SecPaging: Secure Enclave Paging with Hardware-Enforced Protection against Controlled-Channel Attacks

Yunkai Bai, Peinan Li*, Yubiao Huang, Shiwen Wang, Xingbin Wang, Dan Meng and Rui Hou
Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
School of Cyber Security, University of Chinese Academy of Sciences

ABSTRACT

As a prevalent privacy-preserving technology, Trusted Execution Environment has become widely adopted in numerous commercial processors. Nonetheless, they remain susceptible to various controlled-channel attacks. Untrusted operating systems can deduce enclave secrets by manipulating page tables or observing allocation- or swap-based page faults. In this paper, we propose SecPaging, a novel secure enclave paging mechanism based on hardware-enforced and microcode-supported protection to prevent these attacks. First, enclave PTEs are protected through hardware isolation, preventing privileged attackers from malicious tampering or observations. Second, an Eager-Allocation mechanism is employed to prevent allocation-based controlled-channel attacks. Besides, a Record-Reload mechanism is proposed to prevent swap-based controlled-channel attacks. We simulate SecPaging on real SGX. Experiments demonstrate that controlled channel attacks can be defended with minimal performance overhead.

ACM Reference Format:
Yunkai Bai, Peinan Li*, Yubiao Huang, Shiwen Wang, Xingbin Wang, Dan Meng and Rui Hou . 2024. SecPaging: Secure Enclave Paging with Hardware-Enforced Protection against Controlled-Channel Attacks. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM,New York,NY,USA, 6 pages. https://doi.org/10.1145/3649329.3658241

1 INTRODUCTION

In confidential computing, trusted execution environment (TEE) stands out as a crucial innovation for privacy preservation, which serves many critical scenarios such as healthcare, finance, and digital rights management. State-of-the-art technologies like Intel SGX [1, 2] and AMD SEV [3] have been integrated into mainstream processors, establishing industry standards for adoption. In the design of TEEs, the essential strategy is to deploy sensitive codes and data within enclaves, isolated from untrusted software, thereby guaranteeing confidentiality and integrity.

Despite their advantages, current TEEs like SGX1/2 and SEV entrust the untrusted operating system or hypervisor with the enclave paging, encompassing both enclave page table and memory management. This design exposes a large attack surface for privileged attackers to compromise enclave page tables and observe secret-dependent page faults to deduce confidential data [4–7]. These attacks are commonly referred to as controlled-channel attacks. Compared to microarchitectural side-channel attacks, these attacks are much easier to orchestrate and yield higher accuracy due to minimal noise interference. Consequently, numerous attack variants have been reported in the literature [8–10]. The attack approaches are summarized into three categories:

Attack approach 1: Compromise enclave page table entry or observe enclave page state changes. Privileged attackers can proactively invalidate the secret-dependent enclave page table entries [4, 5], triggering page faults in the enclave. As page faults are handled by the untrusted operating system, attackers can obtain the access address of the enclave and infer the associated secret value. Alternatively, by observing the access or dirty bits in page table entries, attackers can infer which pages have been accessed or modified [6, 7], enabling them to deduce sensitive information.

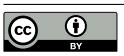
Attack approach 2: Observe page faults caused by on-demand stack page allocation. Dynamic memory allocation enhances enclave efficiency by permitting the augment of memory pages [2]. Stack pages are typically allocated on demand via page faults, which can be observed by an attacker. In cases where stack page faults may occur within a secret-dependent branch, attackers may deduce the secrets based on monitoring page faults or analyzing relevant access addresses.

Attack approach 3: Swap out enclave page to trigger page faults in subsequent execution. Privileged attackers can swap a secret-dependent enclave page to non-enclave memory. Access to the page in subsequent execution will incur page faults. Such events can be manipulated to reveal sensitive information [11, 12].

To defend against these attacks, several countermeasures have been proposed. Since the enclave is created by an application, the enclave and the application share the same virtual address space and public page tables. A straightforward approach is to enable enclaves to independently manage their private page tables [11, 13], thereby eliminating reliance on the untrusted operating system. However, this autonomy introduces a new attack surface for malicious enclaves to potentially compromise the untrusted operating system or other software [12]. An alternative collaborative management strategy between enclaves and the operating system [12] mitigates this particular risk but still leaves opportunities for privileged attackers to manipulate enclave page tables. Another hardware-based solution [14] is designed to defend against attack approach 1 (based on page table management), but it falls short of thwarting the other two attack approaches (based on memory management).

