CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation

Mengyao Xie[1,2], Chenggang Wu[1,2,3], Yinqian Zhang[4,5], Jiali Xu[1,2], Yuanming Lai[1,2], Yan Kang[1,2], Wei Wang[1], Zhe Wang[1,3]✉

{xiemengyao, wucg, xujiali, laiyuanming, kangyan, wangwei2021, wangzhe12}@ict.ac.cn, yinqianz@acm.org

[1] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences

[2] University of Chinese Academy of Sciences [3] Zhongguancun Laboratory

[4] Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology

[5] Department of Computer Science and Engineering, Southern University of Science and Technology

ABSTRACT

Intel control-flow enforcement technology (CET) is a new hardware feature available in recent Intel processors. It supports the coarse-grained control-flow integrity for software to defeat memory corruption attacks. In this paper, we retrofit CET, particularly the write-protected shadow pages of CET used for implementing shadow stacks, to develop a generic and efficient intra-process memory isolation mechanism, dubbed CETIS.

To provide user-friendly interfaces, a CETIS framework was developed, which provides memory file abstraction for the isolated memory regions and a set of APIs to access said regions. CETIS also comes with a compiler-assisted tool chain for users to build secure applications easily. The practicality of using CETIS to protect CPI, CFIXX, and JIT-compilers was demonstrated, and the evaluation reveals that CETIS is performed better than state-of-the-art intra-memory isolation mechanisms, such as MPK.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Intra-process Memory Isolation, Intel CET, Memory File Abstraction

ACM Reference Format:

Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559344

Zhe Wang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559344

1 INTRODUCTION

Intra-process memory isolation is a classic approach for containing the faulty or malicious code. To defeat control-flow hijacking attacks, including code-injection attacks [6, 12, 34], code-reuse attacks [4, 29, 32], and data-only attacks [14, 16, 31], security researchers have proposed techniques such as code-pointer integrity (CPI) [18], shadow stacks [2] and CFIXX [1]. Such techniques protect sensitive pointers, return addresses, or virtual table pointers in isolated regions of a process and permit accesses to said regions only from trusted code. Another use case of intra-process isolation is the use of Just-In-Time (JIT) compilers to isolate the code cache for the JITed code from being tampered with by attackers with arbitrary memory write capabilities.

Existing intra-process memory isolation mechanisms can be roughly categorized into address-based isolation and domain-based isolation [39]. Address-based isolation restricts (e.g., bound-check) each memory access from the untrusted code to ensure that the isolated memory region cannot be accessed [17, 38], such as the Software Fault Isolation (SFI) [38]. As well as causing severe code bloat [2], a large amount of instruction instrumentation will incur high performance overhead [8]. As such, the focus of most recent studies has been on domain-based isolation, which manipulates the memory access permission of the isolated memory region instead. When the trusted code accesses said region, the access permission is enabled, and the permission is disabled after the access is finished. To improve the efficiency of the permission switching, researchers have leveraged various hardware features, such as VMFUNC [13, 17, 19, 30], memory protection keys (MPK)[2, 11, 13, 17, 27, 28, 37] and supervisor memory access prevention (SMAP) [39]. Among such schemes, the MPK-based isolation scheme is the most prominent and extensively studied, as it does not rely on hardware virtualization. However, MPK-based solutions still suffer from high performance overhead when permission switching is frequent [39, 42], e,g., CPI and shadow stack.

In this paper, we proposed a new memory isolation mechanism named CETIS, by retrofitting the shadow stack (SHSTK) mechanism in the newly introduced Intel Control-flow Enforcement Technology (CET) [15]. To ensure the integrity of return addresses, SHSTK introduces a new memory page type called the shadow stack pages (hereinafter referred to as *shstk pages*) to store return addresses. It is achieved by using an unused combination of the `R/W` bit and the

Dirty bit in the page table entry (PTE). Since the access permission of the shstk pages is read-only, regular store instructions cannot write such pages. The CALL and RET instructions update the shstk pages implicitly, and if the return addresses at the top of the main stack and the shadow stack are not matched during the function return, a hardware exception will be raised. To support error handling, SHSTK also provides a WRSS instruction to modify the return addresses stored in the shstk pages.

At present, the general belief is that SHSTK is only useful in protecting the integrity of return addresses and cannot be extended to achieve more generic isolation [8]. However, such isolation was actually achieved with CETIS in this work. CETIS places the isolated memory region on the shstk pages and ensures that the WRSS instruction—the only way to access said region—can only be used by the trusted code. Further, since the NX bit can be orthogonal with the R/W bit and the Dirty bit in PTE, the isolated memory region can also have the execution permission to protect the JITed code. Thus, CETIS provides a brand-new memory isolation abstraction that *protects the integrity of data and code from other compartments*. Additionally, as CETIS separates the isolated memory region from the shadow stacks used by CET, CETIS can work alongside CET.

However, as SHSTK and WRSS are not designed for general-purpose memory isolation, retrofitting such hardware features to build a generic and efficient memory isolation mechanism is technically non-trivial. The primary challenge stems from the constraints of using the WRSS instructions.

First, the WRSS instruction can only write fixed-sized data (i.e., 4/8 bytes) into the shstk pages, and the destination address must be 4/8 byte aligned. As such, to use WRSS, a *caching* mechanism must be developed for software: when writing data at a given address, software needs to read the data at the aligned address first, combine the data, and then write back the combined data at the aligned address. Choosing an optimal write strategy with WRSS's alignment requirement is challenging since the program is required to infer the destination address of WRSS and optionally perform the read and combine operation as needed.

Second, despite also being a memory access instruction, the WRSS instruction is not as efficient as mov. In our empirical evaluation, the latency of WRSS is around 9.3 CPU cycles, while that of mov is less than 1 CPU cycle (when no cache miss occurs). Therefore, the lower number of the WRSS instructions are, the less performance overhead it would introduce. Hence, to support use cases that continuously write small-sized data (e.g., one byte), a *buffering* approach must be adopted, wherein software needs to combine the data into a buffer and flush the buffer into the memory with WRSS as needed. However, there are challenges in maintaining the consistency of the buffer and the isolated memory region and avoiding irrelevant software from disrupting the content of the buffer.

To address the aforementioned challenges, CETIS proposes a new memory file abstraction (called *cmfiles*) for the isolated memory regions and a set of APIs to access said regions. CETIS supports two different access modes: the *read/write mode* and the *append mode*. The read/write mode can read/write data at arbitrary position with arbitrary length in a cmfile. To meet the alignment requirements, CETIS provides APIs for users to provide alignment hints. CETIS can also automatically infer the alignment of the destination position. The append mode is used for frequent updates of small-sized data. A write-combine buffer is also introduced, which is implemented in a reserved general-purpose register. A compiler-assisted tool chain is provided for maintaining the consistency of the buffers and the isolated memory region.

We implemented CETIS on the Linux/X86_64 platform. To evaluate the performance overhead in comparison with alternative solutions, we also deployed the SFI-based isolation scheme, the MPK-based isolation scheme, and CETIS to protect the two defenses: CPI [18] and CFIXX [1]. The experiments on SPEC CPU2006/2017 benchmarks and the Nginx web server revealed that CETIS achieved the lowest performance overhead on average. We also applied CETIS to protect the JITed code of the JavaScript engine, ChakraCore. The results suggest that compared with other isolation schemes, CETIS is the most efficient.

In general, the contributions of this paper are as follows:

- **A novel CET-based isolation mechanism.** We propose a new memory isolation scheme using SHSTK in CET, which can be applied to protect sensitive data in software defenses and the JITed code in JIT compilers.
- **A comprehensive study on CET's SHSTK and WRSS.** We conduct a comprehensive study on CET's SHSTK and the newly introduced WRSS instruction, in terms of their performance, architectural/micro-architectural behavior, etc.
- **A new software framework.** We developed a software framework, including a set of user-friendly and performance-optimized APIs and a compiler-assisted tool chain, for easy integration with CETIS.
- **New insights from implementation and evaluation**. We implement and evaluate a prototype of CETIS, and the results show that it outperforms the existing approaches. Our study suggests that CETIS is not only practical but efficient.

2 BACKGROUND AND RELATED WORK

2.1 Intra-process Memory Isolation

Intra-process memory isolation is usually used to protect the isolated memory region. Information hiding (IH) is a commonly used (pseudo) isolation method [41], which hides the isolated region within a wide address space, and relies on a high random entropy to keep it safe. But, recent studies [9, 20, 26] have shown that IH is no longer safe. Hence, strict memory isolation is needed to protect the isolated region [40]. The existing (strict) isolation methods can be classified into address-based and domain-based isolation.

Address-based isolation method. The address-based isolation restricts (e.g., bound-check) each memory access from the untrusted code to ensure that the isolated region cannot be accessed, such as the Software Fault Isolation (SFI) [38]. To accelerate bound checking, Intel introduced the Memory Protection Extensions (MPX) [15], which introduces BNDCU/BNDCL instruction to quickly check whether a given value is within a boundary. Due to the huge code instrumentation, the MPX-based isolation still incurs high performance overhead. When protecting the shadow stack, an average overhead of 14.57% is incurred on the SPEC CPU2006 [39]. In addition, it causes serious code bloat (e.g., the average of 41.67% [2] on SPEC).

Domain-based isolation method. The domain-based isolation changes the access permission of the isolated memory region by enabling/disabling the access permission before/after accessing it. The *mprotect()* system call is commonly used to switch the access permission. To accelerate the permission switching, the hardware features were used in recent works. Some works [2, 11, 13, 17, 27, 28, 37] used the Memory Protection Keys (MPK) [15], some works [13, 17, 19, 30] used the VMFUNC [3], and SEIMI [39] used the Supervisor Memory Access Prevention (SMAP). The MPK-based isolation was found to be the most efficient without relying on virtualization. MPK divides the user memory space into 16 domains via attaching 4-bit memory protection keys in the page table entry (PTE) and provides a `WRPKRU` instruction (taking about 27 CPU cycles) to change the access permission of each domain. However, the MPK-based isolation is still not fast enough. It incurs an overhead of 61.18% to protect the shadow stack [2].

