_h - j \cdot P_t)) \cdot (1 - P_h - M \cdot P_t)^{i-1-M} & \\ & if\ i > M \end{cases} \tag{2}$$

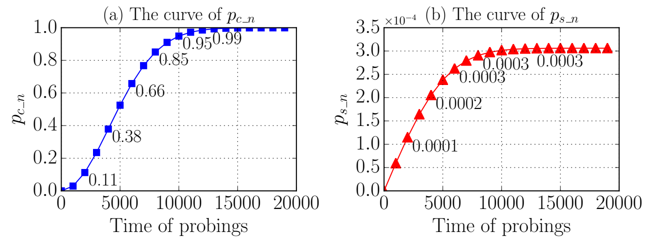

Figure 3: The probability of being captured by SafeHidden within N probes (a) and the probability of locating the safe areas within N probes successfully (b).

Discussion. When the size of the safe area is set to 8 MB, and the maximum size of all trap areas is set to 1 TB, as shown in Figure 3(a), P_{c_n} increases as the number of probes grows. When the number of probes reaches 15K, SafeHidden detects the attack with a probability of 99.9%; P_{c_n} approaches 100% as the number of probes reaches 20K. Figure 3(b) suggests the value of P_{s_n} increases as the number of probes increases, too. But even if the attacker can escape in 15K probes (which is very unlikely given Figure 3(a)), the probability of successfully locating the safe area is still only 0.03% (shown in Figure 3(b)), which is the maximum that could ever be achieved by the attacker. Notice that our abstract model favors the attackers, for example: (1) no *shielded area*s are considered in the analysis; (2) randomization triggered by applications' normal activities and TLB misses is ignored in the analysis. Obviously, in the real world situation, the attacker's success probability will be even lower, and the attack will be caught much sooner.

5 System Implementation

SafeHidden is designed as a loadable kernel module. Users could deploy SafeHidden by simply loading the kernel module, and specifying, by passing parameters to the module, which application needs to be protected and which registers point to the safe area. *No modification of the existing defenses or re-compiling the OS kernel is needed.*

5.1 Architecture Overview of SafeHidden

As described in Section 4.2, SafeHidden needs the hardware virtualization support. It can be implemented within a Virtual Machine Monitor (VMM), such as Xen or KVM. However, the need for virtualization does not preclude its application in non-virtualized systems. To demonstrate this, we integrated a thin hypervisor into the kernel module for a non-virtualized OS. The thin hypervisor virtualizes the running OS as the guest without rebooting the system. The other components inside the kernel module are collectively called GuestKM, which runs in the guest kernel.

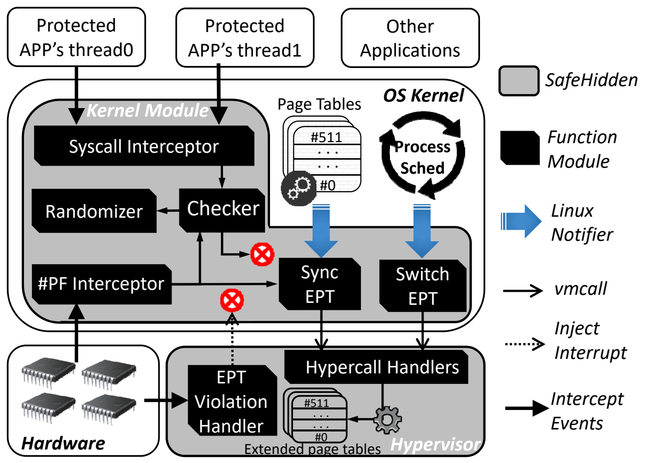

Figure 4: Architecture overview of SafeHidden.

After loading the SafeHidden module, it first starts the hypervisor and then triggers the initialization of GuestKM to install hooks during the *Initialization Phase*. Figure 4 shows an overview of SafeHidden's architecture. We can see that SafeHidden is composed of two parts: the hypervisor and the GuestKM. In the initialization phase, GuestKM installs hooks to intercept three kinds of guest events: *context switching*, *page fault exception*s, and certain *system call*s.

SafeHidden then starts to protect the safe areas by randomizing their locations and isolating the *thread-local* safe areas during the *Runtime Monitoring Phase*. In the GuestKM, the *Syscall Interceptor* and the *#PF Interceptor* modules are used to intercept *system call*s and *page fault exceptions*. When these two types of events are intercepted, they will request the *Checker* module to determine if SafeHidden needs to raise a security alarm, or if it needs to notify the *Randomizer* module to perform randomization. Meanwhile, SafeHidden needs to maintain the *thread-private* EPT to isolate the *thread-local* safe areas. The *sync EPT* module is used to synchronize the protected threads' page tables with their EPTs. The *switch EPT* module will switch EPTs when a protected thread is scheduled. Because both modules need to operate EPTs, they are coordinated by the *Hypercall Handlers* module. The *EPT Violation Handler* module is used to monitor illegal accesses to the *thread-local* safe areas.

5.2 Initialization Phase

Task-1: starting hypervisor. When the kernel module is launched, the hypervisor starts immediately. It configures the EPT paging structures, enables virtualization mode, and places the execution of the non-virtualized OS into the virtualized guest mode (non-root VMX mode). At this time, it only needs to create a default EPT for guest. Because the guest is a mirror of the current running system, the default EPT stores a one-to-one mapping that maps each guest physical address to the same host physical address.

Task-2: installing hooks in guest kernel. When the guest starts to run, GuestKM will be triggered to install hooks to intercept three kinds of events: 1) To intercept the *system calls*, GuestKM modifies the `system_call_table`'s entries and installs an alternative handler for each of them; 2) To intercept the *page fault exception*, GuestKM uses the `ftrace` framework in Linux kernel to hook the `do_page_fault` function; 3) To intercept *context switches*, GuestKM uses the standard preemption notifier in Linux, `preempt_notifier_register`, to install hooks. It can be notified through two callbacks, the `sched_in()` and the `sched_out()`, when a context switch occurs.

