Making Information Hiding Effective Again

Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, *Fellow, IEEE*, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi

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.

Index Terms—Side Channel Attacks, Information Hiding, Intra-process Memory Isolation, TLB Misses.

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 [1], [2] 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 [3] uses IH to protect all targets of indirect control transfer instructions; CPI [4] uses IH to protect all sensitive pointers; ReRanz [5], Shuffler [6], Oxymoron [7], Isomeron [8] and ALSR-Guard [9] use IH to protect the randomization secrets.

For a long time, IH was considered very effective. However, recent advances of software attacks [10], [11], [12], [13], [14] have made it vulnerable again. Some of these attacks use special system features to avoid system crashes when scanning the memory space [11], [12]; some propose new techniques to gauge the unmapped regions and infer the location of a safe area [13]; 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 [10]; others suggest that cache-based side-channel attacks can be used to infer the location of safe areas [14]. 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 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*

- *Z. Wang, C. Wu, B. Tang, M. Xie, Y. Lai, Y. Kang are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, and with the University of Chinese Academy of Sciences, Beijing 100190, P.R. China.*
  Chenggang Wu is the corresponding author.
  E-mail: {wangzhe12,wucg,tangbowen,xiemengyao,laiyuanming,kangyan}@ict.ac.cn
- *Yinqian Zhang is with the Ohio State University.*
  E-mail: yinqian@cse.ohio-state.edu
- *Pen-Chung Yew is with the Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, MN 55455.*
  E-mail: yew@cs.umn.edu
- *Yueqiang Cheng is with the Baidu USA.*
  E-mail: chengyueqiang@baidu.com
- *Zhiping Shi is with the College of Information Engineering, Capital Normal University, and with the Beijing Key Laboratory of Light Industrial Robot and Safety Verification, Beijing 100048, China.*
  E-mail: shizp@cnu.edu.cn

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 [15]) 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) [16]. 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) [15]. 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.

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.

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 [1], [2] 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 [11]. CROP [12] chooses to use the exception handling mechanism to avoid crashes. During the probing, an access violation will occur when an inadmissible address is accessed.

1. It still conforms to the access model. The designated register is %rsp, and the offset is equal to 0. The only difference is that it will change the value of the designated register.

Defense	Protected Objects	Reg	Content in Protected Objects	AF	TL
O-CFI [3]	Bounds Lookup Table	%gs	The address boundaries of basic blocks targeted by an indirect branch instruction.	High	No
RERANZ [5]	Real Return Address Table	%gs	The table that contains the return addresses pushed by call instructions.	High	Yes
Isomeron [8]	Execution diversifier data	%gs	The mapping from the randomized code to the original code.	Hign	No
ASLR-Guard [9]	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 [7]	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 [6]	Code pointer table	%gs	The table that contains all indexes that are transformed from all function pointers at their initialization points.	High	No
CFCI [17]	Protected Memory	%gs	File name and descriptors, and the mapping between file names and file descriptors.	Low	No
CPI [4]	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 IH. **AF** is short for *Access Frequency*. **TL** is short for *Thread Local*.

But, the event 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 [13], 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 [10], 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 caches* by Intel [15]. In addition, the accessed PT entries are also fetched into the last level cache (LLC) during the PT walk.

It has been demonstrated that cache-based side-channels can break coarse-grained address space layout randomization [14]. 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 attack [18] 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 lines mapped by these PTEs are not changed.*

2.3 Strict Intra-process Memory Isolation

Compared to information hiding, intra-process memory isolation could provide a stronger security guarantee in protecting the sensitive data used in the defense mechanisms [19], [20], [21], [22].

Intra-process memory isolation can be categorized into address-based and domain-based isolation. Address-based isolation checks (e.g., bound-check) each memory access from untrusted code to ensure that it cannot access the safe area. The main overhead of this method is brought by the code that performs the checks. Intel provides MPX (with dedicated registers and instructions) for efficient bound-checking, thus offering the most efficient address-based isolation [23]. But it is still not practical due to high-performance overhead: When protecting the safe area of O-CFI using the MPX-based scheme (i.e., address-based), the runtime overhead is 26.63% [22]. Another disadvantage of this scheme is that it is not safe, i.e., un-instrumented instructions can still access the safe area [24].

Domain-based isolation instead controls the access permission of the safe areas. The permission to access this region is granted when requested by the trusted code and is revoked when the access is finished. However, memory accesses from the untrusted code cannot enable the access permission. The main source of the performance overhead of domain-based isolation is the operations for enabling and disabling the memory-access permissions. So, many works proposed to use existing hardware features to accelerate the permission switching overhead. For example, SEIMI uses the *Supervisor-mode Access Prevention*(SMAP) [22], ERIM uses the *Memory Protection Keys* (MPK) [19] in Intel, and Secage uses the VMFUNC hardware feature [25]. Among them, the most efficient domain-based isolation is to use the SMAP [22]. But it is still not efficient enough to protect the frequently accessed safe areas. For example, it incurs 18.29% and 12.49% performance overhead when protecting the O-CFI and the Shadow Stack, repectively [22]. When using the MPK-based scheme, it incurs more performance overhead than the SMAP-based scheme — it incurs 34.83% and 21.08% performance overhead when applied to the O-CFI and the Shadow Stack to protect the SPEC CPU 2006 benchmark, respectively [22]. But existing works, such as

ERIM [19], showed that the MPK-based scheme is fairly fast. For example, ERIM reported that it only incurs 5.3% performance overhead when applied to the CPI to protect the SPEC CPU 2006 benchmark [4]. The reason why the MPK-based scheme has worse performance behavior on O-CFI and Shadow Stack than CPI is due to the different frequency of the permission switching — the more frequently switching, the higher performance overhead. We conducted a statistical experiment on SPEC and found the average frequency of permission switching (per milliseconds) of O-CFI, Shadow Stack, and CPI is 9,018.09, 3,272.28, and 61.59, respectively. So, the scalability of the domain-based isolations on different defenses is not good.

The overhead of neither the address-based isolation nor the domain-based isolation is acceptable for a deployed mechanism [26]. Besides, the existing strict memory isolations also face other deployment issues. The address-based isolation will increase the code size greatly due to the instrumentation of all memory access instructions [21]. The domain-based isolation suffers from the limited number of domains. For example, the SMAP-based scheme only supports two domains [22]; the MPK-based scheme only supports 16 domains [19]. Although the libmpk provides an unlimited number of domains through virtualizing the protection keys of MPK, the domain switching overhead will increase to the overhead of a syscall at most.

So, some works proposed to modify the architecture to provide strong and efficient memory isolation [24], [27]. IMIX [24] extends the x86 ISA with a new memory-access permission to mark safe areas as security-sensitive and allows access to safe areas only using a newly introduced instruction. Similarly, MicroStache [27] achieves it by modifying the Gem5 simulator. However, IMIX and MicroStache are not yet supported by commodity hardware.