Some works proposed the *privileged move* method [2] by introducing a new instruction to only access the isolated memory region [8, 21, 33]. IMIX [8] enables a reserved bit in PTE to identify whether the page is sensitive, and provides a new memory access instruction `SMOV` to access the sensitive pages, while other memory access instructions cannot do. MicroStache [21] follows a similar idea by introducing new `XLD/XST` instructions. HDFI [33] introduces a tag for each machine word, which indicates whether the machine word is sensitive. It adds the `SDSET1` instruction to mark the data as sensitive, and adds the `LDCHK0/LDCHK1` instruction checks whether the data tag meets the expectation when loading the target data. Notably, these approaches need to modify the simulator, and thus, cannot be deployed on commercial processors. CETIS is the first work to achieve this on commercial processors.

2.2 Intel Control-flow Enforcement Technology

The Intel Control-flow Enforcement Technology (CET) is a new hardware feature of Intel's 11th generation processors, and consists of two components: Indirect Branch Tracking (IBT) and Shadow Stacks (SHSTK). The IBT is used to protect the forward control flow of a program. When the program is indirectly called/jumped, the newly added state machine enters the WAIT_FOR_ENDBRANCH state. Only when the next executed instruction is the `ENDBR`, the state machine will enter the IDLE state, allowing the program to continue; otherwise, a control protection exception (#CP) will be triggered. As a result, coarse-grained CFI can be achieved by inserting the `ENDBR` instruction before all indirect call/jump targets. Based on IBT, one could achieve coarse-grained control flow integrity (CFI). Further, FineIBT [22] proposed a more fine-grained CFI by combining IBT with the default LLVM-CFI.

SHSTK is a hardware shadow stack, and the newly added shadow stack pointer (`%SSP`) register points to the top of the shadow stack. The `CALL` and `RET` instructions also push and pop the return address on the shadow stack. When executed, the `RET` instruction checks whether the return addresses on the top of the main stack and the shadow stack are the same. If not, a #CP exception will be raised.

To store the protected shadow stack, SHSTK introduces a new memory page type, the shadow stack page (i.e., shstk page). SHSTK uses an unused combination of bits in the PTE to identify the shstk pages — the R/W bit (set as 0) and the dirty bit (set as 1) [15]. That is, the shstk page is a read-only dirty page and the regular memory access instructions do not have write permission, but the CPU is able to store the return address on the page when executing the `CALL` instruction. Since the mismatch of return address may occur in certain legal scenarios, such as the C++ exception handling, the `WRSS` instruction is introduced to modify the return address on the shstk pages. The `WRSS` instruction has two variants to support different data sizes, i.e., `WRSSQ` (for 8 bytes) and `WRSSD` (for 4 bytes). The `WRSSQ/WRSSD` instruction writes the 8-byte/4-byte data from a 64-bit/32-bit general-purpose register to a destination address on the shstk page. Further, the destination address must be 8-byte aligned for `WRSSQ` and 4-byte aligned for `WRSSD`. If the alignment requirement is not met, a #GP exception will be raised.

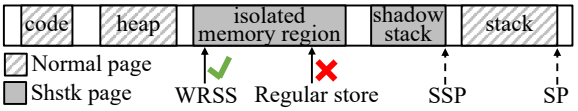


Fig. 1: The high-level idea of CETIS.

Mitigation against spectre attacks. IBT ensures the instructions at the target of an indirect `JMP/CALL` will only speculatively execute if there is an `ENDBR` instruction at the target, and SHSTK ensures the return stack buffer cannot be misused to speculatively return to a location when the return address is mismatched. Swivel [23] leverages CET to harden WebAssembly against Spectre attacks on the branch target buffer (BTB) and the return stack buffer (RSB).

3 OVERVIEW

In this section, we outline the high-level idea of building a memory isolation technique using SHSTK and the empirical analysis of SHSTK's architectural and micro-architectural features for such use cases.

3.1 CETIS: CET-based Isolation Technique

Because the shstk pages can only be written by `WRSS` instructions but not other memory access instructions, we opportunistically retrofit this property to develop a generic memory isolation technique. To expand further, CETIS sets the isolated memory region as the shstk pages, which can only be written by the trusted code using `WRSS` instructions. The integrity is effectively ensured by restricting the ability of the untrusted code to use `WRSS` instructions.

The high-level idea of CETIS is shown in Fig. 1. It allocates contiguous shstk pages as the sensitive memory region to store the sensitive memory objects. There is a guard page (i.e., a read-only page) before and after the isolated memory regions. CETIS does not preclude the normal use of SHSTK, which could allocate other shstk pages as the shadow stack to store return addresses and set the %SSP register to point to said pages. Other regular memory regions, such as the stack and the heap, are still set as the normal memory pages. The isolated memory region can only be written by `WRSS` instructions, and can be read arbitrarily as usual.

Since CETIS separates the memory access operations on the regular memory region and the isolated memory region, it is fundamentally different from existing address-based and domain-based isolation methods. The core idea of existing methods is to restrict the regular memory access instructions, for example, by confining the address range in address-based methods and controlling the access permission in domain-based isolation methods. Contrastingly,

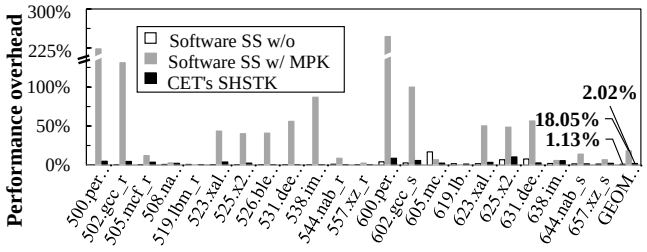


Fig. 2: Performance of software shadow stack mechanism and CET's SHSTK on SPEC CPU 2017 C/C++ benchmarks.

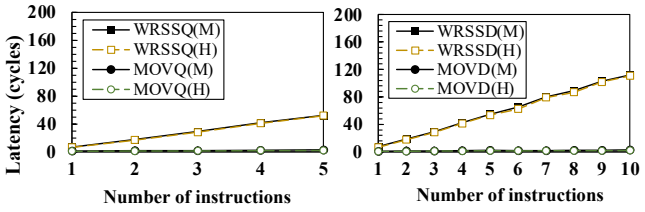


Fig. 3: The latency of writing data using `MOVQ`/`MOVD` and `WRSSQ`/`WRSSD` with different numbers of instructions (average of 1 million repetitions). M indicates cache miss, and H indicates cache hit. The target address of the write operation is cache-line aligned.

CETIS introduces the new `WRSS` instruction for accessing isolated memory regions, which restricts any code without accesses to the new instruction from accessing isolated domains.

3.2 Threat Model

We assume an adversary can exploit the vulnerabilities in the victim program to obtain arbitrary memory read/write primitives, but she is not yet able to alter its control flow (which is likely to be her goal, though). We further assume the system software, e.g., the operating system and the hypervisor, and the hardware are secure and trustworthy. The threat model of CETIS is the same as prior works [17, 27, 39]. While CETIS is designed to be generic, we specifically consider two common use cases.

Use case 1. The protected programs can be server programs, such as the Nginx web server, or user applications, such as browsers. Memory corruption defenses are deployed to prevent attackers from hijacking the control flow. CETIS can be used to protect the sensitive memory objects isolated by the defenses and/or the metadata of the defenses, such as the safe region in the CPI.

We assume the defense mechanism is properly implemented. Therefore, the defense mechanism ensures that the adversary cannot launch code-injection attacks or code-reuse attacks to execute unintended `WRSS` instructions; and CETIS prevents the adversary from tampering with the sensitive memory objects, which is the prerequisite of breaking the defense mechanism. As such, the defense mechanism and CETIS protect each other.

Use case 2. The protected software has integrated a just-in-time (JIT) compiler. The attack target of adversaries is to break the integrity of the code cache and inject shellcode therein, i.e., attackers use the arbitrary write primitive to modify the code cache directly. CETIS can be used to protect the code cache against such attacks. In such a scenario, CETIS shares the same threat model with other isolation works on JIT compilers, such as libmpk [27] and ERIM [37].

3.3 Understanding SHSTK and WRSS

Retrofitting CET's SHSTK for memory isolation seems a promising idea, but the potential impact on the performance of the applications using CETIS needed to be estimated first. Specifically, we aim to empirically analyze the following properties:

- *Performance impact due to SHSTK.* As CETIS needs to enable the SHSTK mechanism, an application using CETIS would have to use the SHSTK mechanism. Thus, we need to evaluate the degree of the slowdown to the application caused by SHSTK.
- *Latency of WRSS.* As CETIS uses WRSS instructions to store data into the isolated memory region, we need to measure the latency of WRSS to estimate the impact of frequent stores.
- *Comparison with MPK.* Though providing a different abstraction, CETIS is considerably similar to MPK in terms of use cases. We need to compare CETIS with the MPK-based scheme, which is generally regarded as the state-of-the-art method for isolation.
- *Other properties of SHSTK.* We need to understand whether SHSTK is restricted to only hold data, and whether there are constraints or performance penalties if the shstk pages can also hold and execute code.

As such, we conducted the empirical tests in consideration of the above aims. All the following experiments were performed on an Intel(R) Core (TM) i7-1165G7 CPU with Linux kernel v5.10.0.

3.3.1 The Performance Evaluation of SHSTK. We used SPEC CPU 2017 C/C++ benchmarks to evaluate the performance of the software-implemented shadow stack and the Intel CET's SHSTK based shadow stack. We used the compact register scheme [2] for the software shadow stack, which reserves the %R15 register to store the shadow stack's base address. We evaluated the performance of the software shadow stack with and without the protection of the MPK-based isolation method. Since Intel CET was already supported by LLVM, we used the Clang-7.0.1 compiler directly to enable the Intel CET's SHSTK mechanism when compiling the SPEC benchmarks.

As shown in Fig. 2, six test cases were missed, i.e., *omnetpp_r/s*, *leela_r/s*, *parest_r* and *povray_r*. This was due to #CP exception being triggered when the return addresses do not match on the main stack and shadow stack. The experiments showed that SHSTK (the geo_mean 2.02%) was highly efficient in protecting the backward edges of the control flow. Further, SHSTK was slightly slower than the software shadow stack without the isolation protection (the geo_mean 1.13%), but was much faster than the software shadow stack with the MPK-based protection (the geo_mean 18.05%).