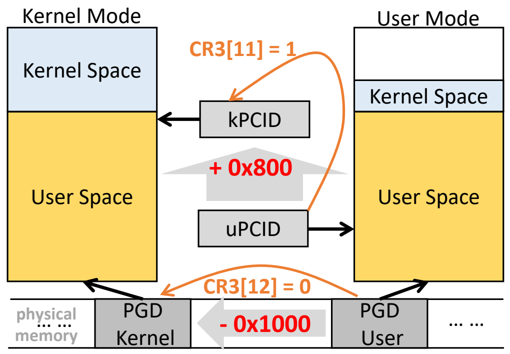

Figure 5: Overview of kernel page-table isolation.

5.3 Runtime Monitoring Phase

Recognizing safe areas. GuestKM intercepts the `execve()` system call to monitor the startup of the protected process. Based on the user-specified dedicated register, GuestKM can monitor the event of setting this register to obtain the value. In Linux kernel, the memory layout of a process is stored in a list structure, called `vm_area_struct`. GuestKM can obtain the safe area by searching the link using this value. According to Table 1, there are two kinds of registers that store the pointer of a safe area: 1) The 64-bit Linux kernel only allows a user process to set the `%gs` or `%fs` segmentation registers through the `arch_prctl()` system call [3]. So, GuestKM intercepts this system call to obtain the values of these registers; 2) All existing methods listed in Table 1 use `%rsp` pointed safe area to protect the stack. So, GuestKM analyzes the execution result of the `execve()` and the `clone()` system calls to obtain the location of the safe area, i.e., the stack, of the created thread or process. Once a safe area is recognized, its PTEs will be set invalid by setting the reserved bits.

To determine whether a safe area is *thread-local* or not, GuestKM monitors the event of setting the dedicated register in child threads. If the register is set to point to a different memory area, it means that the child thread has created its *thread-local* safe area, and the original safe area belongs to the parent. Until the child thread modifies the register to point to a different memory area, it shares the same safe area with its parent.

Randomizing safe areas. As described in Section 4.1 and 4.3, when GuestKM needs to perform randomization, it invokes the customized implementation of `do_mremap()` function in the kernel with a randomly generated address (by masking the output of the `rdrand` instruction with The `0x7ffffffff000`) to change the locations of the safe areas. If the generated address has been taken, the process is repeated until a usable address is obtained. It is worth noting that GuestKM only changes the virtual address of the safe area, the physical pages are not changed. After migrating each safe area (not triggered by the TLB miss event), GuestKM will invoke `do_mmap()` with the protection flag `PROT_NONE` to set the original safe area to be a trap area. For multi-threaded programs, when the execution of a thread triggers a randomization (not triggered by the TLB miss event), the safe areas of all threads need to be randomized. To ensure the correctness, GuestKM needs to block all threads before randomizing all (or *thread-shared*) safe areas.

Although all safe areas used in existing defenses in Table 1 are *position-independent*, we do not rule out the possibility that future defenses may store some *position-dependent* data in the safe area. However, as any data related to an absolute address can be converted to the form of a base address with an offset, they can be made position independent. Therefore, after randomizing all safe areas, SafeHidden just needs to modify the values of the dedicated registers to point to the new locations of the safe areas.

Loading TLB entries under KPTI. The kernel page table isolation (KPTI) feature [1] was introduced into the mainstream Linux kernels to mitigate the Meltdown attack [32]. For each processs, it splits the page table into a user-mode page table and a kernel-mode page table (as shown in Figure 5). The kernel-mode page table includes both kernel-space and user-space addresses, but it is only used when the system is running in the kernel mode. The user-mode page table used in the user mode contains all user-space address mappings and a minimal set of kernel-space mappings for serving system calls, handling interrupts and exceptions. Whenever entering or exiting the kernel mode, the kernel needs to switch between the two page tables by setting the CR3 register. To accelerate the page table switching, the roots of the page tables (i.e., PGD kernel and PGD user in Figure 5) are placed skillfully in the physical memory so that the kernel only needs to set or clear the bit 12 of CR3.

Moreover, to avoid flushing TLB entries when switching page tables, the kernel leverages the Process Context Identifier (PCID) feature [2]. When PCID is enabled, the first 12 bits (bit 0 to bit 11) of the CR3 register represents the PCID of the process which is used by the processor to identify the

[3]Recent CPUs supporting the `WRGSBASE/WRFSBASE` instructions allow setting the `%gs` and `%fs` base directly, but they are restricted by the Linux kernel to use in user mode.

owner of a TLB entry. The kernel assigns different PCIDs to the user and kernel modes (i.e., kPCID and uPCID in the figure). When entering or exiting the kernel mode, the kernel needs to switch between kPCID and uPCID. To accelerate this procedure, kPCID and uPCID of the same process only differ in one bit. Therefore, the kernel only needs to set or clear the bit 11 of CR3.

```
 1 // .S file
 2 .globl asm_load_pte_irqs_off
 3 .align 0x1000
 4 asm_load_pte_irqs_off:
 5 /* 1. Get CR3 (kernel-mode page table with kPCID) */
 6     mov %cr3, %r11
 7 /* 2. Switch to kernel-mode page table with uPCID */
 8     bts 63, %r11 // set noflush bit
 9     bts 11, %r11 // set uPCID bit
10     mov %r11, %cr3 // set CR3
11 /* 3. Access user-mode pages to load pte into TLB */
12     stac // Allow user-mode pages accesses
13     movb (%rdi), %al // Read a byte from user-mode page
14     clac // Disallow user-mode pages accesses
15 /* 4. Get uPCID value */
16     mov %r11, %rax
17     and $0xfff, %rax
18 /* 5. Switch to kernel-mode page table with kPCID */
19     bts 63, %r11 // set noflush bit
20     btc 11, %r11 // clear uPCID bit
21     mov %r11, %cr3 //set CR3
22     retq //return uPCID
23 // .c file
24 void load_pte_into_TLB(unsigned long addr) {
25     unsigned long flags, uPCID;
26     // disable preemption and interrupts
27     get_cpu(); local_irq_save(flags);
28     uPCID = asm_load_pte_irqs_off(addr);
29     // flush the TLB entries for a given pcid and addr
30     invpcid_flush_one(uPCID, asm_load_pte_irqs_off);
31     // enable preemption and interrupts
32     local_irq_restore(flags); put_cpu();
33 }
```

Listing 1: The code snippet to load the TLB entries under KPTI.