In this paper, we propose SecPaging, a novel secure enclave paging mechanism based on hardware-enforced and microcode-supported protection. SecPaging is designed to defend against the controlled-channel attack based on both page tables and memory

*Corresponding author is Peinan Li, lipeinan@iie.ac.cn.



DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3658241

management. SecPaging allows the operating system to manage enclave memory securely and efficiently. This approach avoids the risk that malicious enclaves compromise the operating system. The major contributions are as follows.

- **Hardware-enforced isolation for enclave page table entries (PTEs) protection.** SecPaging separates enclave PTEs from the public page table and stores them in the hardware-protected enclave memory. Modification of the enclave PTEs is exclusively permitted through microcodes. This approach allows the OS to manage the enclave PTEs via microcodes while protecting them from tampering or unauthorized observation.
- **Eager-allocation for dynamic stack page allocation.** Based on static analysis of stack utilization, SecPaging instruments the enclave code to preallocate additional pages when the utilization reaches a specific threshold. This preallocation mitigates page faults caused by on-demand stack allocation.
- **Record-reload for enclave page swap.** SecPaging ensures the reloading of all evicted pages of an enclave before switching to the enclave context. The pages used by the running enclave cannot be evicted, preventing attackers from constructing page faults by evicting enclave pages.
- **Evaluate SecPaging on SGX processors.** We simulate SecPaging on both SGX1 and SGX2 processors. Experiment results demonstrate that the average performance overhead of SecPaging on SGX1 and SGX2 is 2.42% and 1.54%, respectively.

2 BACKGROUND

2.1 Enclave Paging in Intel SGX

Intel SGX (Software Guard Extensions) is a set of microcode of processors designed to protect the confidentiality and integrity of sensitive code and data. SGX processors allocate a physically contiguous memory region known as PRM (Processor Reserved Memory). The PRM holds the Enclave Page Cache (EPC) which stores enclave code and data. Importantly, EPC is invisible to system software or other enclaves. In SGX, each enclave has a control structure (SECS) to save its metadata.

EPC Page Management. EPC pages are allocated to and reclaimed from enclave by system software through SGX microcodes. The system software also updates the enclave page with a virtual-to-physical mapping. In this procedure, the SGX microcodes record the metadata of each EPC page, such as owner enclave, access permission, and page type, in the EPCM (EPC Metadata). The EPCM is utilized by MMU to conduct security checks, ensuring that enclave memory is isolated from system software and enclaves.

Enclave Dynamic Memory Management. In SGX1, the maximum capacity for PRM is 128MB, and enclave memory is allocated at creation and has fixed access permission. SGX2 increases the maximum PRM capacity to 1TB and enables dynamic augments to enclave page numbers and permission during execution. First, the OS initiates operation on enclave page through microcodes EACCEPT/EMODPR/EMODT. Subsequently, the enclave confirms the modifications through microcodes EACCEPT/EACCEPTCOPY.

EPC Page Swap. Since PRM capacity is limited, SGX supports swapping pages between EPC and normal memory, allowing for EPC oversubscription. When EPC runs out, OS writes the ciphertext of an EPC page to normal memory via EWB microcodes. Correspondingly, OS reloads the encrypted page from normal memory to EPC page via ELD/ELDU microcode. During this procedure, the metadata is verified to ensure the confidentiality and integrity of the encrypted EPC page.

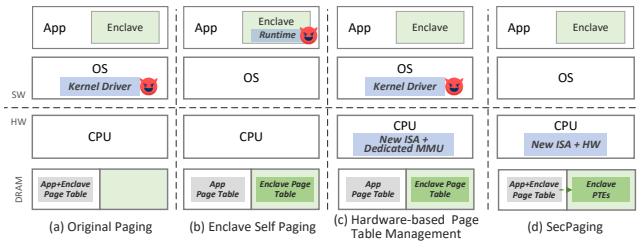


Figure 1: Existing enclave paging methods.

2.2 Threat Model

Our threat model aligns with typical controlled-channel attacks [4], wherein a privileged attacker assumes complete control over the operating system. Specifically, this attacker is capable of: ① accessing enclave Page Table Entries (PTEs); ② managing page faults initiated by the enclave and observing the relevant addresses; ③ allocating and reclaiming physical enclave memory pages, and swapping them with non-enclave memory pages.