> **Observation 1:** SHSTK is highly efficient and could be widely deployed. Compared with software-based solutions, CET's SHSTK is more efficient in isolating the shadow stacks.

3.3.2 Latency of `WRSS`. As a memory access instruction, `WRSS` was compared with the regular memory access instruction, `MOV`. `WRSS` supports memory writes with two different data sizes, i.e., `WRSSQ` with 8B and `WRSSD` with 4B. We compared `WRSSQ` with `MOVQ` and compared `WRSSD` with `MOVD`. We assessed the latency of `WRSS` and `MOV` under controlled cache misses or cache hits. Fig. 3(a) showed

```
1 //write the 1-byte 'a' at the 8-byte aligned address (dst)
2 asm volatile(
3   "mov $0x61,%%rcx \n\t" //%rcx contains the data 'a'
4   "mov (%0),%%rax \n\t"  //read 8 bytes data at the dst
5   "mov $0x0,%%al \n\t"   //clear the lowest 1-byte data
6   "or %%rax,%%rcx \n\t"  //combine the written data
7   "wrssq %%rcx,(%0) \n\t" //write the combined data
8   ::"r"(dst):"rcx","rax");
```

Listing 1: Example code for writing 1-byte data at 8-byte aligned address by using WRSSQ.

the latency of writing data using MOVQ and WRSSQ with different numbers of instructions, while Fig. 3(b) showed the latency of MOVD and WRSSD. The target address of each write operation was cache-line (64-byte) aligned in this experiment. We can see that the latency of a write operation was the same whether or not the write operation encountered a cache hit or a cache miss, which could be attributed to the store buffer (see further analysis in Appendix §A.0.1). And the latencies of WRSSQ and WRSSD were the same (both took 9.3 cycles), which was slower than MOV (0.8 cycles).

> **Observation 2**: The latency of WRSS takes about 9.3 CPU cycles which is slower than the MOV instruction. The latencies of two variants, i.e., WRSSQ and WRSSD, are the same.

3.3.3 Comparison with MPK. To estimate the performance of CETIS with respect to MPK-based isolation schemes, we compared WRSS with isolated memory updates using MPK. We defined PKMOV as the following sequence: enabling write permission (WRPKRU)→writing data (MOV)→disabling write permission (WRPKRU). We constructed an experimented to compare the write latency with WRSS and PKMOV with different data lengths. The destination addresses were cache-line aligned to meet the alignment requirement of WRSS.

The experimental results are shown in Fig. 4. We can see that WRSSQ performed better than WRSSD due to the support of longer data writes. And WRSSQ performed better than PKMOVQ when the data size was less than or equal to 24 bytes. As the data size increased to over 24 bytes, PKMOVQ performed better than WRSSQ.

We can also see that the latency of writing 1 byte to 7 bytes of data by using WRSSQ was slightly higher than that of writing 8 bytes, and the latency of writing 9 bytes to 15 bytes was slightly higher than 16 bytes. This could be attributed to the need for an additional data combination operation: WRSSQ can only write data with a fixed length (i.e., 8 bytes) and the destination address has to be 8-byte aligned. Data combination is illustrated in Listing 1, which provides an example of writing one-byte data at the 8-byte aligned address *dst*. It needs to read the data at the aligned address first (line 4), then combine the data (line 5 and line 6), and write the combined data at the aligned address using WRSSQ (line 7). If the address is not 8-byte aligned, the data combination operation will be more complicated.

> **Observation 3**: A data combination operation is needed when using WRSS to write data with arbitrary length at an arbitrary address. WRSSQ performs better than WRSSD and is better than PKMOV when the data writing is within 24 bytes.

3.3.4 Other properties of shstk pages. As mentioned in §2.2, the shstk pages are identified by using an unused combination of the

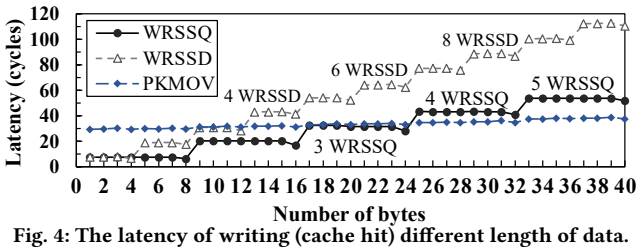


Fig. 4: The latency of writing (cache hit) different length of data.

R/W bit and the dirty bit in PTE. Hence, we found that other bits in PTE could be orthogonal to the bits set of the shstk pages. Such finding could be used to achieve other interesting mechanisms. For example, since the NX bit can be cleared in PTE, the shstk pages can be executable; and since the protection key bits can be enabled, the shstk pages can be also protected by the MPK. §3.3.2 has already evaluated the write operation (i.e., WRSS), we also evaluated the read and execution operations. We used mov instruction to sequentially read 80KiB of data in 20 consecutive 4KiB normal pages and shstk pages, respectively, to evaluate the impact of shstk pages on read operations. To evaluate the impact of shstk pages on execution operations, we set 20 consecutive 4KiB code pages which held consecutive simple `inc` instructions (with the register operand), and measured the latency of the instructions after these pages were configured as normal pages and shstk pages, respectively. We found that (1) the read and execution operations had no additional restrictions (e.g., no alignment requirements) compared to the operations on the normal memory pages; and (2) the latencies of read and execution operations on the shstk pages were the same as the operations on the normal memory pages.

> **Observation 4**: The shstk pages can be executable to protect the code, and there is no difference between the read/execution operation on the shstk pages and the normal pages.

3.4 Challenges of Utilizing SHSTK in CETIS

Based on the above observations, we find that SHSTK is efficient and CETIS could be performed better than MPK-based scheme in the situation wherein a small amount of data is written at a time. Further, CETIS can be used to protect more generic data, such as code. However, several challenges render the efficient use of WRSS difficult for programmers:

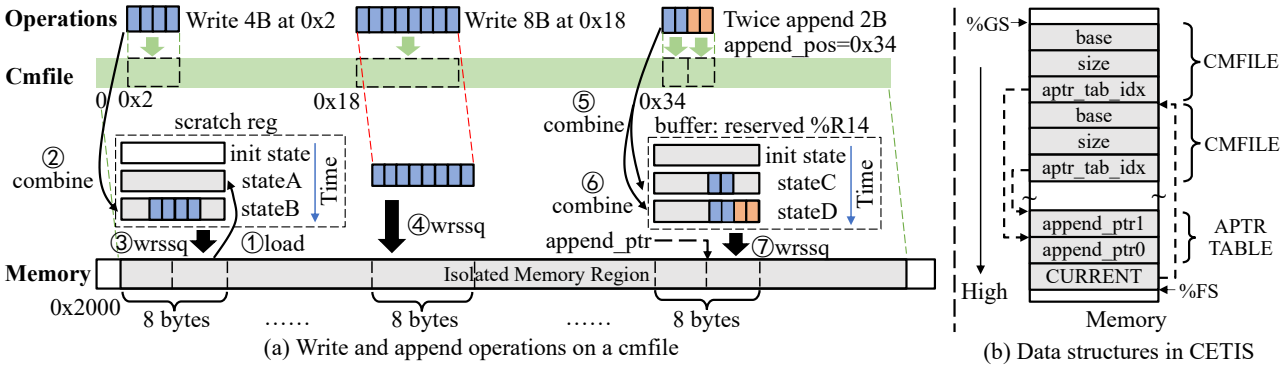
Challenge-1. A data combination operation is needed when using WRSS instructions to write data with arbitrary length. Choosing the optimal write strategy with WRSS instructions based on the situation of the address alignment and the data length is challenging.

Challenge-2. The WRSS instruction is not as efficient as the MOV instruction. To complete a given data write operation, as few WRSS instructions as possible need to be used. Therefore, it is important to buffer small data writes. Accordingly, determining how to maintain and secure this buffer is challenging.

In the next section, we will introduce the CETIS framework and show how CETIS addresses the aforementioned challenges.

4 THE CETIS FRAMEWORK

To address the challenges listed in §3.4, CETIS provides a software framework (as shown in Fig. 6) for developers to adopt CETIS



(a) Write and append operations on a cmfile (b) Data structures in CETIS

Fig. 5: CETIS memory file abstraction and its data structures.

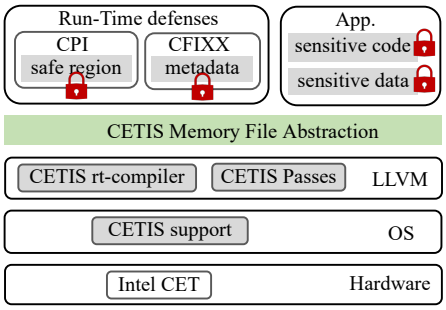


Fig. 6: CETIS framework.

easily. Specifically, CETIS provides a memory file abstraction for the isolated memory regions. One isolated memory region is abstracted as a CETIS memory file structure. Accessing to these regions is performed as read/write to the files using a set of APIs offered by CETIS. As a software framework, CETIS provides a static library (i.e., CETIS rt-compiler in Fig. 6) and a modified compiler tool-chain.

4.1 The CETIS Memory File Abstraction

As shown in Fig. 5, an isolated memory region is abstracted as a new memory file structure, referred to as *cmfile* (CETIS memory file). A similar file abstraction is selected due to the buffer design in CETIS being similar to the file buffer in the GNU C Library. The details of isolated memory regions and complicated `WRSS` usages are completely hidden from users. Users can operate cmfiles directly to ensure the memory objects' integrity. CETIS is responsible for translating the operations on a cmfile to the operations on the corresponding isolated memory region. Each cmfile position is continuously mapped to the corresponding isolated memory location. And they differ by a fixed offset which is 4KiB aligned.

The cmfile supports two different access modes — *the read/write mode* and *the append mode*.