2.4 Motivation of SafeHidden

The IH techniques offer efficient but weaker protection, and strict memory isolation techniques offer strong but expensive protection. So the developers face a dilemma that how to strike a balance between the performance and security in their defenses to protect the safe areas. Our solution to this problem is to harden the more efficient information hiding technique to make it secure and effective again. Our key insight is to perform attack detection and re-randomization for information hiding, which only introduce performance overhead when attacks occur. As such, no additional overhead is observed in the normal use of the system.

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 §2.2). We also assume the implementation of defense system itself is not vulnerable, and it uses IH correctly: That is, (1) any access to the safe area is all done through an offset from a dedicated register; (2) the offset value can be leaked and manipulated by attackers, but the dedicated register cannot be leaked (e.g., it cannot be stored in the regular memory); (3) the safe area should be designed to be very small, and defenses should allocate in advance for a large enough area to store a safe area that may be grown, e.g., the whole memory space of a shadow stack should be allocated in advance; (4) there is no pointer pointing to the safe area that exists in the regular memory space, including obtaining the address indirectly through some calculations. For example, the safe area has a fixed offset from the .text section, attackers could use the location of the .text section and the leaked offset value to infer the location of the safe area. 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 (i.e., web servers and browsers). 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 causing crashes, e.g., by using resumable exceptions;
- **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

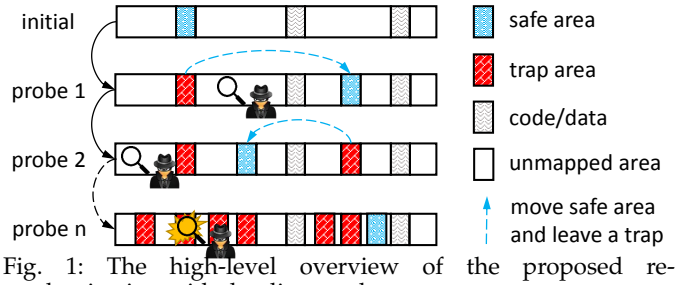


Fig. 1: The high-level overview of the proposed re-randomization with the dispersed trap areas.

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.

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 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 on purpose. 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 §4.1), how SafeHidden achieves the *thread-private* memory (see §4.2) and how SafeHidden defeats cache-based side-channel analysis (see §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 to make the safe areas' locations 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 safe areas through memory access instructions, it will disassemble the instruction to judge if the memory operand is a legal offset from the dedicated register. If it is illegal, SafeHidden will trigger a security alarm. Note that SafeHidden cannot disassemble and judge the access pattern if the execution of a memory access instruction does not trigger a page-fault exception.
- 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.

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*	–/Alarm	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.

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 [28], [29], [30]. 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) in Intel processors, or the *nested page table* (NPT) in AMD processors. Using Intel's

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.

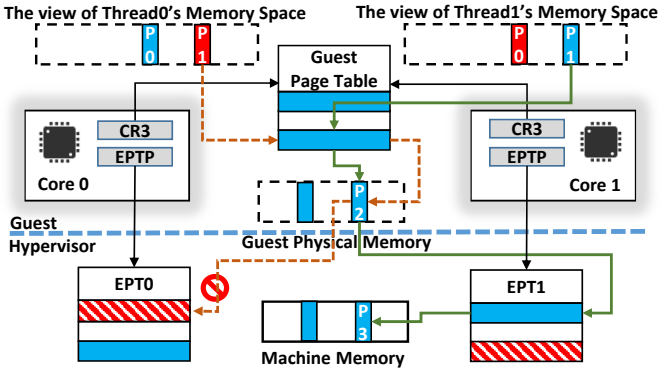


Fig. 2: An example of the *thread-private* memory mechanism. P0 and P1 is *thread-private* memory page of Thread0 and Thread1.

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 two threads of the same program, both the virtual CR3 registers point to the PT 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 §2.2.2, a key step in the cache side-channel attack by Gras et al. [14] 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 §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 lines mapped by these PTEs are also re-randomized that completely defeating cache-based side-channels [14].

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 [15]. 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) [16] 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 §5.

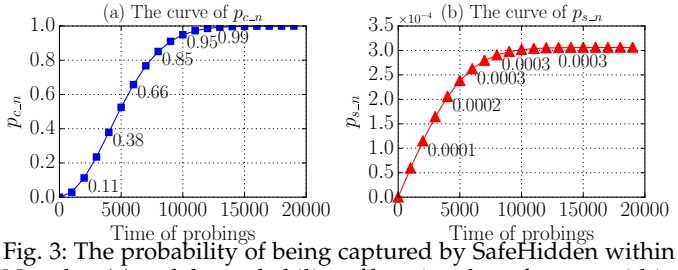


Fig. 3: The probability of being captured by SafeHidden within N probes (a) and the probability of locating the safe areas within N probes successfully (b).

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.

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

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).

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

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; (3) the access pattern check occurred in the safe areas is ignored. 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 §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.

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

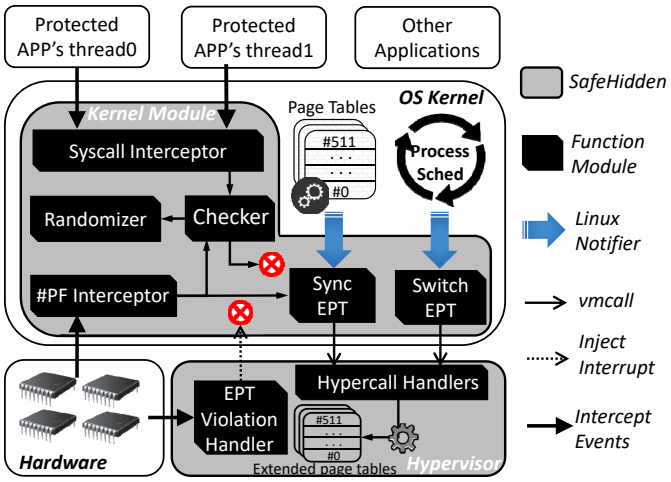


Fig. 4: Architecture overview of SafeHidden.

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 call*s, 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.

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, the reserved bits in its PTEs will be set.

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. 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 §4.1 and §4.3, when GuestKM needs to perform randomization, it invokes the optimized 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 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 [16] was introduced into the mainstream Linux kernels to mitigate the Meltdown attack [31]. For each procecss, it splits the page table into a user-mode page table and a kernel-mode page table. 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.

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.

Moreover, to avoid flushing TLB entries when switching page tables, the kernel leverages the Process Context Identifier (PCID) feature [15]. 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 owner of a TLB entry. The kernel assigns different PCIDs to the user and kernel mode page tables (i.e., uPCID and kPCID respectively). When entering or exiting the kernel mode, the kernel needs to switch between kPCID and uPCID.

As mentioned in §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.

To address this problem, we propose a method that binding uPCID with the kernel-mode page table temporarily: SafeHidden 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.