We do not consider microarchitecture side-channel and off-chip DRAM bus probing attacks. Their countermeasures are orthogonal to our work and they can be integrated together for better security.

3 MOTIVATION AND DESIGN GOAL

In classic TEE designs, as illustrated in Figure 1(a), enclave paging is managed by the kernel driver within the untrusted operating system (OS). This setup allows privileged attackers to manipulate enclave PTEs in the public page table and extract sensitive information through the aforementioned three attack approaches. To avoid reliance on OS, some studies propose to employ a dedicated private enclave page table managed by the runtime inside the enclave, as shown in Figure 1(b). However, this design can be compromised by malicious enclaves, which may modify their pages to illegally access OS memory. In a different approach, Autarky [12] suggests a collaborative approach for secure paging between the enclave and the OS, but this does not fully prevent malicious OS activities, such as inducing page faults by tampering with the enclave page table. Another solution, Iso-X [14], shown in Figure 1(c), introduces dedicated enclave page tables managed via a new Instruction Set Architecture (ISA) and a dedicated Memory Management Unit (MMU). Nevertheless, Iso-X not only introduces significant timing and area overhead but also faces vulnerabilities related to dynamic stack allocation and page swapping.

Therefore, this paper aims to thwart compromises in enclave page tables by a malicious operating system while keeping its management flexibility. Additionally, it aims to prevent any malicious attack from enclaves themselves. Moreover, potential risks associated with memory management should be addressed, encompassing both allocation and swapping aspects.

4 DESIGN OVERVIEW

To meet the design goal, we propose SecPaging, as illustrated in Figure 1(d). SecPaging enables the OS to manage enclave memory and page tables with microcode support securely. Moreover, it employs hardware-enforced isolation to safeguard enclave PTEs. Figure 2 provides an overview of SecPaging, including the following key features:

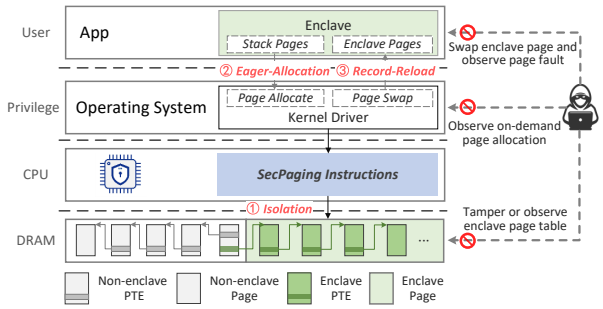


Figure 2: Design overview of SecPaging.

① SecPaging separates the enclave PTEs from the public page table by partitioning the virtual address space between the enclave and the application. Subsequently, the enclave PTEs are stored in enclave memory, which is invisible to both the operating system and the enclave itself, ensuring protection against tampering and observation. Additionally, SecPaging allows the operating system to employ trusted microcodes to manage enclave PTEs.

② To avoid page faults caused by on-demand stack allocation, SecPaging inserts eager-allocation code snippets to enclave code based on static analysis of stack utilization. The enclave proactively requests the OS to allocate additional stack pages upon reaching a threshold. Despite the OS monitoring the address of preallocation, this address doesn't correspond to the one accessed by the enclave. Consequently, the actual access pattern of the enclave remains undisclosed.

③ To prevent page faults caused by page swaps, SecPaging proposes a record-reload mechanism. When enclave pages are evicted, they are recorded in a lightweight bitmap. Once the enclave is rescheduled, these pages are reloaded. During enclave execution, SecPaging ensures that its pages can not be evicted through microcode.

5 IMPLEMENTATION

5.1 Protect Enclave PTE by Hardware Isolation

SecPaging leverages hardware isolation to thwart tampering and observation of enclave PTEs. To separate the enclave PTEs from the public page table, the virtual address space of the enclave is specified in the configuration file provided by SGX SDK during compilation. Subsequently, the enclave PTEs can be securely stored within the isolated EPC memory, as depicted in Figure 3. SecPaging provides the OS with microcodes to manage the enclave PTEs for compatibility with the OS's page table management. As enclave and non-enclave still use the same level-0 page table, the enclave can access the non-enclave memory securely and efficiently. However, it faces three challenges: 1) how to prevent OS from maliciously exploiting the page table management. 2) how to prevent the malicious enclave itself from compromising the PTEs. 3) how to protect the level-0 page table in non-enclave memory. To address these challenges, SecPaging employs the following mechanism.