Read/Write mode. The read/write mode allows users to read and write data with arbitrary length at an arbitrary position of a cmfile. Since read/write operations are directly translated into the accesses of the isolated memory region, no buffering is necessary. In this mode, a read operation is directly translated into a memory load with `mov`, and a write operation is translated into a store with `WRSSQ`, which is optionally proceeded by a load with `mov` and a data combination operation. For example, as shown in Fig. 5 (a), writing 4 bytes of data at position 0x2 will be translated into one memory load with `mov` (①), one data combination operation (②) and one write operation with `WRSSQ` (③).

Since data combination operations are highly dependent on address alignment, if the alignment can be determined statically, the operations can be optimized. For the operation in Fig. 5 (a), 8 bytes of data are being written at the position 0x18, it only needs one `WRSSQ` instruction (④). To support automated alignment inference, CETIS leverages the *known-bits analysis* in LLVM, which performs backward analysis to prove that each individual bit of a value is either zero or one, and to determine if the lowest 3 bits of the write position are known or not. If they are known, CETIS will then use this alignment information to optimize this write operation. CETIS also supports user-provided alignment hints. If users provide the alignment hint for a write position, CETIS will use this information to optimize all relevant write operations. Note that false alignment inference will not cause the crash of the program, since it can be handled lazily in CETIS by capturing the #GP exception.

Append mode. The append mode is designed to support the writing of small-sized data at consecutive positions of a cmfile. In this mode, CETIS introduces a write-combine buffer, which integrates the idea of *caching* and *buffering*, and only flushes the buffer to the isolated memory region when needed. CETIS reserves a 64-bit general purpose register `%R14`, which is used as a buffer that caches up to 8 bytes of data at an 8-byte aligned position in the cmfile. Users can specify the starting position of the append operations on the cmfile. A memory pointer, named *append_ptr*, is introduced to store the translated memory location from this specified position. Each append operation will append data at the append_ptr and then increase the append_ptr by the size of this appended data. The buffer always caches 8 bytes of data starting at the aligned address *append_ptr&~0x7*. Once the append_ptr reaches the buffer's boundary, the buffer will be synchronized, i.e., flushing its content into the corresponding memory chunk and then loading the next 8 bytes of data from the memory. In Fig. 5 (a), the two consecutive append operations each appending 2-byte data from position 0x34 will be translated into two data combination operations into the buffer (⑤ and ⑥). Since the append_ptr reaches the buffer's boundary, the buffer's content will be flushed into the isolated memory region using `WRSSQ` (⑦). If the append_ptr exceeds the buffer's boundary due to changing the starting append position by users, the buffer will be synchronized, i.e., flushing and loading at the new aligned

address. Users can also flush/load the buffer at any time using the CETIS-provided synchronization operation.

Supporting multiple cmfiles. Each isolated memory region corresponds to one cmfile. CETIS supports multiple cmfiles for each process. When operating on multiple cmfiles, however, the append mode introduces new challenges: the write-combined buffer, `%R14`, must be shared by all cmfiles; CETIS needs to switch buffers when performing the append operations on different cmfiles, for which the concept of the *current cmfile* is introduced. Users can only perform the append operations on the current cmfile, and thus need to explicitly switch the current cmfile to perform append operations on other cmfiles. CETIS flushes the contents of the buffer before the switch and loads it from another cmfile after the switch.

The data structures in CETIS. As shown in Fig. 5 (b), to describe a cmfile, CETIS introduces a structure called `CMFILE` (similar to the `FILE` in the GNU C Library) which contains three fields: *base* is the base address of the corresponding isolated memory region; *size* is the size of the isolated memory region; *aptr_tab_idx* is the index of this cmfile's append_ptr in the `APTR_TABLE`, which contains append_ptrs of all cmfiles. To indicate the current cmfile, CETIS introduces a memory pointer called `CURRENT` which points to the current CMFILE. To support multi-threading, the APTR_TABLE and the CURRENT are stored in a dedicated page of the thread-local storage area (at the location `%FS`-PAGE_SIZE). Since the isolated regions may be shared between threads, all CMFILEs are stored at a dedicated page which is pointed to by the `%GS` register.

The Integrity of CETIS's data structures. All data structures in CETIS are stored on the shstk pages, including append_ptr. Updates of these data structures need to use `WRSSQ`. Since the append_ptr needs to be updated in each append operation, using `WRSSQ` for each append operation will negate the performance gain of buffering. To address such problem, CETIS reserves the `%R15` register to store the value of the append_ptr. During the append operation, CETIS only needs to update the %R15 register. When switching cmfiles, this register needs to be synchronized, i.e., storing its value into the append_ptr of the switched-out cmfile in the APTR_TABLE and loading the value from the append_ptr of the switched-in cmfile.

4.2 Maintaining the Append-mode Buffers

As mentioned in §4.1, CETIS needs to reserve two general purpose registers `%R14` and `%R15` for the buffer and the append_ptr in the append mode. We next discuss how to prevent these registers from being used by irrelevant code (§4.2.1) and how to ensure the consistency of the buffered values in these registers(§4.2.2).

4.2.1 Avoiding Buffer Corruption. Intuitively, if the two registers are reserved in the whole program, corruption of the registers can be avoided using compiler-assisted approaches. However, this requires CETIS to re-compile all shared libraries, which can be impractical. Therefore, CETIS chooses to reserve the two registers selectively instead of in the whole program—as long as there is a function in a code file that uses append-mode buffers, all functions in the same folder where this code file is located are assumed to reserve registers. This may be coarse-grained, but it already achieves a good balance between the overhead of the frequent synchronization operations and the overhead of reserving registers on too many functions.

For the convenience of discussion, we denote a function that uses the append-mode buffers as a *rsvdFunc* and denote other functions as the *nmlFunc*s. CETIS identifies all rsvdFuncs and nmlFuncs during compilation, and forces synchronization of the buffers (including both `%R14` and `%R15`) (1) when a rsvdFunc invokes nmlFuncs and (2) when a nmlFunc invokes rsvdFuncs. However, precisely identifying all transitions between rsvdFuncs and nmlFuncs is challenging. To ensure completeness, we adopt a conservative strategy, which may cause unnecessary register synchronizations but no missed ones: (1) In a rsvdFunc, the synchronization will be inserted at the call-site of a direct/indirect call to a nmlFunc, since the control flow may be transferred from a rsvdFunc to a nmlFunc at these call-sites; (2) The synchronization will be inserted at the control flow transition from nmlFunc to rsvdFunc, that is, the entrance of all rsvdFuncs that are called directly or maybe indirectly by nmlFuncs. Only a rsvdFunc is a global function or its address has been taken, it will be recognized as a possible indirect call target by nmlFuncs. If it is also the direct call target of rsvdFuncs, this rsvdFunc will be cloned into two variants: one still has the synchronization; the other does not, which is only used for the direct call by rsvdFuncs.

4.2.2 Preserving Buffer Consistency. The buffer is synchronized either implicitly by CETIS or explicitly by the users. For ease of discussion, we use *append_ptr(reg)* to represent `%R15` and use *append_ptr(mem)* to represent the append_ptr stored in the APTR_TABLE. The synchronization operations contain the flush/load operations of the buffer and the store/load operations of the append_ptr(reg).

Implicit synchronizations. Three situations may trigger implicit synchronizations of the buffer and the append_ptr:

- **Buffer sync. caused by append_ptr changes.** When the append_ptr(reg) exceeds (due to users' setting) or reaches (due to increasing in append operations) the buffer's boundary, CETIS will flush the buffer and load the 8 bytes of data at the aligned address of the current append_ptr.
- **Buffer/append_ptr sync. caused by cmfile switches.** When switching cmfiles, CETIS needs to synchronize the buffer and the append_ptr. Before switching, CETIS flushes the buffer and stores the append_ptr(reg) to the append_ptr(mem); after switching, CETIS loads the corresponding memory chunk of the switched-in cmfile into the buffer, and loads the append_ptr(reg) from the append_ptr(mem) of the switched-in cmfile.
- **Buffer/append_ptr sync. caused by irrelevant code invokes.** When the control flow is transferred between rsvdFuncs and nmlFuncs, CETIS needs to synchronize the buffer and the append_ptr(reg). If the control flow is transferred from rsvdFuncs to nmlFuncs, CETIS flushes the buffer and stores the append_ptr(reg) to the append_ptr(mem); in contrast, CETIS loads the buffer and append_ptr(reg).

Explicit synchronizations. There are two situations where the users need to perform the buffer synchronizations explicitly:

- Users must synchronize the buffer explicitly during mode switch. When switching CETIS from the read/write mode to the append mode, users need to flush the buffer; when switching CETIS from the append mode to the read/write mode, users need to reload

Table 1: CETIS APIs

Cat.	APIs	Description
1	int cetis_init (void)	CETIS initialization.
	CMFILE *cmf_open (size_t len, bool is_exec)	Allocate a cmfile.
	int cmf_close (CMFILE *cmfp)	Deallocate a cmfile.
2	FPI make_pos_ind (CMFILE *cmfp, off_t off)	Construct a position indicator.
	assume_pos_aligned (FPI fpi, size_t align)	Alignment hint for compiler.
	int cmf_write (FPI fpi, void *src, size_t len)	Write data without buffer.
	int cmf_read (FPI fpi, void *dst, size_t len)	Read data without buffer.
3	int cmf_append_byte (uchar val) int cmf_append_word (ushort val) int cmf_append_dword (uint val) int cmf_append_qword (ulong val)	Append val w/ buffer at the append position of the current cmfile.
	int set_curr_append_pos (off_t off)	Set the current append position.
	int cmf_sync_buf (bool is_flush)	If the argument is true, flush the buffer to make its content global visible; otherwise, reload the buffer.
	int set_curr_cmf (CMFILE *cmfp)	Switch cmfile.
	CMFILE *get_curr_cmf (void)	Obtain the current cmfile pointer.

the buffer. We do not anticipate frequent switches between the two modes in the practical use of CETIS.

- Users must flush the buffer explicitly in multi-thread programs. If a thread uses append operations on a cmfile, we do not ensure memory consistency when other threads use read/write/append operations on the same cmfile. Users need to use the flush operation in conjunction with locks to ensure memory consistency. Since both the append_ptr(reg) and the append_ptr(mem) are thread-private, there is no need to synchronize them.