As mentioned in Section 4.3, SafeHidden needs to load PTEs of the safe areas into the TLB every time it randomizes the safe areas. However, it is challenging to make SafeHidden compatible with KPTI. This is because SafeHidden only runs in the kernel mode—it uses the kernel-mode page table with kPCID, but the TLB entries of the safe areas must be loaded from the user-mode page table using uPCID.

An intuitive solution is to map SafeHidden into the kernel space portion of the user-mode page tables. Then the PTE loading is performed with uPCID. However, this method introduces more pages into the user-mode page tables and thus increases the attack surface of the Meltdown attack.

We propose the following alternative solution: SafeHidden still runs in the kernel mode using the kernel-mode page table. Before loading the TLB entries of the safe areas, it switches from kPCID to uPCID temporarily. Then without switching to the user-mode page table, it accesses the safe area pages to load the target PTEs into the TLB with uPCID. There is no need to switch to the user-mode page table for two reasons: (1) TLB entries are only tagged with PCIDs and virtual addresses; (2) the user-space addresses are also mapped in the kernel-mode page table. After the PTE loading, SafeHidden switches back to kPCID and then flushes the TLBs of the instruction/data pages related to the loading operation. This is to avoid these TLB entries (tagged with uPCID) to be exploited by the Meltdown attack.

Listing 1 illustrates the details of how to load user PTEs into the TLB from the kernel mode code under KPTI. Line 24 shows the function definition of this loading operation. Line 27 disables interrupts and preemptions to avoid unintended context switches. Line 28 invokes the assembly code for the loading operation. Line 6 reads the current CR3 register which contains the root of the kernel-mode page table and the kPCID. Line 8-10 switch to use uPCID (but keeping the kernel-mode page table unchanged). Line 8 sets the *noflush* bit to avoid flushing the target PCID's TLB entries when setting the CR3 register. Line 12 enables data access to user pages by disabling SMAP temporarily. Line 13-14 load the target PTE into TLB with uPCID by reading a byte from this page. Line 16-21 switch back to kPCID. Because line 12-21 code runs under the kernel-mode page table with uPCID, this code page mapping will be loaded into the TLB that can be accessed by user-mode code later. This page content could be leaked from the malicious process using the Meltdown attack. So line 30 flushes the mapping from the TLB.

Reloading TLB entries after randomization. SafeHidden uses Intel TSX to test which PTEs of the safe areas are loaded in the TLB. The implementation is very similar to the method of loading the user-mode TLB entries. The only difference is that SafeHidden encloses the code of line 13 (Listing 1) into a transaction (between `xbegin` and `xend` instructions). In fact, not all PTEs of the safe area need to be tested. SafeHidden only tests the PTEs that were reloaded in the last re-randomization.

Tracking GPT updates. The GPT entries of safe areas will be updated dynamically. In order to track such updates efficiently, we choose to integrate the Linux MMU notifier `mmu_notifier_register` in GuestKM. The MMU notifier provides a collection of callback functions to notify two kinds of page table updates: invalidation of a physical page and migration of a physical page. But it does not issue a callback when OS maps a physical page to a virtual page. To address this problem, we handle it in a lazy way by intercepting the *page fault exception* to track this update. Once GuestKM is notified about these updates, GuestKM makes the modified entry invalid or valid, and then issues a *hypercall* to notify the hypervisor to synchronize all EPTs.

Creating and destructing thread-private EPT. If a thread has no *thread-local* safe area, it shares its parent's EPT. If it is the main thread, it will be configured to use the default EPT. If a thread has a *thread-local* safe area, GuestKM will issue a *hypercall* to notify the hypervisor to initialize an EPT for this thread. When initializing an EPT, SafeHidden will configure the entries based on other threads' local safe areas by walking the GPT to find all physical pages in the safe areas. Meanwhile, SafeHidden will also modify the entries of other thread's EPT to make all *thread-local* safe areas isolated from each other. Whenever SafeHidden changes other thread's EPT, it will block the other threads first. GuestKM also intercepts the `exit()` system call to monitor a thread's destruction. Once a thread with a private EPT is killed, GuestKM notifies the hypervisor to recycle its EPT.

Monitoring context switches. When a thread is switched out, GuestKM will be notified through the `sched_out()` and it will switch to the default EPT assigned to the corresponding VCPU. When GuestKM knows a new thread is switched in through the `sched_in()`, it will check whether the thread has a private EPT or not, and switches to its EPT in if it does.

Monitoring illegal accesses. GuestKM intercepts all system calls in Table 2 and checks their access areas by analyzing their arguments. If there is an overlap between their access areas with any of the trap areas, the safe areas, or the shielded areas, GuestKM will trigger a security alarm. Because there is no physical memory allocated to the trap areas, any memory access to those areas will be captured by intercepting the *page fault exception*. With the isolation of the *thread-local* safe area, any memory access to the shielded areas will trigger an *EPT violation exception*, which will be captured by the hypervisor (that notifies GuestKM). GuestKM triggers a security alarm in cases of any of these events.

Handling security alarms. How these security alarms are handled depends on the applications. For example, when SafeHidden is applied in browsers to prevent exploitation using JS code, it could mark the website from which the JS code is downloaded as malicious and prevent the users from visiting the websites. When SafeHidden is used to protect web servers, alarms can be integrated with application firewalls to block the intrusion attempts.

6 Evaluation

We implemented SafeHidden on Ubuntu 18.04 (Kernel 4.20.3 with KPTI enabled by default) that runs on a 3.4GHZ Intel(R) Core(TM) i7-6700 CPU with 4 cores and 16GB RAM. To evaluate the security and performance of SafeHidden, we implemented by ourselves two defenses that use safe areas, *OCFI* and *SS*. *OCFI* is a prototype implementation of O-CFI [40], which uses *thread-shared* safe areas (Table 1). *OCFI* first randomizes the locations of all basic blocks and then instruments all indirect control transfer instructions that access the safe areas, i.e., indirect calls, indirect jumps, and returns. Each indirect control transfer instruction has an entry in the safe areas, which contains the boundaries of possible targets. For each instrumented instruction, *OCFI* obtains its jump target and checks if it is within the legal range.