Perform sanity checks on arguments of microcodes to prevent malicious exploiting. SecPaging adds microcode to manage enclave PTEs, as listed in Table 1. The OS employs these instructions to manage the enclave PTEs as normal. The OS creates or removes virtual-to-physical mappings through EMAP or EUNMAP. To avoid execution failure of EMAP resulting from absent

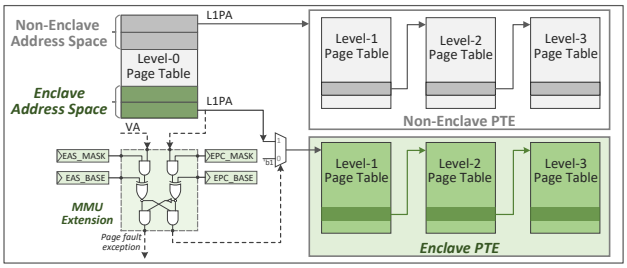


Figure 3: Protect enclave PTEs by isolation and MMU.

Table 1: Semantics of instrucitons in SecPaging.

Instruciton	Caller	Semantics	Type
EADDPD	Priv.	Add new page directory or page table	New
EMAP	Priv.	Create virtual to physical mapping	New
EUNMAP	Priv.	Delete virtual to physical mapping	New
ECHANGE	Priv.	Modify permission to an EPC page	New
EACCEPT	User	Confirm changes to an EPC page	Modify
EWB	Priv.	Evict an EPC page to normal memory	Modify
ELD	Priv.	Reload an evicted page to an EPC page	Modify
EENTER	User	Check evicted pages and start enclave	Modify
ERESUME	User	Check evicted pages and resume enclave	Modify

page directories, we introduce the EADD to allocate page directories. Since the permissions of the enclave page may be modified in SGX2, ECHANGE is added to modify the permission bits of enclave PTE. To prevent malicious OS from injecting illegal arguments that might result in page faults, the arguments of microcode are verified according to the permission bits in the EPCM.

Mark the enclave PTE invisible to itself to prevent it from being compromised. Malicious enclaves may temper the PTEs in EPC memory to access any HostOS memory. To mitigate this risk, SecPaging isolates the enclave PTEs from the enclave itself. Specifically, we introduce a new EPC page type, PT_PTE, designed to mark the enclave PTE page as invisible to the enclave itself. The page type is checked during the EMAP/EUNMAP to prohibit malicious enclaves from mapping the enclave PTE page.

Protect level-0 page table from tempering via MMU extension. To allow OS to maintain the page table as normal, especially the non-enclave page table, the level-0 page table remains visible to OS. Therefore, it is essential to prevent OS from maliciously mapping the enclave virtual address space to non-enclave PTEs. SecPaging introduces an MMU extension to check whether the level-0 page table points to the level-1 page table in EPC. EAS_BASE and EAS_MASK registers are introduced to represent the enclave address space and verify whether a virtual address (VA) falls within the space. SGX maintains the EPC address range in EPC_BASE and EPC_MASK registers. We use them to verify whether the level-1 page table resides in EPC. If the level-1 page table of the enclave does not reside in EPC, MMU triggers a page fault exception.

5.2 Prevent Observable Stack-Based Page Fault by Eager-Allocation

SecPaging proposes eager-allocation for the enclave stack to eliminate the page fault caused by on-demand stack allocation. As shown in Figure 4, SecPaging instruments the enclave code to preallocate the stack pages. To minimize the frequency of instrumentation, we employ an automated tool to statically analyze enclave stack utilization. Consequently, instrumentation is selectively applied to functions whose utilization reaches a predefined threshold.

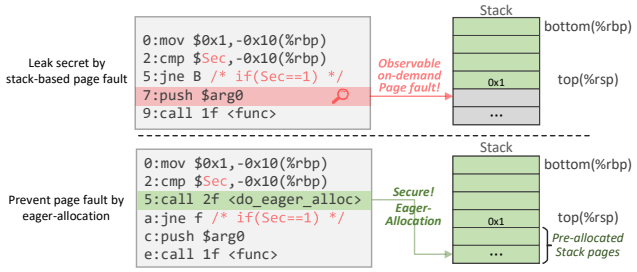


Figure 4: Prevent observing on-demand stack allocation by Eager-Allocation.