4.3 The CETIS API

Based on the cmfile abstraction of CETIS, we design a set of APIs (as shown in Table 1) to support the *read/write mode* and the *append mode*, rather than using different sets of APIs to support different modes. Such design draws on the APIs of the *FILE*, such as open()/close(), fseek() and fflush(), and encapsulates the internal implementation as much as possible, thereby providing the convenience of use for users. We demonstrate the use of the APIs in Listing 2. The APIs are divided into three categories.

The first category consists of the APIs for initialization. Before using CETIS, users should invoke the `cetis_init()` (line 7 in Listing 2) to initialize the CETIS framework. CETIS will first check whether the current system supports Intel CET, then allocate the data structures used in CETIS on the shstk pages, and finally register a signal handler to intercept #GP. Users may allocate a cmfile by invoking `cetis_open()` (lines 8-9). CETIS will then allocate contiguous shstk pages to the isolated memory region and initialize the CMFILE and the APTR_TABLE structures. Users can also de-allocate a cmfile by using `cetis_close()` (line 20).

The second category consists of the APIs for the read/write mode. Users can read (via `cmf_read()`) and write (via `cmf_write()`) data with arbitrary length at the arbitrary position through a file position indicator `FPI` (lines 12 and 19), which is constructed by invoking `make_pos_ind()`. `FPI` contains the translated memory location from the cmfile position (lines 1-5). As mentioned in §4.1, to further improve efficiency, users can use `assume_pos_aligned()` (line 11) to provide the alignment hint of the `FPI` for the compiler. The compiler uses the address alignment of this `FPI` to optimize the internal code of the following `cmf_write()`.

The third category consists of the APIs for the append mode. Four APIs (i.e., `cmf_append_{byte|word|dword|qword}()`) are provided to append fixed-sized data (i.e., 1/2/4/8 bytes) to the append position of the current cmfile (lines 15 and 17). To set the current cmfile, users can use `set_curr_cmf()` (line 13) and the buffer will be switched to this cmfile accordingly. The default append position of a cmfile is at the beginning of this cmfile. Users can also invoke `set_curr_append_pos()` (line 14) to specify a new append position, and CETIS will synchronize the buffer (only when the new position exceeds the buffer's boundary) and switch it to the new position. CETIS is not responsible for ensuring the APIs' correctness if users read/write data at the same cmfile position by using `cmf_read()/cmf_write()` and the append APIs simultaneously. Users should invoke `cmf_sync_buf()` (line 18) explicitly to synchronize the buffer. The reason `cmf_sync_buf()` is not invoked between `cmf_write()` (line 12) and `cmf_append_byte()` (line 15) is that the APIs in lines 13-14 synchronize the buffer implicitly.

```
1  typedef struct file_pos_ind{ulong addr; CMFILE *cmfp;}FPI;
2  FPI make_pos_ind (CMFILE *cmfp, off_t off) {
3      FPI cmfile_pi; cmfile_pi.cmfp=cmfp;
4      cmfile_pi.addr=cmfp->base+off; return cmfile_pi;
5  }
6  int main() {
7    cetis_init();
8    CMFILE *fp1=cmf_open(0x1000,true);
9    CMFILE *fp2=cmf_open(0x2000,false);
10   FPI cmfile_pi=make_pos_ind(fp1,0); /* construct a FPI */
11   assume_pos_aligned(cmfile_pi,8); /* alignment hint */
12   cmf_write(cmfile_pi,src,8); /* write 8B w/o buffer */
13   set_curr_cmf(fp1);/* switch current cmfile to fp1 */
14   set_curr_append_pos(0x20); /* set current append pos.*/
15   cmf_append_byte(value1); /* append 1 byte to fp1 */
16   set_curr_cmf(fp2);/* switch current cmfile to fp2 */
17   cmf_append_word(value2);/* append 2 bytes to fp2 */
18   cmf_sync_buf(true); /* flush the buffer's content */
19   cmf_read(cmfile_pi,dst,8); /* read 8B w/o buffer*/
20   cmf_close(fp1); cmf_close(fp2); return 0;
21 }
```

Listing 2: A code snippet for using CETIS APIs.

As mentioned in §4.1, global variables such as CMFILEs and APTR_TABLE are isolated in the shstk pages. Local variables may be corrupted in concurrent scenarios, for example, one thread may invoke `cmfile_write()`, and another thread may leverage the arbitrary write primitive to corrupt the local variables of the API, thereby changing the semantics of the API and indirectly breaking the isolation. To overcome the problem, local variables inside the APIs are promoted to registers, instead of being spilled to memory. Arguments passing between APIs such as position indicators may also be corrupted, which is discussed in §8.

5 CASE STUDIES

5.1 Code-pointer Integrity (CPI)

CPI protects the programs' code pointers, data pointers that may be used to reference the code pointers, and the return address, all of which are called *sensitive pointers*. As shown in Fig. 7 (a), the safe region of CPI includes *safe pointer store* and *safe stacks*. In safe pointer store, each sensitive pointer owns a data structure, which contains the value, upper and lower bound of the object pointed to by the sensitive pointer. The size of each structure is 24 bytes, and its address is 8-byte aligned. Safe stacks are used to protect the return address. The integrity of the safe region needs to be ensured.

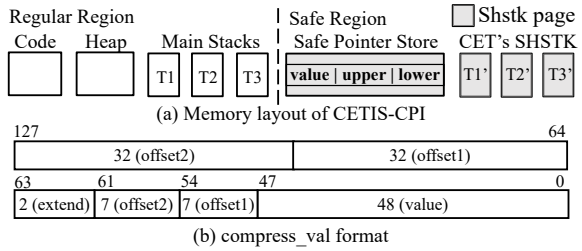


Fig. 7: Losslessly compress CPI's metadata.

When using CETIS to protect CPI's safe region, the safe stacks are no longer needed as SHSTK is enabled to protect the return address. We use `cmf_open()` to allocate a cmfile to hold the safe pointer store and use `cmf_read()`/`cmf_write()` to access each sensitive pointer's data structure in the safe pointer store. Each update to this 24-byte data structure requires executing the `WRSSQ` instruction 3 times. To improve the performance, we further compress the data structure. {*Value*, *upper*, *lower*} are all within the range of the object pointed to by the pointer, that is, using the *value* as the base address, the *upper* and *lower* can be obtained by adding or subtracting the offsets, thus, the metadata can be compressed losslessly. When reading metadata, the compressed value can be restored to the original metadata according to the compression rules.

The X86_64 processors can index 2^{48} bytes of address space. Only the lower 48 bits of the userspace pointer are valid, and the upper 16 bits are all 0. As shown in Fig. 7 (b), we compress the metadata of each sensitive pointer into 16 bytes, called *compress_val*. The lower 48 bits in compress_val are *value* of the sensitive pointers. The remaining bits are mainly used to store *offset1* and *offset2*, among which *offset1* is equal to *value* minus *lower*, and *offset2* is equal to *upper* minus *value*. Bits 62 and 63 are extend bits, which are used to identify the type of the sensitive pointers corresponding to the metadata: when extend=1, the pointer is a code pointer; when extend=2, the pointer is a data pointer that points to a "small object", whose size is less than 128 bytes, and offset1 and offset2 can be encoded within 7 bits; when extend=3, the pointer is a data pointer that points to a "large object", whose size is larger than or equal to 128 bytes; and when extend=0, the pointer has been freed.

Different types of pointers have different metadata compression strategies, as shown in Fig. 8, in which the data on the gray background represent the content that needs to be written to the cmfile. Our goal is to minimize the amount of data written to the cmfile each time. We can see that the code pointers only need to store the value to compress_val (i.e., 8 bytes); the data pointers that point to the "small object" store the value and two offsets (i.e., 8 bytes); the 48 to 61 bits of the compress_value of the data pointers pointing to the "large object" are not enough to store the offsets, and thus, an extra 8 bytes are needed. When the pointers are free, the lower 8 bytes are set to 0. Under the strategy of compressing metadata, only the data pointers pointing to the "large object" need to store 16 bytes, and in other cases, only 8 bytes need to be stored, which minimizes the number of `WRSSQ` instructions for better performance.

5.2 CFIXX

Dynamic dispatch is the core feature of polymorphism in C++, which allows the child class to rewrite virtual functions inherited

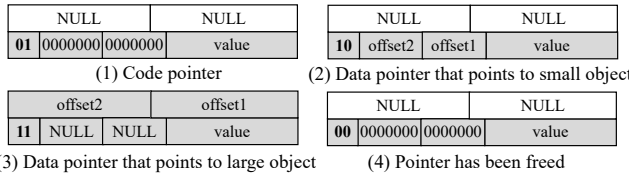


Fig. 8: Compress_val of different types of sensitive pointers.

from the parent class. Polymorphism is implemented through virtual tables, with each table containing the pointers of all virtual functions in a class. To protect the integrity of virtual tables, they are all stored in the `.rodata` section with read-only permission. However, the virtual table pointers are still stored in the writable memory which can be tampered with to launch VTable hijacking attacks [43]. To prevent such attacks, more fine-grained CFIs [24, 25, 36] have been proposed to check the validity of each virtual function invoking; other works have been proposed to ensure the integrity of the virtual table pointers, such as CFIXX [1] and VTrust [44].

CFIXX [1] stores a backup of the virtual table pointers in a metadata table which is isolated by using the address-based isolation method. CFIXX uses the virtual table pointer stored in the metadata table to index the target function pointers to complete the invoking of the virtual function. In detail, CFIXX modifies the compiler to intercept the creation of the virtual table pointers in two places — the constructor of the class and the initialization code of each RTTI object, and then additionally stores the pointers into the metadata table. CFIXX also modifies the compiler so that the virtual table pointer is obtained from the protected metadata table every time.

CETIS replaced the address-based isolation used in CFIXX to ensure the integrity of the metadata table. We use `cmf_open()` to allocate a cmfile to hold the metadata table. Storing the virtual table pointers into the metadata table is replaced to use `cmf_write()`. Since the virtual table pointer is 8 bytes and its address is 8-byte aligned, each write operation to the cmfile only needs one `WRSSQ` instruction. The virtual table pointers are obtained from the metadata table by using `cmf_read()`. Although CFIXX aims to protect the forward edges of control flows, the protection of the backward edges is still needed (also mentioned in its paper). When using CETIS, the return addresses are protected by SHSTK by default.