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 the accessing to the safe area pages is placed in a transaction (between `xbegin` and `xend` instructions). The KPTI flushes the TLB in a lazy mode: If the page tables are modified, it does not flush the TLB immediately. It delays the flush operation until returning to the user mode. The reloading operation of TLB entries will be done after randomization but before TLB flush. So the reloaded TLB entries will be flushed during the TLB flush operation. To address this problem, we flush the TLB immediately after randomization instead of flushing the TLB in the lazy mode.

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. 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/safe/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 also monitors the page fault exception triggered in the safe areas. If a page fault exception occurred, GuestKM will disassemble the fault instruction and judge if the memory operand is the legal access pattern or not (i.e., if it contains the dedicated register or not). If it is illegal, GuestKM will trigger a security alarm. Note that because the reserved bits are set in the safe areas, all TLB misses in the safe areas trigger this check, which decreasing the success probability of memory probing further.

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.

5.4 Optimizations in SafeHidden

In this subsection, we introduce some optimizations to reduce the performance overhead in SafeHidden.

Disabling VPID. The Virtual Processor Identifier (VPID) is intended for avoiding TLB flush during the context switch between the guest and the host. This is done by assigning a unique VPID for each guest VM and the host, and they can only access their TLB entries which are grouped by VPID. In SafeHidden, since the hypervisor is a part of the kernel module of SafeHidden, it shares the same virtual address and physical address with the kernel. So there is no need to assign VPID for the VM and the hypervisor. And disabling VPID can also enlarge the capacity of TLB used in the VM which is the key factor that affects the number of TLB misses in the safe areas.

Disabling EPT when there are no thread-local safe areas. In SafeHidden, the EPT is used to achieve the thread-private memory. But the two-dimensional paging mechanism will incur the high-performance overhead when the TLB misses are very frequent [32]. So disabling EPT at some situations could gain performance benefit. Actually, for the single-threaded programs and the multi-threaded programs without thread-local safe areas, there is no requirement for the EPT. So SafeHidden only enables the EPT mechanism when it detects the thread-local safe areas were allocated.

Only testing reloaded TLB entries in TSX. SafeHidden uses Intel TSX to test which PTEs of the safe areas are loaded in the TLB during the randomization of the safe areas. Scanning all PTEs of safe areas in the transaction of Intel TSX is a time-consuming task. In fact, not all PTEs of the safe area need to be tested. Since SafeHidden is responsible for loading the PTE of safe areas into TLB, SafeHidden only tests the PTEs that were reloaded in the last re-randomization.

Avoiding parsing the arguments in I/O system calls. There are a lot of I/O system calls in the Linux kernel. Hooking all these system calls and parsing the arguments to identify the type of access memory area is complicated and time-consuming. Since all these system calls will access the user memory space in the kernel-mode page table under KPTI, we could monitor page fault exceptions of the user memory space in the kernel mode to capture the memory access of I/O system calls. For example, if a system call accesses the unmapped area or the trap area, the page fault exception will be raised. Thanks to the KPTI and the reserved bits set in the safe areas, the page fault exception will be also raised when these system calls access the safe areas. This is because SafeHidden will not load the PTEs of the safe areas with kPCID into TLB, any access to the safe areas in the kernel-mode will raise the page fault exception.

Fast checking access pattern in the page fault exception. Since the dedicated register is fixed, there is no need to disassemble the fault instruction during the page fault exception occurred in the safe areas to check if it is illegal or not. SafeHidden only matches the unique bytes in the encoding of the fault instruction. For example, if the dedicated register is the `%gs` segmentation register, SafeHidden only checks the first byte of the fault instruction is `0x65` or not. This is because the `%gs` register is the segment override prefix in the X86 ISA, and the `0x65` is not encoded into any instruction opcodes. So SafeHidden could use this unique byte to check the access pattern.

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. SafeHidden is implemented with a total of 7,822 LoC (Lines of Code), including the 186 lines assembly code and the

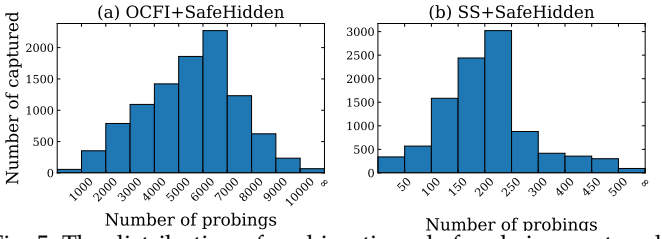


Fig. 5: The distribution of probing times before being captured (10,000 probes launched).

7,636 lines C language code. 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 [3], 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. *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 [4], ASLR-Guard [9], and RERANZ [5]. *SS* adopts a compact shadow stack scheme [21] (in contrast to a parallel shadow stack scheme). To be compatible with uninstrumented libraries, *SS* instruments function prologues and epilogues to access the shadow stacks (i.e., the safe areas). 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.

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 [13] 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 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 [12] 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, and these probes are all captured by trap areas. Figure 5 (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 5 (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 [33], which is a tool based on [34]. 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, disk I/O, and jitter respectively. In this section, we only evaluate the performance of SafeHidden with all optimizations enabled. In §6.3, we will evaluate the impact of optimizations on SafeHidden specially.

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 the experiment of jitter, we chose the SpiderMonkey (v.59.0a1.0) which is the JavaScript engine of Firefox. For each benchmark, we prepared two versions of the benchmark: (1) protected by *SS*, and (2) protected by *OCFI*. We

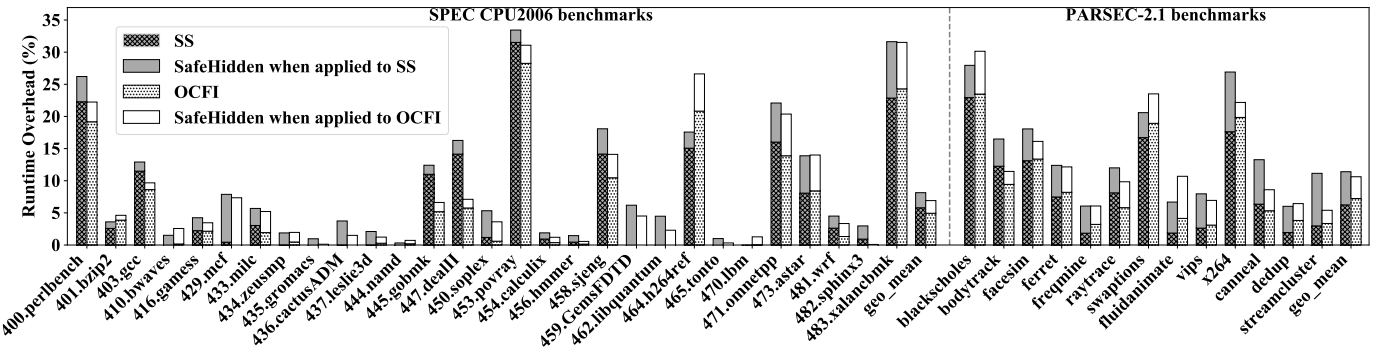


Fig. 6: Performance overhead of SPEC and Parsec-2.1 benchmarks brought by SafeHidden when applied to SS and OCFI.