Static analysis of stack utilization to reduce performance overhead on enclave execution. Analyzing stack utilization dynamically through instrumentation before each function avoids page faults resulting from stack exhaustion, yet it introduces a notable performance overhead. A possible solution is to reduce the frequency of instrumentation by statically analyzing the stack size of enclave functions before compilation. This is attributed to that the majority of functions' stack sizes can be determined through static analysis. Therefore, we compute the stack utilization by accumulating the stack size for each function in the program's call graph, and identify the specific function where the stack utilization reaches a threshold.

Instrument stack eager-allocation code to minimize memory wastage. Based on the analysis, SecPaging instruments eager-allocation code before the function where the stack utilization reaches a threshold. To reduce memory wastage, only one page is preallocated for the enclave. Nevertheless, there are two cases where static analysis cannot determine their stack size. 1) Recursive functions whose recursion depth depends on the external arguments. To avoid page fault, SecPaging instruments stack utilization analysis code before the functions and preallocates additional pages when the stack utilization reaches a threshold. 2) Functions that call `alloca` API, and the argument of `alloca` depends on external input. To avoid potential page faults caused by the alloca function, we preallocate a certain amount of stack memory.

Prevent the malicious OS from creating illegal PTE of stack pages. Since the OS is untrusted, we must ensure that new pages are actually allocated and the corresponding page table entries are created correctly through trusted microcode. The existing EACCEPT instruction solely serves to confirm the correct allocation of pages by OS. Therefore, we extend the EACCEPT to validate the corresponding PTEs, ensuring the permissions in PTEs are consistent with those in EPCM.

5.3 Prevent Observable Swap-based Page Fault by Record-Reload

SecPaging permits the swap of pages only for non-executing enclaves through microcode. When enclave pages are swapped out, they are recorded in a lightweight bitmap. Once the enclave starts or resumes execution, SecPaging reloads all evicted pages. Therefore, enclave does not trigger page faults arising from page swap during enclave execution.

Restricted eviction of enclave pages. Since evicting a page used by a running enclave would immediately trigger a page fault within the enclave. It is essential to refine the EWB instruction to restrict the eviction of enclave pages. During enclave execution,

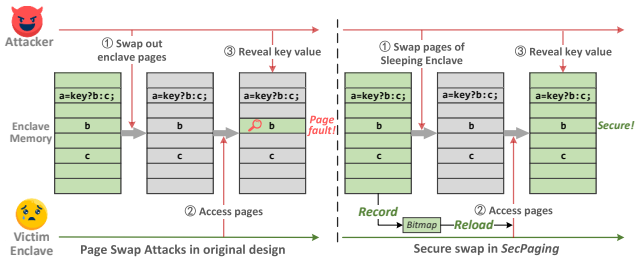


Figure 5: Defend against swap-based attacks through the Record-Reload.

its state in the SECS is *In use*. When the enclave exits execution, its state in the SECS is updated to *Not in use* by the microcode. EWB checks the state of the enclave, only pages from a sleeping enclave can be swapped, as such operations do not trigger page faults within the inactive enclave.

Lightweight recording of evicted enclave pages. Benefiting from the contiguous virtual address space of enclave, SecPaging records all evicted pages of an enclave in a lightweight bitmap. Each bit in the bitmap indicates whether the corresponding page is evicted or not. To protect the bitmap from tempering, it is stored in the EPC memory. Since OS evicts/reloads an EPC page to/from normal memory via the EWB/ELD instruction, we modified EWB/ELD to set/reset the corresponding bit in the bitmap according to the virtual address of the page.

Guaranteed reloading of evicted enclave pages. When the enclave starts execution, SecPaging checks the bitmap to ensure all of the evicted pages are reloaded into the EPC through microcode. This approach eliminates the page faults caused by accessing evicted pages. Since the EENTER/ERESUME instructions are utilized to start/resume enclave execution, the value of the bitmap is checked during the execution of the instructions to ensure that all bits have been reset. If there are evicted pages, the execution of the enclave is refused, effectively forcing OS to reload the evicted enclave pages before execution.

The record-reload results in minimal performance overhead. This is attributed to two key factors: Firstly, reloading pages when the enclave starts execution effectively mitigates page fault, context switches and TLB shutdowns. Secondly, page swap is a relatively low-probability event, given the large EPC size in SGX2. We evaluated the performance overhead in Section 6.2.