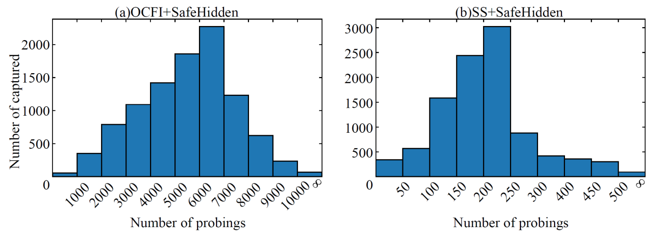

Figure 6: The distribution of probing times before being captured (10,000 probes launched).

SS is our implementation of a shadow stack, which is an example of the *thread-local* safe areas (see Table 1). Shadow stacks are used in Safe Stack [30], ASLR-Guard [36], and ReRanz [57]. *SS* adopts a compact shadow stack scheme [41] (in contrast to a parallel shadow stack scheme). In our implementation, the pointer (i.e., offset) to the stack top is stored at the bottom of the shadow stack. To be compatible with uninstrumented libraries, *SS* instruments function prologues and epilogues to access the shadow stacks (i.e., the safe areas). Listing 2 shows the function prologue for operating shadow stacks. The epilogue is similar but in an inverse order. The epilogue additionally checks if the return address has been modified.

In both cases, the size of the safe area is set to be 8 MB. To use SafeHidden with *SS* and *OCFI*, one only needs to specify in SafeHidden that the `%gs` register points to the safe areas. No other changes are needed.

```
mov (%rsp), %rax //get the return address
mov %gs:0x0, %r10 //get the shadow stack (ss) pointer
mov %rax, %gs:(%r10) //push the return address into ss
mov %rsp, %gs:0x8(%r10) //push the stack frame into ss
add $0x10, %gs:0x0//increment the shadow stack pointer
```

Listing 2: The shadow stack prologue.

6.1 Security Evaluation

We evaluated SafeHidden in four experiments. Each experiment evaluates its defense against one attack vector.

In the first experiment, we emulated an attack that uses the *allocation oracle*s [43] to probe *Firefox* browsers under *OCFI*'s protection. The prerequisite of this attack is the ability to accurately gauge the size of the unmapped areas around the safe areas. To emulate this attack, we inserted a shared library into *Firefox* to gauge the size of the unmapped areas. When SafeHidden is not deployed, the attack can quickly locate the safe area with only 104 attempts. Then we performed 10,000 trials of this attack on *Firefox* protected by *OCFI* and SafeHidden. The result shows that all the 10,000 trials failed, but in two different scenarios: In the first scenario (9,217 out of 10,000 trials), the attacks failed to gauge the size of the unmapped areas even when the powerful binary search method is used. The prerequisite of a

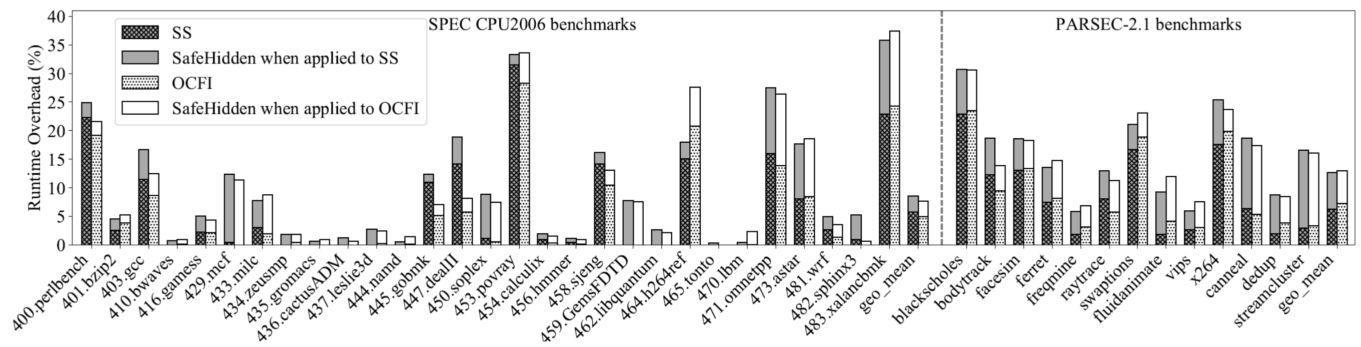

Figure 7: Performance overhead of SPEC and Parsec-2.1 benchmarks brought by SafeHidden when applied to the SS and OCFI defenses.

binary search is that the location of the target object does not change. However, SafeHidden's re-randomization confuses the binary search because the safe area moves continuously. In the second scenario, even though the attacks can gauge the exact size of an unmapped area, they always stumble into one of the trap areas when accessing the surroundings of the unmapped area, which triggers security alarms.

In the second experiment, we launched 10,000 trials of CROP attacks [19] to probe a *Firefox* protected by *OCFI*. The result shows that the attacks always successfully identified the location of the safe area when SafeHidden is not deployed. The time required is less than 17 minutes with no more than 81,472,151 probes. However, the attacks always fail when SafeHidden is deployed. Figure 6 (a) shows the distribution of the number of probes before an attack is detected by hitting a trap area. We can see that the distribution is concentrated in the range between [2000, 9000]. This experiment shows that SafeHidden can prevent the continuous probing attacks effectively.

In the third experiment, we launched 10,000 trials of the CROP attack using thread spraying to probe *Firefox* protected by *SS*. We sprayed 2^{14} (=16,384) threads with more than 16,384 *thread-local* safe areas, and then scanned the *Firefox* process with a CROP attack. The result shows that when SafeHidden is not deployed, the attacks can probe the locations of the safe areas successfully. The time taken is 0.16s, with only 2,310 probes. With SafeHidden deployed, all probes are captured before succeeding. Figure 6 (b) shows the distribution of the number of probes before being captured. The distribution is concentrated in the range between [50, 300], which is much lower than those in the second experiment. There are two reasons for that: 1) The other threads' local safe areas become the current thread's *shielded area*s, which increases the probability of the probes being captured; 2) All safe areas will be randomized after each probe, which increases the number of trap areas quickly.

In the fourth experiment, we emulated a cache side-channel attack against page tables using Revanc [54], which is a tool based on [46]. This tool allocates a memory buffer and then measures the access time of different pages in this buffer repeatedly. It could infer the base address of this buffer. To utilize this attack method against IH, we kept this memory buffer in a safe area by modifying the source code to force any access to this memory buffer through an offset from the %gs register. When SafeHidden is not deployed, this attack can obtain the correct base address of this buffer. The attack fails when SafeHidden is deployed.