Program	#randomization		Details of #randomization				Program	#randomization		Details of #randomization			
	SS	OCFI	#brk()	#mmap()	#tlb_miss SS	#tlb_miss OCFI		SS	OCFI	#brk()	#mmap()	#tlb_miss SS	#tlb_miss OCFI
SPEC CPU2006 benchmark													
perlbench	436,517	125,340	7,957	100	428,460	117,283	povray	521	335	26	30	465	279
bzip2	4,972	3,318	36	100	4,836	3,182	calculix	42,562	40,443	32,095	139	10,328	8,209
gcc	165,286	148,649	6,816	194	158,276	14,1639	hmmer	3,839	2,337	13	25	3,801	2,299
bwaves	23,684	29,140	701	45	22,938	28,394	sjeng	229,414	210,149	3	10	229,401	210,136
gamess	25,487	35,489	27	30	25,430	35,432	GemsFDTD	192,209	175,121	11	160	192,038	174,950
mcf	418,379	441,804	3	11	418,365	441,790	libquantum	414,355	219,887	14	39	414,302	219,834
milc	640,095	546,554	2,687	44	637,364	543,823	h264ref	7,839	3,095	545	60	7,234	2,490
zeusmp	9,743	10,294	3	10	9,730	10,281	tonto	12,452	10,241	298	20	12,134	9,923
gromacs	6,663	8,473	44	36	6,583	8,393	lbm	6,252	7,218	3	11	6,238	7,204
cactusADM	149,799	121,272	8,997	66	140,736	112,209	omnetpp	343,126	238,111	1,245	56	341,825	236,810
leslie3d	17,960	18,134	5	27	17,928	18,102	astar	931,365	714,905	3,928	46	927,391	710,931
namd	866	750	100	31	735	619	wrf	72,608	62,645	419	253	71,936	61,973
gobmk	11,692	89,923	59	594	11,039	89,270	sphinx3	4,398	3,483	144	146	4,108	3,193
dealII	50,392	113,977	40,103	53	10,236	73,821	xalancbmk	952,216	835,238	3,099	94	949,023	832,045
soplex	185,070	192,956	168	49	184,853	192,739	**average**	**184,819**	**152,044**	**3,778**	**85**	**180,956**	**148,181**
Parsec-2.1 benchmark													
blackscholes	149,408	98,318	3	22	149,383	98,293	fluidanimate	159,641	192,288	231	23	159,387	192,034
bodytrack	13,871	11,890	2,486	6,558	4,827	2,846	vips	14,217	15,961	4	115	14,098	15,842
facesim	38,318	21,360	359	69	37,890	20,932	x264	7,132	9,407	42	162	6,928	9,203
ferret	106,646	63,384	222	39,032	67,392	24,130	canneal	294,270	289,342	5,917	24	288,329	283,401
freqmine	5,856	3,582	499	64	5,293	3,019	dedup	73,322	84,120	1,571	715	71,036	81,834
raytrace	30,266	29,166	1,279	57	28,930	27,830	streamcluster	301,924	290,417	7	23	301,894	290,387
swaptions	8,128	5,857	3	22	8,103	5,832	**average**	**92,538**	**85,776**	**971**	**3,607**	**87,961**	**81,199**

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

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 6 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 *blackscholes*) 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 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 1.97% and 2.35%, respectively. For Parsec benchmarks, the geometric mean performance overhead incurred by SafeHidden is 3.39% and 5.16%, 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 6, we can see that for SPEC benchmarks, the performance overhead is related to the total number of re-randomization. Except *x264* using *SS*, the overhead of most Parsec benchmarks is also related to the total number of re-randomization. *x264* spawns child threads more frequently than other benchmarks, which causes SafeHidden to frequently create and initialize *thread-private* EPTs. We can also see that SafeHidden incurs much higher overhead for *canneal* and *streamcluster* when protecting *SS* than *OCFI*. This is because SafeHidden needs to enable EPT to isolate

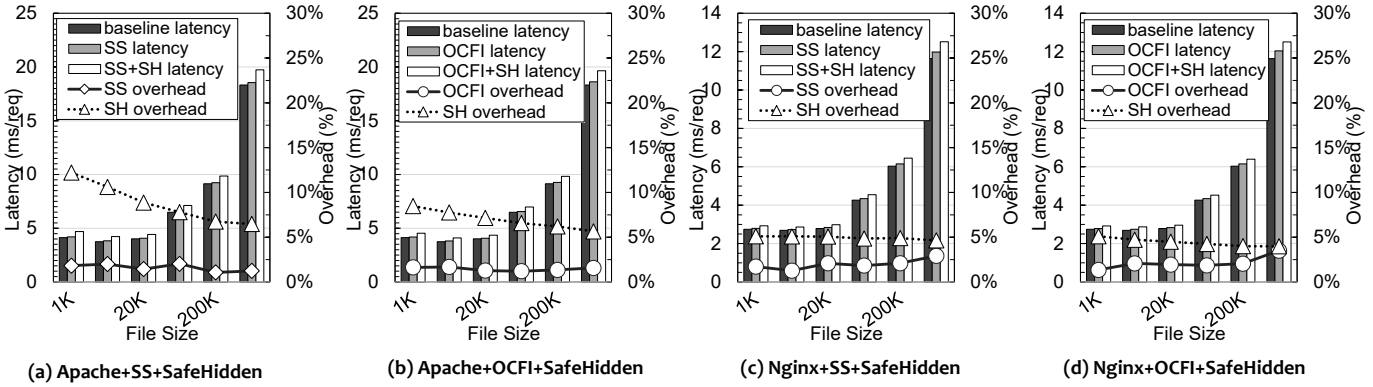


Fig. 7: Network I/O performance overhead brought by SafeHidden (short for SH) when applied to the SS and OCFI defenses.

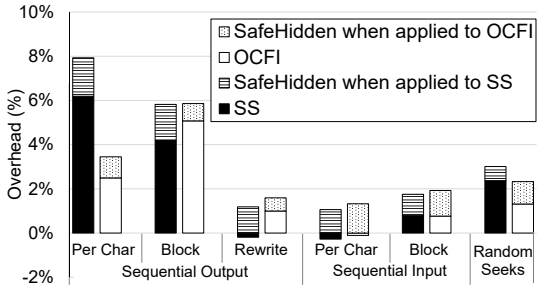


Fig. 8: Disk I/O performance overhead brought by SafeHidden when applied to the SS and OCFI defenses.

thread-private safe areas when protecting *SS*, but EPT is disabled in SafeHidden when protecting *OCFI*. In the two-dimensional page table mechanism, the more TLB misses triggered, the more memory virtualization overhead will be incurred. Compared with other benchmarks in Parsec, *canneal* and *streamcluster* trigger more TLB misses during their execution. So the reason why the overhead difference is so great between *SS* and *OCFI* is due to the memory virtualization overhead in EPT.

6.2.2 Network I/O Performance Evaluation

Figure 7 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 8.78% and 6.97% on average when applied to *SS* and *OCFI* to protect Apache. The reason why SafeHidden is more performant when protecting *OCFI* than *SS* is also due to the memory virtualization, which is already discussed in §6.2.1.