5.3 JIT Compiler

To improve efficiency, the JavaScript engine introduces the Just-In-Time (JIT) compiler. For the bytecodes that are frequently interpreted and executed, such as loop bodies and hot functions, the JIT compiler compiles them into semantically and functionally equivalent JITed code which is stored into a *code cache*. When these bytecodes are executed again later, the JIT engine directly jumps to the corresponding code cache for execution, without interpreting it. To allow the JIT compiler to write the JITed code into the code cache, both write and execute permissions are required, which violates the $W \oplus X$ policy. Attackers can launch code injection attacks by tampering with the code cache.

Some JIT engines such as ChakraCore protect the code cache based on the mprotect() system call. Fig. 9(a) shows the JIT compilation procedure of ChakraCore. When emitting the code cache, a JIT engine compiles the Intermediate representation (IR) one by one, and the generated JITed code fragments will be stored

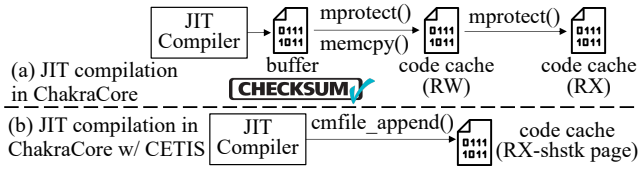


Fig. 9: The workflow of JIT compilation in ChakraCore.

in an encodeBuffer. After the emitting process, the encodeBuffer will be written into the code cache through memcpy(). mprotect() enables the write permission before memcpy() and disables it afterwards. Notably, mprotect() only protects the code cache from being tampered with, but the encodeBuffer that temporarily stores the binary code fragments is also readable and writable. Attackers may indirectly tamper with the code cache by manipulating the encodeBuffer. To resist attacks against the encodeBuffer, ChakraCore adds a checksum[35] for the encodeBuffer. When the JIT engine compiles each IR into machine code for storage in the encodeBuffer, it calculates the checksum of the machine code. After the code is copied to the code cache, the JIT engine will calculate the checksum again, and an exception will be raised if it does not match the previous checksum. Some works, such as libmpk [27], use the MPK-based isolation method to protect the code cache, while retaining the checksum mechanism to protect the encodeBuffer.

When using CETIS to protect the code cache, we modify ChakraCore's JIT compiler to invoke `cmf_open()` for allocating an executable cmfile to hold the code cache. Since the code is emitted almost continuously, we mainly use the append APIs, which have a built-in security buffer, to store the compiled binary code directly into the cmfile without temporary storage in the encodeBuffer. Fig. 9(b) shows the JIT compilation in ChakraCore with CETIS. The temporary encodeBuffer is no longer needed and therefore the attack vector is eliminated. Based on the method mentioned in §4.2.1, CETIS identifies that all functions in the `ChakraCore/lib/Backend` folder need to reserve the %R14 and %R15 registers.

6 IMPLEMENTATION

Enabling CET for the whole system. CET is enabled when the 23rd bit (CET bit) of the `%CR4` register is set. The MSR registers IA32_U_CET_MSR and IA32_S_CET_MSR configure CET for the user mode and kernel mode, respectively. Taking IA32_U_CET_MSR as an example, in the user mode, bit 0 controls whether the SHSTK is enabled, bit 1 controls whether the `WRSS` instruction can be used, and bit 2 controls whether the IBT is enabled.

Enabling CET's SHSTK for a process. A process' SHSTK capability is marked in its ELF header and can be verified from the readelf output. To build a SHSTK-enabled application, Binutils v2.31, GLIBC v2.28, GCC v8.1 or LLVM v10.0.1 or later are required. The process can use the compiler option *-fcf-protection* to enable CET, where *full* and *none* are used to enable and disable the CET support of the process, *return* and *branch* respectively indicates that the SHSTK and IBT are enabled. GCC enables CET by default. If the programs need any shared libraries, the loader checks all dependencies and enables CET when all requirements are met. For backward compatibility, GLIBC provides a few CET tunables via GLIBC_TUNABLES environment variable, such as glibc.tune.x86_shstk=on, which enables SHSTK for programs that need legacy shared libraries.

Allocating the shadow stack pages. Programs can allocate shstk pages by using mmap() with VM_SHADOW_STACK flag, and the permission cannot include PROT_WRITE. In the CET patch, when a process enables the CET's SHSTK through the compilation option, only the SHSTK is enabled by default, while the `WRSS` instruction is not supported. CETIS modified the CET patch (3 LoC) to support the `WRSS` when the SHSTK is enabled. Additionally, the CET patch does not support the kernel to enable the CET. CETIS will be extended to kernel space when the patch is available.

7 EVALUATION

We implemented CETIS on Ubuntu 20.10 (Kernel v5.10.0 with CET patch) that runs on a 2.80 GHz 11th Gen Intel(R) Core (TM) i7-1165G7 CPU with 8 cores and 32GB RAM. In this section, we evaluated the performance overhead of protecting CPI's safe region (§7.1) and the CFIXX's metadata table and the code cache in the JIT compiler of the ChakraCore (1.12.0.0-beta) (§7.2).

7.1 Protecting Memory-Corruption Defenses

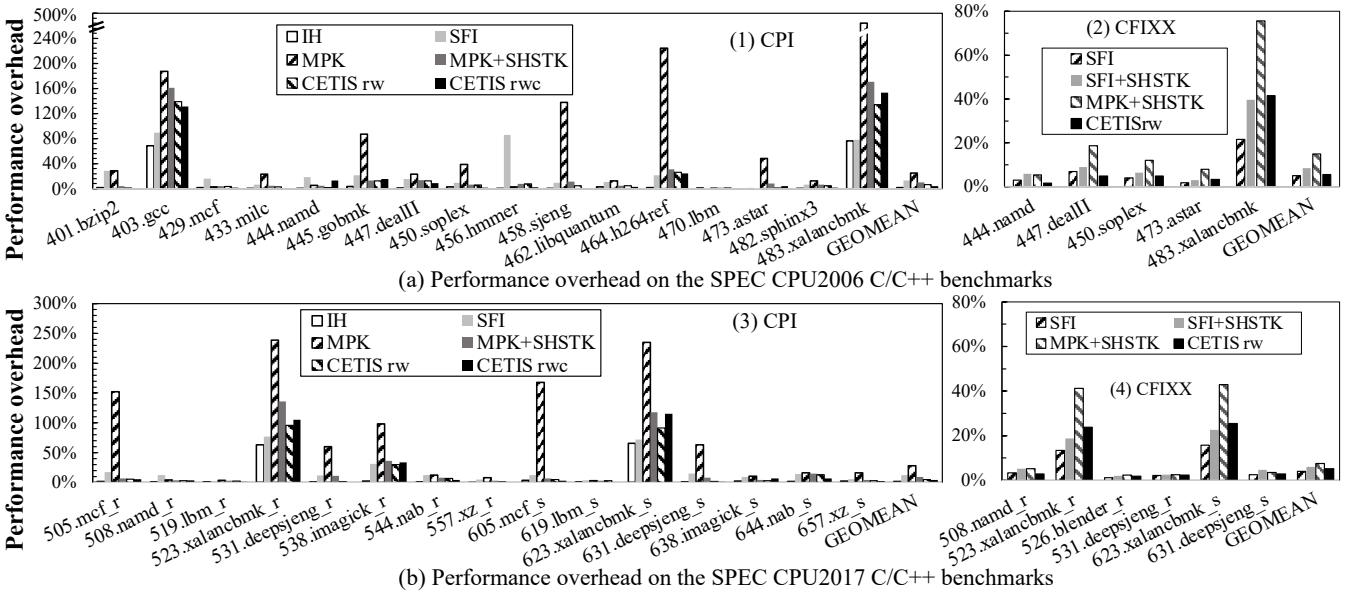
Defenses Configuration. To evaluate the practicality and performance of CETIS in protecting the isolated regions, we adopted two defenses, CPI [18] and CFIXX [1], and applied CETIS to protect their sensitive data, i.e., CPI's safe region and CFIXX's metadata table. For comparison, we implemented the SFI-based and MPK-based schemes for these defenses. Since MPX [15] is discarded in the new processors (with CET especially), we omitted MPX-based solutions in our study. We used the address sandboxing SFI scheme [38], and only memory writes were masked in the LLVM backend pass.

Macrobenchmarks. To evaluate the performance of different isolation techniques, we chose the CPU-intensive benchmarks, i.e., SPEC CPU2006 and SPEC CPU2017 C/C++ benchmarks. We compiled them at the O2 optimization level with the link-time optimization, and ran them with the *ref* dataset. For SPEC CPU2017, we used 2 copies for SPECrate benchmarks and 2 threads for SPECspeed benchmarks. We used two defenses, CPI and CFIXX, to protect each benchmark. For each combination of benchmark and defense, we conducted experiments for four cases: (1) protected only by the IH-based defense, (2) protected by the SFI-based isolation, (3) protected by the MPK-based isolation, and (4) protected by the CETIS-based isolation. The baseline does not enforce any protection.

Real-world application. To evaluate CETIS 's robustness and impact on real-world applications, we chose a popular web server, i.e., Nginx-1.14.2, as our protection target. Since Nginx is written in the C language and CFIXX is a defense mechanism for C++ programs, we only evaluate CPI on protecting Nginx. Similar to macrobenchmarks, we also conduct experiments with four protection cases.