6.2 Performance Evaluation

We evaluated SafeHidden's impact on the application's performance in terms of CPU computation, network I/O, and disk I/O, respectively. For the experiment of CPU computation, we ran SPEC CPU2006 benchmarks with *ref* input and multi-threaded Parsec-2.1 benchmarks using *native* input with 8 threads; For the experiment of network I/O, We chose the Apache web server httpd-2.4.38 and Nginx-1.14.2 web server. Apache was configured to work in *mpm-worker* mode, running in one worker process with 8 threads. Nginx was configured to work with 4 worker processes; For the experiment of disk I/O, we chose benchmark tool Bonnie++ (version 1.03e). For each benchmark, we prepared two versions of the benchmark: (1) protected by *SS*, and (2) protected by *OCFI*. We evaluated both the performance overhead of protecting these benchmarks using *SS* and *OCFI* defenses and the additional overhead of deploying SafeHidden to enhance the *SS* and *OCFI* defenses.

6.2.1 CPU Intensive Performance Evaluation

Figure 7 shows the performance overhead of the *OCFI* and *SS* defenses, and also the performance overhead of SafeHidden when applied to enhance the *OCFI* and *SS* defenses. For SPEC benchmarks, we can see that the geometric mean performance overhead incurred by *OCFI* and *SS* is 4.94% and 5.79%, respectively. For Parsec benchmarks, the geometric mean performance overhead incurred by *OCFI* and *SS* is 7.23% and 6.24%. The overhead of some applications (e.g., *perlbench*, *povray*, *Xalancbmk* and *blacksholes*) is higher because these applications frequently execute direct function calls and indirect control transfer instructions, which trigger accesses to safe areas. Note these overheads were caused by

Program	#randomization		Details of #randomization				Program	#randomization		Details of #randomization			
	SS	OCFI	#brk()	#mmap()	#tlb_miss			SS	OCFI	#brk()	#mmap()	#tlb_miss	
					SS	OCFI						SS	OCFI
SPEC CPU2006 benchmark													
bzip2	3,260	4,816	36	100	3,124	4,680	calculix	40,914	37,319	32,095	139	8,680	5,085
gcc	150,550	148,649	6,816	194	143,540	141,639	hmmer	2,851	2,430	13	25	2,813	2,392
bwaves	757	764	701	45	11	18	sjeng	207,559	196,562	3	10	207,546	196,549
gamess	192	311	27	30	135	254	GemsFDTD	184	205	11	160	13	14
mcf	435,637	424,266	3	11	435,623	424,252	libquantum	373,904	201,652	14	39	373,851	201,599
milc	685,788	576,056	2,687	44	683,057	573,325	h264ref	6,650	2,496	545	60	6,045	1,891
zeusmp	108	425	3	10	95	412	tonto	327	335	298	20	9	17
gromacs	287	134	44	36	207	54	lbm	6,110	5,822	3	11	6,096	5,808
cactusADM	11,884	11,826	8,997	66	2,821	2,763	omnetpp	320,474	223,832	1,245	56	319,173	222,531
leslie3d	60	94	5	27	28	62	astar	872,397	667,817	3,928	46	868,423	663,843
namd	474	630	100	31	343	499	wrf	53,018	49,230	419	253	52,346	48,558
gobmk	10,062	64,491	59	594	9,409	63,838	sphinx3	3,572	2,790	144	146	3,282	2,500
dealII	53,113	53,618	40,103	53	12,957	13,462	xalancbmk	921,406	781,973	3,099	94	918,213	778,780
soplex	168,463	186,807	168	49	168,246	186,590	**average**	**151,131**	**129,452**	**3,778**	**85**	**147,268**	**125,588**
Parsec-2.1 benchmark													
blackscholes	156,968	114,375	3	22	156,943	114,350	fluidanimate	168,896	175,816	231	23	168,642	175,562
bodytrack	11,205	10,426	2,486	6,558	2,161	1,382	vips	2,375	1,961	4	115	2,256	1,842
facesim	41,775	22,813	359	69	41,347	22,385	x264	5,768	8,055	42	162	5,564	7,851
ferret	93,815	62,870	222	39,032	54,561	23,616	canneal	244,669	251,238	5,917	24	238,728	245,297
freqmine	3,729	2,386	499	64	3,166	1,823	dedup	58,868	33,631	1,571	715	56,582	31,345
raytrace	27,510	22,859	1,279	57	26,174	21,523	streamcluster	273,684	219,572	7	23	273,654	219,542
swaptions	6,477	5,127	3	22	6,452	5,102	**average**	**84,288**	**71,625**	**971**	**3,607**	**79,710**	**67,048**

Table 3: Statistical data of SafeHidden when applied to the SS and OCFI defenses to protect SPEC CPU2006 and Parsec-2.1 benchmarks.

the adoption of *OCFI* and *SS*, but not SafeHidden.

For SPEC benchmarks, we can see that the geometric mean performance overhead incurred by SafeHidden when protecting *OCFI* and *SS* is 2.75% and 2.76%, respectively. For Parsec benchmarks, the geometric mean performance overhead incurred by SafeHidden is 5.78% and 6.44%, respectively. It shows that SafeHidden is very efficient in protecting safe areas. Based on the experimental results, we can also see that SafeHidden is more efficient in protecting single-threaded applications. This is due to two reasons: (1) All threads need to be blocked when randomizing the *thread-shared* safe areas or the *thread-local* safe areas (when not triggered by a TLB miss); (2) When protecting the *thread-local* safe areas, SafeHidden needs to synchronize the *thread-private* EPTs with the guest page table, which could introduce VM-Exit events.