SafeHidden incurs only 4.96% and 4.42% 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 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

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 8 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 1.20% overhead on average for *SS* and 2.73% 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.

6.2.4 JavaScript Engine Performance Evaluation

We evaluated the SpiderMonkey with the Kraken benchmark [35] from Mozilla, which is widely used to test realistic workloads. We evaluated each of the 14 test suites in Kraken and calculated the geo_mean of the overheads. *OCFI* and *SS* incur 1.75% and 1.84% overhead on average when protecting SpiderMonkey. SafeHidden incurs 4.27% and 5.49% on average when applied to *OCFI* and *SS* to protect SpiderMonkey. The overhead of three test suites (i.e., *audio-beat-detection*, *audio-fft*, and *audio-oscillator*) is higher than other test suites obviously. When protecting *OCFI*, SafeHidden incurs 7.29%, 6.68%, and 8.03% overhead on these three test suites, respectively; when protecting *SS*, SafeHidden incurs 8.41%, 7.94%, and 9.26% overhead, respectively. This is because these test suites trigger SpiderMonkey to allocate a large amount of memory via using the `mmap()` system call which results in a frequent re-randomization.

6.3 Performance Evaluation of Optimizations

In this section, we evaluate the impact of some optimizations mentioned in §5.4 on performance overhead. As shown in Table 4, we design six optimization levels in

Optimization Levels	Disabling VPID	Disabling EPT when there are no thread-local safe areas	Testing reloaded TLB entries in TSX	Avoiding parsing the arguments in I/O system calls	Fast checking access pattern in the page fault exception
-O0					
-O1	✔				
-O2	✔	✔			
-O3	✔	✔	✔		
-O4	✔	✔	✔	✔	
-O5	✔	✔	✔	✔	✔

TABLE 4: Which optimizations are enabled among the different optimization levels in SafeHidden.

SafeHidden. The optimization level **-O0** disables all optimizations, and the optimization level **-O5** enables all optimizations. From the **-O0** to the **-O5**, each optimization level only enables one more optimization compared to a lower optimization level. For example, the **-O3** enables an extra optimization (named *Testing reloaded TLB entries in TSX*) compared to the **-O2**.

6.3.1 The Impact on CPU Intensive Benchmarks

Table 5 shows the impact of different optimization levels in SafeHidden when applied to enhance the *OCFI* and *SS* defenses. Note that the overhead shown in the table is only brought by SafeHidden when applied to the defenses.

The optimization level **-O1** enables the optimization, *Disabling VPID*, compared to the **-O0**. Disabling VPID could enlarge the capacity of TLB used in the guest (including protected processes), which can reduce the number of TLB misses in the safe areas. In SafeHidden, the more TLB misses in the safe areas, the more number of randomization, and the higher performance overhead. So we can see that SafeHidden achieves the performance improvement for almost all SPEC and Parsec benchmarks. Particularly, the performance improvements are more significant for 7 programs (bold in the column "**-O1**") in SPEC and Parsec benchmarks, this is due to the behaviors of these programs have more frequent TLB misses.

The optimization level **-O2** enables an extra optimization, *Disabling EPT when there are no thread-local safe areas*, compared to the **-O1**. In the two-dimensional paging mechanism, the more frequent TLB misses, the more performance overhead in the memory virtualization. So similar to the **-O1**, the **-O2** achieves performance improvement for almost all benchmarks and performs better on the same 7 programs. Note that since the shadow stack of *SS* is a thread-local safe area in multi-threaded programs, the **-O2** does not disable EPT for Parsec benchmarks and achieves no performance improvement.

The optimization level **-O3** enables an extra optimization, *Testing reloaded TLB entries in TSX*, compared to the **-O2**. Testing reloaded TLB entries is only performed during the randomization of the safe areas. So the more frequent randomizations, the more performance improvement will be achieved in this optimization. Combined with Table 3, the programs which trigger more randomizations of the safe areas will have better performance improvement in Table 5 (bold in the column "**-O3**").

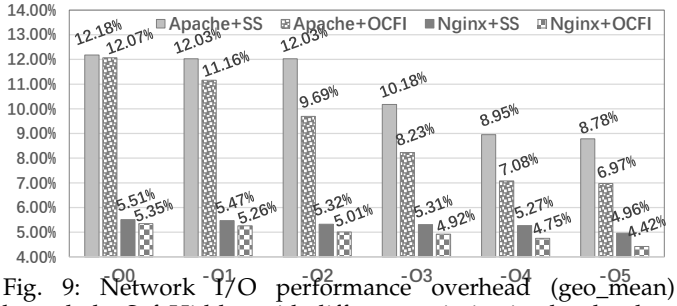


Fig. 9: Network I/O performance overhead (geo_mean) brought by SafeHidden with different optimization levels when applied to the SS and OCFI defenses.

The optimization level **-O4** enables an extra optimization, *Avoiding parsing the arguments in I/O system calls*, compared to the **-O3**. Parsing the arguments in I/O system calls is used to identify the type of access memory area, and then SafeHidden performs different actions. As shown in Table 5, this optimization achieves the performance improvement for almost all benchmarks. We can also see that some programs achieve a better performance improvement, which is related to the number of the trap areas (#brk()+#mmap() in Table 3): the more trap areas, the better performance improvement. This is because all memory areas in the user space are stored in the `vm_area_struct` structure which is stored as a tree. Identifying the type of access memory area is performed by traversing this tree. The overhead of traversing this tree is related to the complexity of the tree—the higher the complexity, the more overhead. Among the actions in SafeHidden, only leaving the trap areas will increase the complexity of that tree.

The optimization level **-O5** enables an extra optimization, *Fast checking access pattern in the page fault exception*, compared to the **-O4**. In SafeHidden, all TLB miss events occurred in the safe areas will trigger the page fault exception. SafeHidden intercepts this exception and then checks if the fault instruction is illegal or not. This optimization will improve the performance of this process, so the more TLB misses occurred in the safe areas, the more performance improvement will be achieved by this optimization. Combined with the column "#tlb_miss" in Table 3, we can see that our theoretical analysis is supported by the experimental results.

6.3.2 The Impact on Network I/O Benchmarks

Figure 9 shows the performance overhead brought by SafeHidden with different optimizations when applied to enhance the SS and OCFI defenses to protect the Apache and Nginx. We can see that every optimization could bring performance improvement for SafeHidden. We can also see that the optimization level **-O2** achieves no performance improvement on the case, Apache+SS, compared to the **-O1**. This is because EPTs must be used to isolated thread-local safe areas, and they cannot be disabled. As discussed in §6.2.2, compared with Nginx, Apache could trigger more randomization of safe areas due to invoking the `mmap()` system call in each request. And this randomization event also leaves trap areas. So as shown in the figure, we can see that there are two optimizations, which are related to randomizations and trap areas, perform better in such a situation. They are *Testing reloaded TLB entries in TSX* (i.e., the **-O3** - the **-O2**) and *Avoiding parsing the arguments in I/O system calls* (i.e., the **-O4** - the **-O3**).