7.1.1 *Macro-benchmarks Evaluation.*

CPI. Fig. 10 (1) and (3) show the performance overhead on the SPEC CPU2006 and CPU2017 incurred by CPI [18] when using IH/SFI/MPK/CETIS to protect its safe region (i.e., safe pointer store and safe stacks). Some benchmarks are missing, because the CPI failed to compile or run, or they raise #CP exception. They are *perlbench*, *sjeng*, and *omnetpp* in CPU2006; for CPU2017, besides the benchmarks listed in §3.3.1, *perlbench_r/s*, *gcc_r/s*, *x264_r/s*, and *blender_r* are missing. We found the following two situations may



(a) Performance overhead on the SPEC CPU2006 C/C++ benchmarks

(b) Performance overhead on the SPEC CPU2017 C/C++ benchmarks

Fig. 10: Performance overhead on the SPEC CPU2006/2017 C/C++ benchmarks incurred by CPI and CFIXX when using IH/SFI/MPK/CETIS to protect their sensitive data. All overheads are normalized to the unprotected benchmarks.

raise the #CP exceptions during function returns: (1) pushing return address on the stack without using the CALL instruction; (2) changing return address before executing the RET instruction. *IH*, *SFI* and *MPK* indicate that CPI's safe region (including safe stacks) is protected by information hiding, the SFI-based scheme and the MPK-based scheme; with *MPK+SHSTK*, the MPK only protects the safe pointer store, and the return addresses are protected by SHSTK instead of the safe stacks. It can be used to evaluate the performance improvement effectuated by the SHSTK compared with the safe stack; $CETIS_{rw}$ indicates the return addresses are protected by SHSTK, and the safe pointer store is protected by CETIS and updated by only using the read/write mode APIs; $CETIS_{rwc}$ indicates that use the lossless compression method in §5.1 compared with $CETIS_{rw}$. Note that the compression is only used in $CETIS_{rwc}$.

In summary, when using IH, SFI, MPK, MPK+SHSTK, $CETIS_{rw}$ and $CETIS_{rwc}$ to protect CPI, the geometric mean of the performance overhead on SPEC2006 is 1.93%, 12.94%, 24.88%, 10.30%, 6.22%, and 4.05%, respectively; and the geometric mean of the performance overhead on SPEC2017 is 1.85%, 12.26%, 27.88%, 9.23%, 4.77%, and 3.97%, respectively. We can see that, CETIS is more efficient than SFI-based and MPK-based schemes, and the CETIS with lossless compression incurs the lowest performance overhead.

To better compare CETIS with SFI-based/MPK-based schemes, we define OH_{scheme} as the overhead incurred by a defense with a specific isolation scheme. We also define Δ_{mpk} (=$OH_{mpk+shstk} - OH_{cetis_{rw}}$) as the relative overhead between MPK+SHSTK and $CETIS_{rw}$; Δ_{mpkC} (=$OH_{mpk+shstk} - OH_{cetis_{rwc}}$) as the relative overhead between MPK+SHSTK and $CETIS_{rwc}$; Δ_{sfi} is the relative overhead between SFI and $CETIS_{rw}$; Δ_{sfiC} is the relative overhead between SFI and $CETIS_{rwc}$.

Compared with MPK+SHSTK, $CETIS_{rw}$ is faster in all 31 cases, and the range of Δ_{mpk} is [0.15%, 40.66%]; $CETIS_{rwc}$ is faster in 28 cases. For these 28 cases, the range of Δ_{mpkC} is [0.02%, 30.71%]; for the remaining cases, the range of Δ_{mpkC} is [-8.17%, -2.79%].

Table 2: The statistics of write and read operations used on CPI.

Benchmarks	Write	Read
namd	3.7E+03 (0.19%)	1.9E+06 (99.81%)
gobmk	2.6E+08 (10.38%)	2.2E+09 (89.62%)
xalancbmk	3.9E+09 (7.73%)	4.7E+10 (92.27%)
namd_r	8.1E+05 (30.17%)	1.9E+06 (69.83%)
xalancbmk_r/s	5.7E+09 (8.65%)	6.1E+10 (91.35%)
imagick_r/s	3.7E+07 (0.23%)	1.6E+10 (99.77%)

Compared with SFI, $CETIS_{rw}$ is faster in 25 cases. For these 25 cases, the range of Δ_{sfi} is [0.20%, 78.70%]; for the remaining cases, the range of Δ_{sfi} is [-56.56%, -0.48%]. $CETIS_{rwc}$ is faster than SFI in 24 cases. For these 24 cases, the range of Δ_{sfiC} is [0.02%, 83.72%]; for the remaining cases, the range of Δ_{sfiC} is [-75.41%, -2.32%].

The impact of the compressed write on CPI. Although $CETIS_{rwc}$ performs better on average than $CETIS_{rw}$, we still found that the performance overheads of $CETIS_{rwc}$ in several cases, which are listed in Table 2, are higher than that of $CETIS_{rw}$. This is because that, when the sensitive pointers are de-referenced, decompressing the metadata after reading from the isolated region in the compression method incurs additional overhead. As shown in Table 2, the read operations on the isolated region account for the majority.

CFIXX. Fig. 10 (2) and (4) show the performance overheads on the SPEC CPU2006 and SPEC CPU2017 incurred by CFIXX [1] when using SFI/MPK/CETIS to protect its metadata table, which stores the virtual table pointers. Some benchmarks are missing because of raising #CP exception, they are *povray* and *omnetpp* in CPU2006, and *parest_r*, *povray_r*, *omnetpp_r/s*, *leela_r/s* in CPU2017. *SFI* represents that metadata table is protected by the SFI-based scheme; *SFI+SHSTK* indicates that the return addresses are additionally protected by CET's SHSTK; *MPK+SHSTK* uses the MPK-based scheme to protect the metadata table and CET's SHSTK to protect the return address; $CETIS_{rw}$ indicates that the return addresses are protected

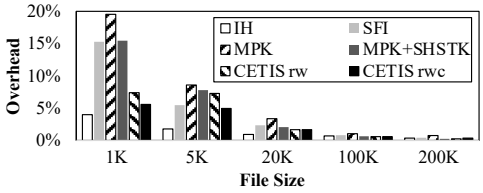


Fig. 11: Performance overhead on Nginx incurred by CPI when using IH/SFI/MPK/CETIS to protect its safe region.

by CET's SHSTK, and the metadata table is protected by CETIS and updated by only using the read/write mode APIs.

In summary, when using SFI, SFI+SHSTK, MPK+SHSTK, and CETIS$_{rw}$ to protect CFIXX, the geometric mean of the performance overhead on SPEC2006 is 5.08%, 8.38%, 14.92%, and 5.64%, respectively; and the geometric mean of the performance overhead on SPEC2017 is 4.06%, 6.08%, 7.61%, and 5.28%, respectively. CETIS is more efficient than SFI+SHSTK and MPK+SHSTK schemes, and is 1.22% slower than SFI. We define Δ_{mpk} (=$OH_{mpk+shstk} - OH_{cetis_{rw}}$) as the relative overhead between MPK+SHSTK and CETIS$_{rw}$. Compared with MPK+SHSTK, CETIS$_{rw}$ is faster in all 11 cases, and the range of Δ_{mpk} is [0.18%, 34.23%].

7.1.2 *Real-world Application Evaluation.*

Nginx. ApacheBench (ab) is used to simulate 10 concurrent clients sending 10,000 requests; each request asks the Nginx server to transfer a file remotely. We vary the size of the requested file, i.e., {1K, 5K, 20K, 100K, 200K}, to represent different configurations. Fig. 11 shows the performance overhead of Nginx under the protection of CPI with *IH*, *SFI*, *MPK*, *MPK+SHSTK*, *CETIS*$_{rw}$, and *CETIS*$_{rwc}$. The protection deployed in the six experiments are the same as SPEC in §7.1.1. The geometric means of performance overhead are 1.10%, 2.29%, 3.43%, 2.14%, 1.68% and 1.56%, respectively. As the file size increases, the overheads of all schemes decline. Further, CETIS performed better than the SFI-based and MPK-based schemes.

7.2 Protecting Code Cache of ChakraCore

To evaluate the practicality and performance of CETIS to protect sensitive code, we applied CETIS to protect ChakraCore's code cache. For comparison, we also implemented the MPK-based scheme (the same as ERIM [37] and libmpk [27]) for ChakraCore. We evaluated their performance overheads with the Octane benchmark [10], which is the JIT-heavy benchmark at runtime. Each JavaScript program in the benchmarks was executed 30 times, and we calculated the average score. Five cases, that is, *code-load*, *gbemu*, *mandreel*, *typescript* and *zlib*, can not execute normally because of #CP exception triggered by the CET.

Fig. 12 shows the performance overhead on the Octane benchmark. ChakraCore uses mprotect() syscall to ensure the W⊕X strategy on code cache, and the encodeBuffer of ChakraCore is protected by using the checksum; *mprotect+SHSTK* protects the return addresses by CET's SHSTK additionally, and the geometric mean performance overhead is 11.08%; *MPK+SHSTK* represents that using the MPK-based scheme to ensure the integrity of the code cache, the encodeBuffer is protected by the checksum, and the return addresses are protected by the SHSTK, and the geometric mean overhead is 9.84%; *CETIS*$_{rw}$ represents that the JITed code is directly written to the code cache by using the read/write mode APIs, thus

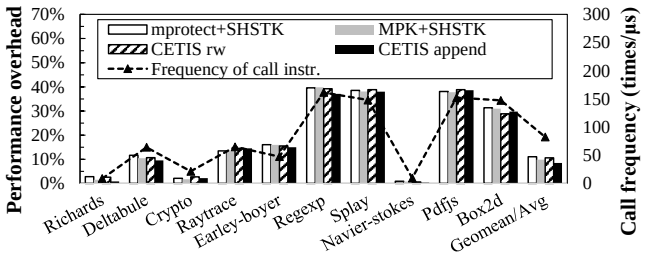


Fig. 12: Performance overhead on Octane benchmark when using mprotect/MPK/CETIS to protect ChakraCore's code cache. The baseline is mprotect() without SHSTK.

the encodeBuffer is no longer needed, and the return addresses are protected by the SHSTK. And the geometric mean overhead is 10.59%; *CETIS*$_{append}$ mainly uses the append APIs to write the JITed code, and the geometric mean overhead is 8.53%.

We can see that using the append mode of CETIS to protect code cache outperformed the mprotect-based and MPK-based schemes. Compared with MPK+SHSTK, CETIS$_{append}$ exhibited a performance improvement of up to 2.66% (Regexp), with the worst example being 1.14% slower than MPK+SHSTK (Raytrace). We can also see that CET's SHSTK incurred 11.08% performance overhead on ChakraCore, which was much higher than that on SPEC (as shown in Fig. 2). This was due to the `CALL`/`RET` instructions being executed considerably frequently in the JITed code. The two instructions were updated to access the shadow stack in CET, and the frequent execution would cause a higher performance overhead. To further verify it, we conducted an experiment to measure the execution frequency of the `CALL`/`RET` instructions in ChakraCore by using the Intel PT [15]. The frequency on average was 83/μs which was much higher than that in SPEC2006 (19/μs on average). Fig. 12 also shows the frequencies of each case, we can see that the higher the frequency, the higher the performance overhead.