Table 3 details some statistical data of SafeHidden when applied to the *OCFI* and *SS* defenses to protect SPEC and Parsec benchmarks. The column "#randomization" shows the number of re-randomization to safe areas. On SPEC and Parsec benchmarks, there are three operations that can trigger a re-randomization: (1) Using `brk()` to move the top of the heap; (2) Using `mmap()` to allocate a memory chunk; (3) TLB misses occurred in safe areas. Because *OCFI* and *SS* did not introduce extra invocation of system calls, the numbers of `brk()` and `mmap()` are the same. Combined with Figure 7, we can see that for most of SPEC benchmarks (except *mcf*, *soplex*, *GemsFDTD* and *omnetpp*), the performance overhead is related to the total number of re-randomization. The reason why those four benchmarks had different performance overhead is the virtualization overhead incurred by the hypervisor. For example, the hypervisor introduced 7.18% performance overhead for *GemsFDTD*. Except *x264* using *SS*, *canneal* and *streamcluster*, the overhead of most Parsec benchmarks is also related to the total number of re-randomization. For *canneal* and *streamcluster*, most of performance overhead is introduced by the virtualization. For *x264*, it spawns child threads more frequently than other benchmarks, which causes SafeHidden to frequently create and initialize *thread-private* EPTs.

6.2.2 Network I/O Performance Evaluation

Figure 8 shows the performance degradation of Apache and Nginx servers under the protection of *SS* and *OCFI* with and without SafeHidden. We use *ApacheBench (ab)* to simulate 100 concurrent clients constantly sending 10,000 requests, each request asks the server to transfer a file. We also varied the size of the requested file, i.e., {1K, 5K, 20K, 100K, 200K, 500K}, to represent different configurations. From the figure, we can see that *SS* only incurs 1.60% and 1.98% overhead on average when protecting Apache and Nginx. *OCFI* only incurs 1.45% and 2.13% overhead on average when protecting Apache and Nginx. We can also see that SafeHidden incurs 12.18% and 12.07% on average when applied to *SS* and *OCFI* to protect Apache. But SafeHidden incurs only 5.51% and 5.35% on average when applied to SS and OCFI to protect Nginx. So SafeHidden is more efficient in protecting Nginx than Apache. This is due to two reasons: (1) For each request to Nginx, Nginx will invoke several I/O system calls, such as `recvfrom()`, `write()`, `writev()`, etc., which only access the allocated memory space in the

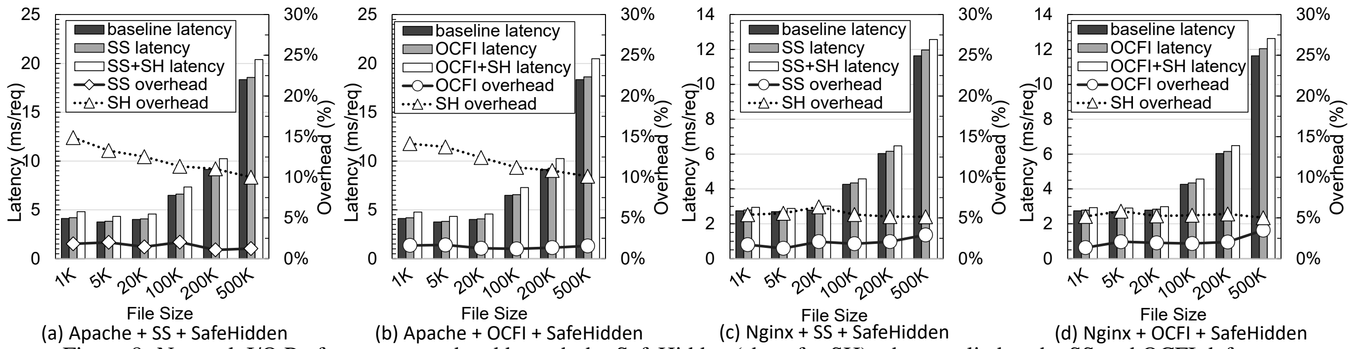

Figure 8: Network I/O Performance overhead brought by SafeHidden (short for SH) when applied to the SS and OCFI defenses.

Nginx process. The system calls in Nginx will not trigger randomization of the safe area. But for each request to Apache, Apache will invoke the `mmap()` system call to map the requested file into the virtual memory space which could trigger the extra randomization of all safe areas compared with Nginx; (2) Apache is a multi-threaded program. SafeHidden needs to block all threads when performing randomization of safe areas triggered by the `mmap()` system call.

6.2.3 Disk I/O Performance Evaluation

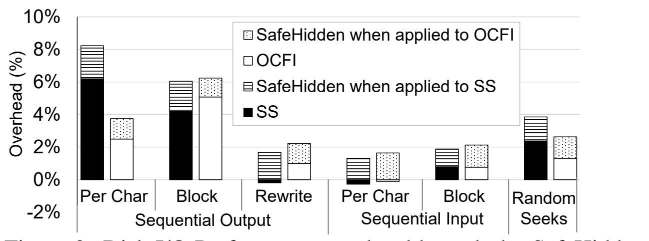

Figure 9: Disk I/O Performance overhead brought by SafeHidden when applied to the SS and OCFI defenses.

The Bonnie++ sequentially reads/writes data from/to a particular file in different ways. The read/write granularity varies from a character to a block (i.e., 8192 Bytes). Furthermore, we also test the time cost of the random seeking. Figure 9 shows the disk I/O measurement results: *SS* and *OCFI* defenses incur low performance overhead, i.e., 2.18% overhead on average for *SS* and 1.76% overhead on average for *OCFI*. SafeHidden brings only 1.58% overhead on average for *SS* and 3.08% overhead on average for *OCFI*. Compared with SPEC and Parsec benchmarks, this tool invokes the *write()* and *read()* system calls to write and read a very large file frequently. But these system calls only access the allocated memory space that does not trigger randomization of safe areas.

7 Discussion

TLBleed attack. TLBleed [21] exploits the shared TLBs between the hyper-threads on the same core to infer victim programs' memory access patterns. Potentially, it could be used to reduce the entropy of ASLR by triggering TLB misses and observing into which TLB set the target object is mapped. When TLBleed is used against SafeHidden, by triggering only L1 DTLB misses without L2 TLB misses, TLBleed may reduce the entropy of the safe area location by 4 bits (in the case of a 16-set L1 DTLB), which leads to roughly 20 bits entropy remaining for 8 MB safe area. However, attempts to further reduce the entropy will trigger re-randomization of safe areas with high probability. So, TLBleed is not able to defeat SafeHidden.