Program	Overhead brought by SafeHidden when applied to SS						Program	Overhead brought by SafeHidden when applied to OCFI					
	-O0	-O1	-O2	-O3	-O4	-O5		-O0	-O1	-O2	-O3	-O4	-O5
SPEC CPU2006 benchmark													
perlbench	5.31%	5.03%	4.41%	**4.26%**	4.03%	3.93%	perlbench	4.16%	4.03%	3.64%	**3.38%**	3.09%	3.07%
bzip2	1.05%	1.05%	1.03%	1.03%	1.04%	1.02%	bzip2	1.15%	1.13%	0.82%	0.81%	0.79%	0.75%
gcc	5.17%	**4.42%**	**2.47%**	2.01%	1.89%	**1.42%**	gcc	3.85%	**3.14%**	**1.29%**	1.22%	1.14%	**1.05%**
bwaves	2.41%	2.39%	2.02%	1.92%	1.92%	1.91%	bwaves	2.82%	2.81%	2.41%	2.42%	2.40%	2.41%
gamess	2.86%	2.86%	2.26%	2.03%	2.01%	2.01%	gamess	2.18%	2.15%	1.31%	1.31%	1.30%	1.31%
mcf	12.91%	**11.34%**	**8.51%**	**8.36%**	7.98%	**7.44%**	mcf	13.85%	**12.39%**	**9.82%**	**8.99%**	8.91%	**8.23%**
milc	5.26%	4.93%	3.41%	**3.09%**	3.04%	2.65%	milc	6.17%	5.92%	4.04%	**3.75%**	3.71%	3.28%
zeusmp	2.07%	2.05%	2.04%	1.95%	1.89%	1.88%	zeusmp	2.30%	2.24%	1.52%	1.51%	1.51%	1.49%
gromacs	1.73%	1.73%	1.52%	1.53%	1.51%	1.47%	gromacs	1.16%	1.14%	1.13%	1.13%	1.11%	1.11%
cactusADM	4.16%	4.16%	4.11%	4.03%	3.75%	3.74%	cactusADM	3.46%	3.41%	2.03%	2.03%	1.82%	1.82%
leslie3d	2.76%	2.77%	2.34%	2.19%	2.10%	2.10%	leslie3d	2.22%	2.18%	0.99%	0.97%	0.95%	0.95%
namd	0.81%	0.92%	0.82%	0.77%	0.78%	0.76%	namd	1.11%	1.11%	0.56%	0.56%	0.53%	0.53%
gobmk	1.74%	1.63%	1.53%	1.42%	1.42%	1.41%	gobmk	2.46%	2.35%	1.48%	1.46%	1.44%	1.44%
dealII	3.90%	3.40%	2.42%	2.39%	**2.11%**	2.13%	dealII	2.11%	2.05%	1.57%	1.57%	**1.36%**	1.35%
soplex	8.65%	**7.43%**	**5.36%**	4.84%	4.34%	**4.15%**	soplex	6.84%	**5.34%**	**3.72%**	3.57%	3.56%	**3.02%**
povray	2.10%	2.09%	2.01%	2.02%	1.93%	1.92%	povray	3.15%	3.04%	2.88%	2.83%	2.82%	2.82%
calculix	1.13%	1.13%	1.12%	1.12%	**0.95%**	0.96%	calculix	1.43%	1.41%	1.02%	1.02%	**0.83%**	0.83%
hmmer	1.10%	1.11%	1.01%	1.02%	1.01%	1.01%	hmmer	0.69%	0.67%	0.42%	0.42%	0.42%	0.42%
sjeng	5.52%	5.03%	4.63%	**4.14%**	4.13%	3.93%	sjeng	5.84%	5.51%	4.32%	**4.05%**	4.04%	3.64%
GemsFDTD	10.25%	**9.32%**	**7.04%**	7.01%	6.74%	**6.56%**	GemsFDTD	9.40%	**8.15%**	**5.58%**	5.17%	5.17%	**4.88%**
libquantum	6.83%	6.02%	5.72%	**5.11%**	5.05%	4.93%	libquantum	5.29%	4.75%	3.26%	**3.01%**	3.01%	2.75%
h264ref	2.88%	2.75%	2.63%	2.53%	2.51%	2.51%	h264ref	6.73%	6.45%	5.84%	5.83%	5.81%	5.79%
tonto	1.42%	1.43%	1.39%	1.35%	1.34%	1.33%	tonto	1.33%	1.33%	1.01%	1.01%	1.01%	0.98%
lbm	1.30%	1.31%	0.91%	0.72%	0.69%	0.69%	lbm	2.11%	2.10%	1.25%	1.25%	1.23%	1.23%
omnetpp	11.49%	**10.26%**	**7.80%**	**6.53%**	6.53%	**6.08%**	omnetpp	12.20%	**11.03%**	**7.42%**	**6.92%**	6.81%	**6.49%**
astar	10.20%	**9.43%**	**7.36%**	**6.91%**	6.19%	**5.81%**	astar	10.74%	**9.36%**	**6.34%**	**5.99%**	5.97%	**5.58%**
wrf	2.48%	2.47%	1.97%	1.88%	1.89%	1.88%	wrf	2.38%	2.35%	2.07%	2.07%	2.02%	2.02%
sphinx3	2.79%	2.80%	2.10%	2.07%	2.05%	2.05%	sphinx3	2.10%	2.09%	1.03%	1.03%	1.01%	1.01%
xalancbmk	14.24%	**13.35%**	**10.92%**	**9.84%**	9.81%	**8.77%**	xalancbmk	12.83%	**11.70%**	**8.74%**	**8.01%**	7.98%	**7.33%**
geo_mean	**3.35%**	**3.21%**	**2.68%**	**2.52%**	**2.43%**	**2.35%**	**geo_mean**	**3.30%**	**3.13%**	**2.16%**	**2.10%**	**2.04%**	**1.97%**
Parsec-2.1 benchmark													
blackscholes	6.84%	5.51%	5.51%	**5.23%**	5.19%	5.02%	blackscholes	9.28%	8.36%	7.29%	**6.73%**	6.74%	6.66%
bodytrack	5.10%	5.02%	5.02%	5.01%	**4.24%**	4.24%	bodytrack	4.32%	4.31%	2.95%	2.95%	**2.02%**	2.02%
facesim	5.19%	5.06%	5.06%	5.03%	4.96%	4.95%	facesim	4.25%	4.25%	3.01%	2.78%	2.77%	2.75%
ferret	6.44%	6.12%	6.12%	**5.89%**	**5.13%**	4.92%	ferret	7.19%	6.26%	5.92%	**5.01%**	**4.24%**	3.95%
freqmine	4.47%	4.38%	4.38%	4.34%	4.25%	4.22%	freqmine	3.11%	3.09%	2.87%	2.87%	2.87%	2.86%
raytrace	4.25%	4.19%	4.19%	3.93%	3.91%	3.89%	raytrace	5.20%	5.17%	4.73%	4.24%	4.21%	4.03%
swaptions	4.04%	4.02%	4.02%	4.01%	3.88%	3.88%	swaptions	4.84%	4.84%	4.72%	4.72%	4.72%	4.58%
fluidanimate	6.30%	6.01%	6.01%	**5.25%**	5.19%	4.82%	fluidanimate	9.87%	9.01%	7.56%	**6.89%**	6.87%	6.55%
vips	5.62%	5.43%	5.43%	5.41%	5.36%	5.33%	vips	6.27%	6.25%	3.98%	3.97%	3.95%	3.84%
x264	9.63%	9.51%	9.51%	9.48%	9.37%	9.29%	x264	3.44%	3.41%	2.37%	2.37%	2.37%	2.33%
canneal	11.83%	**9.47%**	9.47%	**8.78%**	**7.45%**	**6.92%**	canneal	14.79%	**13.11%**	**4.92%**	**4.11%**	**3.74%**	**3.27%**
dedup	6.29%	6.22%	6.22%	**5.24%**	**4.11%**	4.08%	dedup	7.25%	7.24%	3.77%	**3.05%**	**2.62%**	2.62%
streamcluster	11.74%	**9.97%**	9.97%	**9.16%**	9.04%	**8.18%**	streamcluster	13.93%	**12.49%**	**2.76%**	**2.37%**	2.33%	**2.05%**
geo_mean	**6.34%**	**5.94%**	**5.94%**	**5.65%**	**5.30%**	**5.16%**	**geo_mean**	**6.40%**	**6.12%**	**4.09%**	**3.77%**	**3.53%**	**3.39%**