6 EVALUATION

6.1 Experiment Methodology

Our experiments are designed to evaluate both (1) the performance overhead introduced by SecPaging on the SGX1 and SGX2 enclave and (2) the overhead incurred by the hardware modifications and the complexity of implementing SecPaging.

Experimental setup. To comprehensively evaluate the overhead of SecPaging compared to native SGX, we establish experimental environments on SGX1 and SGX2 processors, as outlined in Table 2. We evaluate SecPaging on real SGX instead of simulation mode in SGX which does not emulate memory encryption

Table 2: Experiment environment setup.

CPU	System	EPC Size	SDK	Driver	LibOS
SGX1 (Intel Core i7-6700)	Linux 5.4	128MB	v2.21	v2.13	Gramine v1.5
SGX2 (Intel Xeon 6580)	Linux 5.19	256GB	v2.16	v2.6	Gramine v1.5

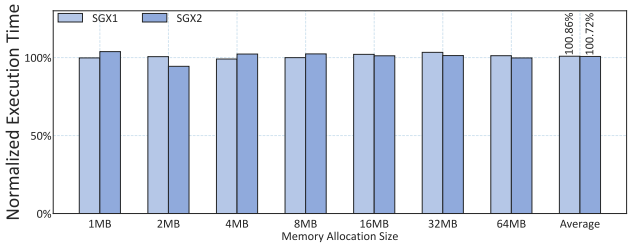


Figure 6: Performance overhead of enclave PTE isolation on memory allocation.

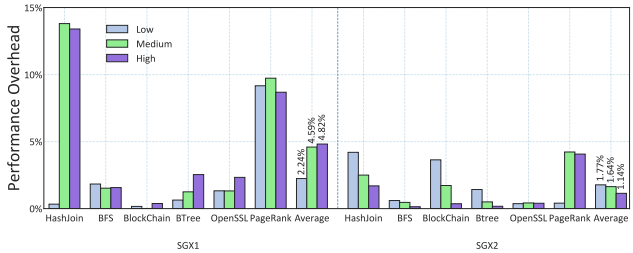


Figure 7: Performance overhead of record-reload.

engine and shows much better performance than real SGX. We implement the SecPaging instructions in the SGX kernel driver to simulate the performance overhead. SecPaging stores the enclave PTE and bitmap in EPC. Since the SGX kernel driver cannot direct access to the EPC in real SGX, we allocate a contiguous memory region within the kernel to hold these metadata. The MMU extension can operate parallel with the original permission checking logic, resulting in a negligible performance overhead.

Benchmarks. To evaluate the overhead of stack eager-allocation, we select SGX-NBench benchmark which has frequent function invocation. To evaluate the overhead of record-reload, we use 6 applications of different types including CPU-intensive (*e.g.*, Blockchain), Data-intensive (*e.g.*, HashJoin, Pagerank, BFS, OpenSSL, BTree). Each application is executed with 3 different sizes of datasets: Low (<EPC size), Medium (≈ EPC size), and High (>EPC size). To evaluate the overall performance overhead of SecPaging, we run the real-world application Lighttpd, a lightweight web server, in an enclave.

6.2 Performance Evaluation

Performance overhead of enclave PTE isolation. The isolation incurs overhead on page table management during memory allocation. Thus, for SGX2, which supports dynamic memory augment, we evaluate the overhead of using the `malloc` function to allocate memory of varying sizes from 1MB to 64MB. Since SGX1 enclaves allocate all enclave memory at creation, we evaluate the performance overhead of creating enclaves with sizes from 1MB to 64MB. Figure 6 demonstrates that the average overhead on SGX1 and SGX2 is 0.86% and 0.72%, respectively. Since memory allocation is just a small part of the enclave's execution, the overhead is even lower when considering the whole enclave execution.

Performance overhead of eager-allocation. We employ an automated tool to instrument stack preallocation code into SGX-NBench. As illustrated in Table 3, the average performance overhead for dynamic and static analysis is 16.30% and <1%, respectively. The overhead of dynamic analysis on Huffman, reaching 85.25%, is primarily attributed to the numerous short functions within Huffman and dynamic analysis of stack utilization instruments code before every function, contributing significantly to this overhead. Through static analysis, eager-allocation instruments preallocation code only when the stack is poised for potential exhaustion. The overhead of Huffman is reduced to only 0.07%.