Spectre attack. The Spectre attack [28] leverages speculative execution and side-channels to read the restricted virtual memory space. As the memory protection related exceptions are suppressed in the speculatively executed code, SafeHidden could not detect Spectre attacks.

Resilience to attacks. SafeHidden is resilient to all known attacks against safe areas. Variants of existing attacks would also be prevented: (1) The attacker may try to fill up the address space quickly by using the *persistent allocation oracle* [43] to avoid SafeHidden from creating too many trap areas. But as SafeHidden sets an upper limit for the total mapped memory regions, such attacks are prevented; (2) The attacker could exploit the *paging-structure caches* to conduct the side-channel analysis. However such attacks will also trigger TLB misses, which will be detected by SafeHidden. Although it is difficult to prove SafeHidden has eliminated all potential threats, we believe it has considerably raised the cost of attacks in this arms race.

The impact of NMI on the solution of integrating KPTI. During the execution of the assembly code in listing 1, the interrupts are disabled to avoid unintended context switches. But the non-maskable interrupt (NMI) could break this protection. If a NMI occurs when the code is running, the NMI handler will run on the kernel-mode page table with the uPCID. So the memory pages accessed in the NMI handler could be leaked via the Meltdown attack. To avoid this, the entry of the NMI handler could be instrumented (by rewriting the NMI entry in IDT) to switch back to the kPCID.

8 Related Work

Protecting safe areas. MemSentry [29], IMIX [16], MicroStache [39], and ERIM [52] are the closest to our work. MemSentry adopts a *software-fault isolation* (SFI) approach to protecting frequently accessed safe areas by leveraging Intel's memory protection extensions (MPX) technology. It restricts the addresses of all memory accesses that can not access the safe area. But it is still not practical because it significantly increases the performance overhead [16]. The main disadvantage of MemSentry is the SFI approach is not safe, i.e., un-instrumented instructions can still access the safe region [16]. By modifying the Intel's simulation, IMIX extends the x86 ISA with a new memory-access permission to mark safe areas as security sensitive and allows accesses to safe areas only using a newly introduced instruction. Similarly, MicroStache achieves it by modifying the Gem5 simulator. However, IMIX and MicroStache are not yet supported by commodity hardware. ERIM protects safe areas by turning on access permission only when accesses are requested. To quickly switch the access permission on and off, it adopts the newly released Intel hardware feature memory protection keys (MPK) [2]. But it is still not suitable to protect the frequently accessed safe areas. For example, it incurs $>$1X performance overhead when protecting the shadow stack [29]. Different from SafeHidden, all these methods require modification of the source code of both the defense and the protected applications. Please note that most defenses listed in Table 1 (except two) work on COTS binaries. In particular, Shuffler [59] mentioned that defeats probing attacks by moving the location of its code pointer table (i.e., the safe area) continuously. But this method only blocks attacks from **Vector-1**. For example, using **Vector-2**, persistent attacks could always succeed. Different from Shuffler, SafeHidden blocks all existing attack vectors against IH.

Protecting CFI metadata. CFI is an important defense against code reuse attacks [3]. A CFI mechanism stores control-flow restrictions in its metadata. Like other types of safe areas, the metadata of CFI mechanisms needs to be protected. However, many CFI metadata only needs write protection without concerning about its secrecy. Therefore, these CFI mechanisms do not need IH. In contrast, some CFI metadata is writable, as it needs to be dynamically updated [8, 41, 42, 37], and others need to be kept as secrets [40, 61, 53, 60]. These CFI mechanisms must protect their metadata either by memory isolation [8, 53, 42, 41, 37] or IH [40, 60, 61]. SafeHidden can be applied to improve the security of IH for these CFI mechanisms.

Intra-process isolation. SFI is commonly used to restrict intra-process memory accesses [55]. However, both software-only and hardware-assisted SFIs incur high performance overhead [20, 43]. SeCage uses double-EPT to protect sensitive data, e.g., the session key and the private key [34]. Shreds [11] utilizes the domain-based isolation support provided by the ARM platform to protect the thread-sensitive data. Intel software guard extension (SGX) [2] protects the sensitive data using a secure enclave inside the application which cannot be accessed by any code outside the enclave. However, none of the approaches mentioned above can be used to protect frequently accessed safe areas because of their high switching overhead.

Tracking TLB misses. Intel performance monitoring units (PMU) [2] can be used to profile the TLB miss, but it is not precise enough. In contrast, setting reserved bits in PTE can help to track the TLB miss precisely. Some works had used this feature for performance optimization [17, 6, 5]. SafeHidden extends this method to detect side-channel attacks against the safe areas, which is the first time to our best knowledge such a feature is used in security.

Trap areas as security defenses. Booby-traps [12] first proposes to defeat code reuse attacks by inserting the trap gadgets in applications. CodeArmor [10] inserts the trap gadgets into the virtual (original loaded) code space. To protect the secret table's content against probing attacks, Readactor++ [14] inserts trap entries into the PLT and vtable, and Shuffler [59] inserts the trap entries into its code pointer table. To defeat the JIT-ROP [49] attacks, Heisenbyte [51] and NEAR [58] propose to trap the code after being read. Different from these works, SafeHidden uses the trap to capture the probing attacks against IH.

TSX for Security. The TSX is proposed to improve the performance of multi-threaded programs, but many studies have utilized TSX to improve system security. For example, Mimosa [23] uses TSX to protect private keys from memory disclosure attacks. TxIntro leverages the strong atomicity to ensure consistent and concurrent virtual machine introspection (VMI) [33]. In addition, TSX has been used to perform or detect the side-channel attacks against the Kernel ASLR [27] or the enclave in SGX [48, 9]. Different from these works, SafeHidden uses the TSX to identify TLB entries.

EPT for Security. The EPT has been used to isolate VMs [26], to protect processes from the malicious OS and/or other processes [18, 50, 24], and to protect sensitive code/-data within a process [34]. The EPT also supports more restrict memory permission check (i.e., the execute-only permission), which has been used to prevent the JIT-ROP [49] attacks [51, 13, 58]. Different from prior works, SafeHidden uses the EPT to achieve the *thread-private* memory.