TABLE 5: The impact of different optimization levels in SafeHidden when applied to enhance SS and OCFI to protect SPEC CPU2006 and Parsec-2.1 benchmarks. Table 4 shows which optimizations are enabled among the different optimization levels.

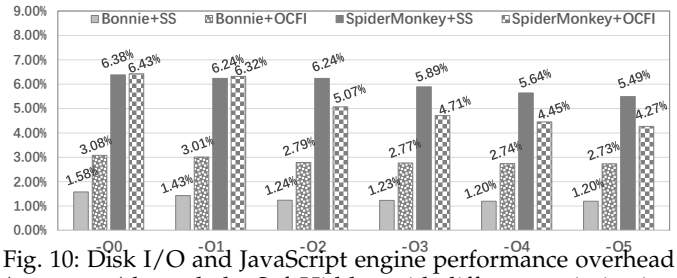


Fig. 10: Disk I/O and JavaScript engine performance overhead (geo_mean) brought by SafeHidden with different optimization levels when applied to the SS and OCFI defenses.

6.3.3 The Impact on Disk I/O Benchmarks

Figure 10 shows the performance overhead brought by SafeHidden with different optimizations when applied to enhancing the SS and OCFI defenses to protect the Bonnie++ benchmark. We can see that every optimization could bring performance improvement for SafeHidden. Similar to Nginx, since the Bonnie++ benchmark is the single-threaded program and trigger fewer number of randomization of safe areas, the optimizations do not achieve significant performance improvement compared to Apache.

6.3.4 The Impact on JavaScript Engines

Figure 10 shows the performance overhead brought by SafeHidden with different optimizations when applied to enhancing the SS and OCFI defenses to protect SpiderMonkey. We can see that every optimization could bring performance improvement for SafeHidden and the tendency is similar to Apache due to SpiderMonkey is also a multi-threaded application. Compared with Apache, we can also see that the optimization level **-O4** achieves few performance improvement compared to the **-O2**. This is because there is few I/O-related operations in the Kraken benchmark and the number of re-randomizations is much less than Apache.

7 Discussion

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* [13] 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.

Porting SafeHidden to the ARM platform. Porting SafeHidden to the ARM platform needs to re-implement some architecture-specific mechanisms, including the thread-private mechanism and the TLB miss intercepting mechanism. For the thread-private mechanism, we can use the Stage-2 page table in ARM virtualization technique to replace the EPT in Intel virtualization technique; For the TLB miss intercepting mechanism, there are still reserved bits in PTEs of ARM, but setting these bits will not trigger any exception during the hardware page table walk. ARM supports disabling the hardware page table walk but it will introduce high-performance overhead due to all TLB misses in the protected process will trigger the translation exception. So one feasible method is using the present bit instead of the reserved bits. We could clear the present bits of all PTEs in safe areas, so the page fault exception will be also triggered during the page table walk. Note that configuring the present bit to intercept the TLB misses could cause kernel crashes since the kernel performs active checks of the spresent bit in a lowest-level PTE in multiple cases. For instance, when a process is forking a new child process, the kernel checks the present bit in the process's lowest-level PTEs. To avoid such side-effects, we need to modify the kernel source code to cooperate with SafeHidden. This is also the reason why we choose the reserved bits in SafeHidden on the X86 platform — in contrast to the present bit check, the kernel does not perform any check against lowest-level PTEs' reserved bits.

8 Related Work

Other hardening the IH works. Some defense mechanisms have been proposed to harden information hiding. Shuffler [6] 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. ProbeGuard [36] detects probing attacks that try to de-randomize information hiding and patches the vulnerable code to prevent probing by exploiting the vulnerability repeatedly. This method cannot block all attack vectors, e.g., clone-probing attacks and cache-based side-channel attacks. Different from Shuffler and ProbeGuard, SafeHidden blocks all existing attack vectors against IH.

Protecting the metadatas of CFI. CFI is an important defense against code reuse attacks [37]. 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 the write protection without concern about its secrecy. For example, there is no need to use IH to protect the pure label based CFI due to the labels are stored in the unwritable code section. Therefore, these CFI mechanisms do not need IH. In contrast, some CFI metadata is writable, as it needs to be dynamically updated [21], [38], [39], [40], and others need to be kept as secrets [3], [17], [41], [42]. These CFI mechanisms must protect their metadata either by memory isolation [21], [38], [39], [40], [41] or IH [3], [17], [42]. SafeHidden can be applied to improve the security of IH for these CFI mechanisms.

Code re-randomization. Many researchers studied how to use the code re-randomization technique, which randomizes the application code on-the-fly, to mitigate the memory disclosure attacks, such as the JIT-ROP attack [43]. If code is re-randomized between the time that it is leaked and the time a payload is invoked, the attack fails because the gadgets do not longer exist. So, the time to perform the code re-randomization is the key factor to the security. OS-level ASR [44] and Shuffler [6] choose to perform re-randomization at a fixed time interval; TASR [45], ReRanz [5], and RuntimeASLR [46] choose to perform re-randomization at some syscalls that could leak the information. Although SafeHidden is proposed to protect the safe area (the secret data) not the code, it is orthogonal to these code re-randomization works.