8 DISCUSSION

Other attacks on JavaScript engines. As mentioned in §5.3, some works [27] only protected the code cache and the encodeBuffer, which is actually insufficient in terms of safety. Some works found that the source data of the code cache are also potential attack targets, such as the IR in the JIT compiler [7], the object tables in the JS Interpreter [28]. If the attacker modifies such data, the code cache will eventually be tampered with. NoJITSu [28] uses Intel MPK [15] to isolate sensitive memory objects such as bytecode, object tables, IR, and code cache. CETIS can be integrated with it, and stores the sensitive memory objects in shstk pages for isolating.

Comparison with MPK-based isolation technique. The MPK-based isolation ensures integrity and confidentiality of the isolated region, while CETIS cannot independently ensure confidentiality because the shstk pages do not restrict read operations. CETIS could be integrated with address-based isolation (such as SFI and Intel MPX[15]) to restrict the read operations. CETIS only supports 2 domains, while MPK supports up to 16 domains. How to expand CETIS to support more domains is left for future work.

Both CETIS and MPK-based isolation need to be deployed in conjunction with defenses, such as CFI and CPI, to ensure control-flow integrity. Without these defenses, neither CETIS nor MPK-based isolation can prevent the attacks in the PKU pitfall [5], such as constructing `WRPKRU` or `WRSS` instructions in executable pages.

Protecting OS kernel and enclave with CETIS. Since CET already supports kernel-space protection, CETIS can be used to protect sensitive memory objects (e.g., the page table) in the kernel. Further, the preview manual released by Intel has shown that the future CET will be supported in the SGX enclave [15]. Since the shstk page type is supported in the EPCM, CETIS can isolate code and data inside an enclave, while the MPK-based isolation cannot be used, since the protection keys are not trusted by the enclave.

Possible attacks against CETIS. Attackers may attack against CETIS in three ways:

- **Function-calls spraying.** Because the isolated regions start and end with a guard page, when attackers try to execute a large number of `CALL`s to adjust `%SSP`, SHSTK will crash when reaching the guard pages.
- **Arguments corrupting.** Attackers may tamper with the arguments, such as position indicators, when they are passed between APIs through non-control data attacks. MPK faces the same threat, i.e., the dynamically computed addresses within the isolated region could be tampered with when passed to the MPK-protected `MOV` instructions. However, a successful attack has not been demonstrated as of yet. A possible solution is to pass arguments through registers.
- **Unintended gadgets.** Attackers could also use other WRSS and INCSSP/RSTORSSP (for adjusting %SSP) to corrupt the isolated regions. This could be mitigated by inserting the bound-checks before these instructions, thereby ensuring the isolated regions cannot be accessed.

Thread safety of CETIS APIs. The data combination in write mode reads the data at the aligned address, combines the data, and writes to the aligned address. The write operation is not an atomic write and is not thread-safe. Additionally, if a thread uses append operations on a cmfile, CETIS does not ensure memory consistency when other threads operate on the same cmfile. Therefore, memory consistency needs to be ensured by users.

The impact of reserving registers. To support the write-combine buffer, two general-purpose registers were reserved in the append mode in CETIS. To evaluate the impact of reserving registers, we reserved `%R14` and `%R15` in the whole ChakraCore, and found that when testing the Octane [10], the performance slowdown is 0.89%. In the experiments, we reserved registers for the functions in a specific folder, and incurred 0.54% overhead.

9 CONCLUSION

Intra-process memory isolation is an important mechanism for preventing memory-corruption attacks. In this paper, we propose CETIS, a generic and efficient memory isolation technique based on Intel CET's SHSTK. In order to allow users to use CETIS efficiently and conveniently, CETIS is offered as a software framework, with a set of user-friendly APIs and a library. Experiments show that CETIS is more efficient than existing in-process isolation techniques, such as the MPK-based isolation schemes.

ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 61902374 and U1736208.

REFERENCES

[1] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. Cfixx: Object type integrity for c++ virtual dispatch. In *Symposium on Network and Distributed System Security (NDSS)*.
[2] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.
[3] Chapter 23.1 Introduction to virtual machine extensions. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual.
[4] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. 559–572.
[5] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1409–1426.
[6] Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*. 15–26.
[7] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2405–2419.
[8] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security*.
[9] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*.
[10] Google. 2017. The JavaScript Benchmark Suite for the modern web. http://chromium.github.io/octane/.
[11] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. 2019. IskiOS: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654* (2019).
[12] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, Vol. 1. IEEE, 13–15.
[13] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*.
[14] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.
[15] Intel. 2020. Intel 64 and IA-32 Architectures Software Developer's Manual.
[16] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1868–1882.
[17] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys* (Belgrade, Serbia). 16 pages. https://doi.org/10.1145/3064176.3064217
[18] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *OSDI*.
[19] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *CCS* (Denver, Colorado, USA). ACM, 1607–1619. https://doi.org/10.1145/2810103.2813690
[20] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *NDSS*.
[21] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *RAID*.
[22] Joao Moreira. 2021. FineIBT. https://lssna2021.sched.com/event/ljR8?iframe=no.
[23] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX) Security 21)*.

[24] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 577–587.

[25] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 914–926.

[26] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security*.

[27] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC)*. 241–254.

[28] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking Down JavaScript Engines. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nojitsu-locking-down-javascript-engines/

[29] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

[30] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*. 563–577. https://doi.org/10.1109/SP40000.2020.00041

[31] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z Snow, and Michalis Polychronakis. 2017. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 366–381.

[32] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*. 552–561.

[33] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*. 1–17. https://doi.org/10.1109/SP.2016.9

[34] Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in web applications. *Acm Sigplan Notices* 41, 1 (2006), 372–382.

[35] Theori. 2016. Chakra JIT CFG Bypass. http://theori.io/research/chakra-jit-cfg-bypass.

[36] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 941–955.

[37] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security*.

[38] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216. https://doi.org/10.1145/173668.168635

[39] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 592–607.

[40] Z. Wang, C. Wu, Y. Zhang, B. Tang, P. Yew, M. Xie, Y. Lai, Y. Kang, Y. Cheng, and Z. Shi. 5555. Making Information Hiding Effective Again. *IEEE Transactions on Dependable and Secure Computing* 01 (mar 5555), 1–1. https://doi.org/10.1109/TDSC.2021.3064086

[41] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. 2019. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1239–1256.

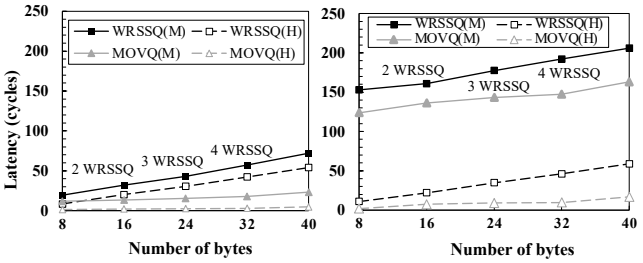
[42] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. 2021. Making Information Hiding Effective Again. *IEEE Transactions on Dependable and Secure Computing* (2021).

[43] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity.. In *NDSS*.

[44] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *NDSS*.

A OTHER EVALUATIONS ON WRSS

A.0.1 The analysis of WRSS's latency. To prove the latency affect of cache hit and cache miss comes from the store buffer, we designed an additional experiment to measure the latency of the write operation that indeed writes into the cache (i.e., globally visible). In the experiment, we inserted the `MFENCE` instruction after each write operation (i.e., `MOVQ` and `WRSSQ`). `MFENCE` ensures the write operation is globally visible [15]. Fig. 13(a) shows the experimental results. We can see that the latency difference is less than 20 CPU cycles, which is much lower than our common sense that the cache miss takes about >100 CPU cycles. We guessed `MFENCE` has some internal optimizations that guarantees all preceding memory accesses are global visible without waiting for actually write to the cache. To prove this, we performed an extra experiment that replaces `MFENCE` with the serialization instruction `CPUID`. `CPUID` forces the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed [15]. The experimental results are given in Fig. 13 (b) and shown that the latency difference between the cache miss and the cache hit is about 150 CPU cycles. We can also see that the latency impact of `MOVQ` and `WRSSQ` is the same regardless of the cache hit or not. We can conclude that compared to `MOV`, the additional latency of `WRSS` comes from the hardware component of `WRSS` not the cache system and they all use the store buffer to avoid blocking the pipeline.



(a) Latency that followed by MFENCE (b) Latency that followed by CPUID

Fig. 13: The latency of writing data that is a multiple of 8 bytes by the `MOVD` and `WRSSQ` followed by `MFENCE` or `CPUID` instruction (average of 1 million repetitions). M indicates cache miss, and H indicates cache hit. The target address of the write operation is cache-line aligned.

A.0.2 The Impact of `WRSS` on the CPU Pipeline. The latency of `PKMOV` and `WRSS` is the key to the efficiency of the isolation schemes. Furthermore, the impact on the CPU pipeline is as important as the latency. The `WRPKRU` instruction used in the MPK-based isolation scheme will not be executed speculatively, and it is only a memory access serialization instruction, it does not affect non-memory access instructions such as `INC` [15]. Although `WRSS` is not a memory access serialization instruction, it is a memory access instruction which is constrained by the X86 weaker Total Store Order (TSO) memory consistency model. In the TSO model, among the four possible Read and Write orders, i.e. Read→Write, Read→Read, Write→Write, and Write→Read, the processor will not guarantee the order of Write→Read if they access different memory locations. It allows Read to be execute before the Write if they access different memory locations. Hence, the load instruction that following `WRSS`, which access different location, can be executed ahead. In summary, `WRSS` is not a memory access serialization instruction, but it follows the X86 weaker TSO model that the subsequent load instruction can be executed ahead if they access different locations.