9 Conclusion

This paper presented a new IH technique, called SafeHidden, which is transparent to existing defenses. It re-randomizes the locations of safe areas at runtime to prevent attackers from persistently probing and inferring the memory layout to locate the safe areas. A new *thread-private* memory mechanism is proposed to isolate the *thread-local* safe areas and prevent the adversaries from reducing the randomization entropy via thread spraying. It also randomizes the safe areas after the TLB miss event to prevent the cache-based side-channel attacks. The experimental results show that our prototype not only prevents all existing attacks successfully but also incurs low performance overhead.

Acknowledgments

We are grateful to our shepherd Mathias Payer for guiding us in the final version of this paper. We would like to thank the anonymous reviewers for their insightful suggestions and comments. This research is supported by the National High Technology Research and Development Program of China under grant 2016QY07X1406 and the National Natural Science Foundation of China (NSFC) under grant U1736208. Pen-Chung Yew is supported by the National Science Foundation under the grant CNS-1514444. Yinqian Zhang is supported in part by gifts from Intel and DFINITY foundation.

References

[1] Kernel page-table isolation. https://www.kernel.org/doc/html/latest/x86/pti.html.

[2] Intel corporation. intel 64 and ia-32 architectures software developer's manual.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), CCS '05, ACM.

[4] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*

[5] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News* (2013).

[6] BHATTACHARJEE, A. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46.

[7] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting applications into reduced-privilege compartments. In *the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008).

[8] BUROW, N., MCKEE, D., A. CARR, S., AND PAYER, M. Cfixx: Object type integrity for c++. In *NDSSS 2018.*

[9] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ASIA CCS '17.

[10] CHEN, X., BOS, H., AND GIUFFRIDA, C. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy* (2017).

[11] CHEN, Y., REYMONDJOHNSON, S., SUN, Z., AND LU, L. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy* (2016).

[12] CRANE, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Booby trapping software. In *NSPW* (2013), ACM, pp. 95–106.

[13] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A. R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy* (2015).

[14] CRANE, S. J., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., DE SUTTER, B., AND FRANZ, M. It's a trap: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security* (2015).

[15] DAVI, L., LIEBCHEN, C., SADEGHI, A., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[16] FRASSETTO, T., JAUERNIG, P., LIEBCHEN, C., AND SADEGHI, A.-R. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium.*

[17] GANDHI, J., BASU, A., HILL, M. D., AND SWIFT, M. M. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News* (2014).

[18] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03.

[19] GAWLIK, R., KOLLENDA, B., KOPPE, P., GARMANY, B., AND HOLZ, T. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23nd Annual Network and Distributed System Security Symposium, NDSS* (2016).

[20] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium*.

[21] GRAS, B., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*.

[22] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. Aslr on the line: Practical cache attacks on the mmu. In *NDSS* (2017).

[23] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy* (2015).

[24] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[25] HSU, T. C.-H., HOFFMAN, K., EUGSTER, P., AND PAYER, M. Enforcing least privilege memory views for multithreaded applications. In *the 2016 ACM Conference on Computer and Communications Security*.

[26] JAMES E. SMITH AND RAVI NAIR. *Virtual machines - versatile platforms for systems and processes*. Elsevier, 2005.

[27] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[28] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *CoRR abs/1801.01203* (2018).

[29] KONING, K., CHEN, X., BOS, H., GIUFFRIDA, C., AND ATHANASOPOULOS, E. No need to hide: Protecting safe regions on commodity hardware. In *the Twelfth European Conference on Computer Systems* (2017).

[30] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *the 11th USENIX Conference on Operating Systems Design and Implementation* (2014).

[31] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), SIGMETRICS '10.

[32] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[33] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)* (2014).

[34] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[35] LU, K., LEE, W., NÜRNBERGER, S., AND BACKES, M. How to make ASLR win the clone wars: Runtime re-randomization. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016*.

[36] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15.

[37] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security* (2015), ACM.

[38] MERRIFIELD, T., AND TAHERI, H. R. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2016), VEE '16.

[39] MOGOSANU, L., RANE, A., AND DAUTENHAHN, N. Microstache: A lightweight execution context for in-process safe region isolation. In *RAID* (2018).

[40] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque control-flow integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*.

[41] NATHAN BUROW, X. Z., AND PAYER, M. Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy* (2019).

[42] NIU, B., AND TAN, G. Per-input control-flow integrity. In *the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[43] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *25th USENIX Security Symposium*.

[44] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *6th Cryptographers' Track at the RSA conference on Topics in Cryptology* (2006).

[45] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically Returning to Randomized lib(c). In *ACSAC* (2009).

[46] SCHAIK, S. V., RAZAVI, K., GRAS, B., BOS, H., AND GIUFFRIDA, C. Revanc: A framework for reverse engineering hardware page table caches. In *European Workshop on Systems Security* (2017).

[47] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *the 11th ACM Conference on Computer and Communications Security* (2004).

[48] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[49] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Security and Privacy 2013* (2013).

[50] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *the 7th Symposium on Operating Systems Design and Implementation* (2006).

[51] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. J. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[52] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., DUARTE, N. O., GARG, D., AND DRUSCHEL, P. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *ArXiv e-prints* (2018).

[53] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*.

[54] VUSEC. Reverse engineering page table caches in your processor, 2017. https://github.com/vusec/revanc.

[55] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993).

[56] WANG, X., ZANG, J., WANG, Z., LUO, Y., AND LI, X. Selective hardware/software memory virtualization. In *the 7th ACM Conference on Virtual Execution Environments* (2011).

[57] WANG, Z., WU, C., LI, J., LAI, Y., ZHANG, X., HSU, W.-C., AND CHENG, Y. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In *the 13th ACM Conference on Virtual Execution Environments* (2017).

[58] WERNER, J., BALTAS, G., DALLARA, R., OTTERNESS, N., SNOW, K. Z., MONROSE, F., AND POLYCHRONAKIS, M. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ASIA CCS '16.

[59] WILLIAMS-KING, D., GOBIESKI, G., WILLIAMS-KING, K., BLAKE, J. P., YUAN, X., COLP, P., ZHENG, M., KEMERLIS, V. P., YANG, J., AND AIELLO, W. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Conference on Operating Systems Design and Implementation*.

[60] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy* (2013).

[61] ZHANG, M., AND SEKAR, R. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *the 31st Annual Computer Security Applications Conference* (2015).