Tracking TLB misses. Intel performance monitoring units (PMU) [15] 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 [47], [48], [49]. 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 [50] first proposes to defeat code reuse attacks by inserting the trap gadgets in applications. CodeArmor [51] inserts the trap gadgets into the virtual (original loaded) code space. To protect the secret table's content against probing attacks, Readactor++ [52] inserts trap entries into the PLT and vtable, and Shuffler [6] inserts the trap entries into its code pointer table. To defeat the JIT-ROP [43] attacks, Heisenbyte [53] and NEAR [54] propose to trap the code after being read. Different from these works, SafeHidden uses the trap to capture the probing attacks against IH.

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

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 61902374 and U1736208. Pen-Chung Yew is supported by the NSF under the grant CNS-1514444. A gift to Yinqian Zhang from Intel and DFINITY foundation.

REFERENCES

[1] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically Returning to Randomized lib(c)," in *IEEE ACSAC*, 2009.
[2] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *ACM CCS*, 2004.
[3] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS*, 2015.
[4] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI*, 2014.
[5] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng, "Reranz: A light-weight virtual machine to mitigate memory disclosure attacks," in *ACM VEE*, 2017.
[6] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *USENIX OSDI*, 2016.
[7] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security*, 2014.
[8] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.
[9] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *ACM CCS*, 2015.
[10] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX Security*, 2016.
[11] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization," in *NDSS*, 2016.
[12] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding," in *NDSS*, 2016.
[13] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *USENIX Security*, 2016.
[14] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu," in *NDSS*, 2017.
[15] "Intel 64 and ia-32 architectures software developer's manual." 2016.
[16] L. K. Document, "Kernel page-table isolation." https://www.kernel.org/doc/html/latest/x86/pti.html.
[17] M. Zhang and R. Sekar, "Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks," in *ACM ACSAC*, 2015.
[18] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *6th Cryptographers' Track at the RSA conference on Topics in Cryptology*, 2006.
[19] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, D. Garg, and P. Druschel, "Erim: Secure and efficient in-process isolation with memory protection keys," in *USENIX Security*, 2018.
[20] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *ACM EuroSys*, 2017.
[21] X. Z. Nathan Burow and M. Payer, "Shining light on shadow stacks," in *IEEE SP*, 2019.
[22] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation," in *IEEE SP*, 2020.
[23] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: An empirical study of intel mpx and software-based bounds checking approaches," 2018.
[24] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *USENIX Security*, 2018.
[25] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *ACM CCS*, 2015.
[26] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE SP*, 2013.
[27] L. Mogosanu, A. Rane, and N. Dautenhahn, "Microstache: A lightweight execution context for in-process safe region isolation," in *RAID*, 2018.
[28] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *ACM CCS*, 2016.
[29] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *USENIX NSDI*, 2008.
[30] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *ACM SIGMETRICS*, 2010.
[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.
[32] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective hardware/software memory virtualization," in *ACM VEE*, 2011.
[33] VUSec, "Reverse engineering page table caches in your processor," 2017, https://github.com/vusec/revanc.
[34] S. V. Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, "Revanc: A framework for reverse engineering hardware page table caches," in *European Workshop on Systems Security*, 2017.
[35] Mozilla, "Kraken." https://krakenbenchmark.mozilla.org.
[36] K. Bhat, E. van der Kouwe, H. Bos, and C. Giuffrida, "ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations," in *ASPLOS*, 2019.
[37] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM CCS*, 2005.
[38] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++," in *NDSS*, 2018.
[39] B. Niu and G. Tan, "Per-input control-flow integrity," in *CCS*, 2015.
[40] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: cryptographically enforced control flow integrity," in *CCS*, 2015.
[41] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *ACM CCS*, 2015.
[42] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE SP*, 2013.
[43] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *IEEE SP*, 2013.
[44] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization," in *USENIX Security*, 2012.
[45] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in *ACM CCS*, 2015.
[46] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization," in *NDSS*, 2016.
[47] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Badgertrap: A tool to instrument x86-64 tlb misses," *SIGARCH Comput. Archit. News*, 2014.
[48] A. Bhattacharjee, "Large-reach memory management unit caches," in *ACM MICRO*, 2013.
[49] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM ISCA*, 2013.
[50] S. B. Stephen Crane, Per Larsen and M. Franz, "Booby trapping software," in *ACM NSPW*, 2013.
[51] X. Chen, H. Bos, and C. Giuffrida, "Codearmor: Virtualizing the code space to counter disclosure attacks," in *IEEE EuroSP*, 2017.
[52] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a trap: Table randomization and protection against function-reuse attacks," in *ACM CCS*, 2015.
[53] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *ACM CCS*, 2015.
[54] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ACM AsiaCCS, 2016.

Zhe Wang is currently an assistant professor at Institute of Computing Technology, Chinese Academy of Sciences. His research interests are in dynamic binary translation, multi-threaded program record-and-replay, operating systems, system virtualization, and memory corruption attacks and defenses.

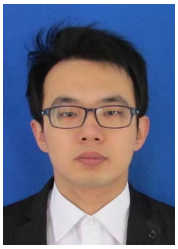
Mengyao Xie is currently working toward the PhD degree in the State Key Laboratory of Computer Architecture, Institute of Computing Technology. Her research interests include software security and virtualization.

Chenggang Wu is a professor at Institute of Computing Technology, Chinese Academy of Sciences. His research was supported by National Science Foundation of China (NSF), the National High Technology Research and Development Program of China, and the National Science and Technology Major Project of China. His research interests include the dynamic compilation, including binary translation, dynamic optimization, bug detection on concurrent program, and software security.

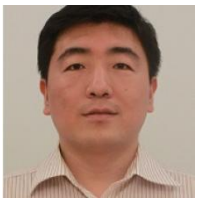
Yuanming Lai received the BS degree in Digital Media Technology from Central China Normal University in 2013, and the Master degree in Pattern Recognition and Intelligence System from Huazhong University of Science and Technology, in 2016. Now he is in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include information security and machine learning.

Yinqian Zhang is an associate professor of the Department of Computer Science and Engineering of the Ohio State University. His research interest is computer system security, with particular emphasis on cloud computing security, OS security and side-channel security.

Yan Kang received the BS degree and the Master degree in Software Engineering from Beijing University of Aeronautics and Astronautics (BUAA) in 2014 and 2017, Now she is currently working in Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include software and system security.

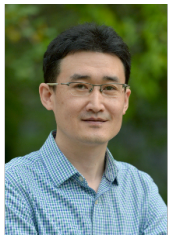


Bowen Tang is currently working toward the PhD degree in the State Key Laboratory of Computer Architecture, Institute of Computing Technology. His research interests include computer architecture, compiler optimization and software security.



Yueqiang Cheng is a Senior Staff Security Scientist at Baidu X-Lab. Yueqiang's research revolves around building secure systems and software and includes SGX security, virtualization security, rowhammer security, side-channel security, and self-driving car security.

Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization, compilers and architectural issues related multi-core/many-core systems. He is a IEEE fellow.



Zhiping Shi is a professor at Captain Normal University. He is the dean of the college of information engineering. His research interests including formal verification, machine vision and AI.