Table 3: Eager-allocation performance on SGX-NBench.

	Perf.	NumSort	StrSort	Fourier	Assign	Idea	Huffman	NerualNet
Native	Results	1639.2	585.47	59917	67.993	15520	7356.4	137.67
Dyn. Ana.	Results	1524.1	522.31	58689	67.256	10273	3971	136.18
	Overhead	7.55%	12.09%	2.09%	1.10%	51.08%	85.25%	1.09%
Sta. Ana.	Results	1629.8	585.33	59868	67.905	15517	7351.5	137.65
	Overhead	0.58%	0.02%	0.08%	0.13%	0.02%	0.07%	0.01%

Performance overhead of record-reload. Record-reload introduces latency to page eviction as well as enclave execution. We comprehensively evaluate the record-reload on 6 applications with data sets of various sizes (Low, Medium, and High). As depicted in Figure 7, the average performance overhead on SGX1 for data sets of different sizes is 2.24% (Low), 4.59% (Medium), and 4.82% (High). Importantly, record-reload mitigates the time-consuming page faults, resulting in minimal overhead. As the dataset size increases, the performance overhead increases as the number of page swaps increases accordingly. The case HashJoin has the highest overhead because the number of page swaps ($\approx 10^7$) is orders of magnitude higher than the other cases ($<10^6$).

The average performance overhead on SGX2 for data sets of different sizes is 1.77% (Low), 1.64% (Medium), and 1.14% (High). The performance overhead of record-reload for SGX2 is significantly reduced compared to SGX1. This is attributed to the increase in SGX2 EPC size, which reduces the number of page swaps.

Overall performance overhead of SecPaging. We execute the Lighttpd v1.4.59 within the enclave, keeping the default settings for Lighttpd and operating requests within a single thread. Then we send 1 million requests to download a 20KB file from the Lighttpd server, using varying numbers of threads from 1 to 256. As depicted in Figure 8, the average performance overhead of implementing SecPaging in SGX1 and SGX2 for Lighttpd is only 2.42% and 1.54%, respectively.

6.3 Implementation complexity of SecPaging

EPC Memory Capacity Overhead. SecPaging stores the enclave PTE in EPC memory. Taking a typical 64-bit processor as an example, it requires 8 bytes to store the PTE for every 4KB page. Moreover, the page dictionaries require 8 bytes for every 4KB PTE page. Consequently, the total memory capacity overhead of the enclave

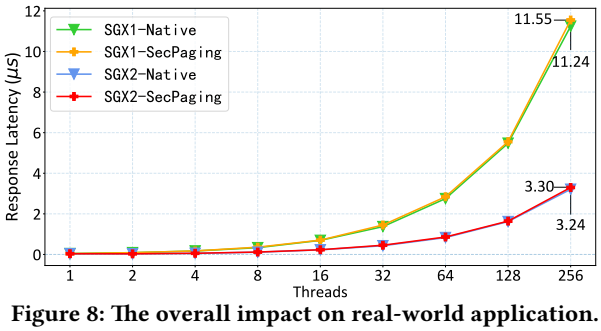


Figure 8: The overall impact on real-world application.

Table 4: Security analysis for related works. ● represents corresponding attacks can be defended. ◐ represents corresponding attacks can be partially mitigated. ○ represents that attacks still exist.

	Page Table Based Attack	Page Allocation Based Attack	Page Swap Based Attack	Malicious Enclaves Attack HostOS
T-SGX [15]	◐	○	○	●
Varys [17]	◐	○	○	●
Déjà Vu [16]	◐	○	○	●
SGXLAPD [18]	◐	◐	◐	●
Heisenberg [19]	◐	◐	○	●
Autarky [12]	◐	●	◐	●
Keystone [13]	●	●	○	○
InvisiPage [11]	●	●	○	○
SecPaging	●	●	●	●

PTEs is approximately 0.2% of EPC memory. Additionally, we utilize a bitmap in EPC memory to track evicted pages of enclave, requiring only 1 bit for every 4KB page. Consequently, the overhead introduced by bitmap is <0.01% of enclave memory size.

Hardware overhead. The SecPaging instructions can be implemented through a firmware update. SecPaging adds the MMU extension and two additional registers, EAS_BASE and EAS_MASK to verify whether a VA falls within the enclave virtual address space. EPC_BASE and EPC_MASK registers used to check whether the level-1 page table falls within the EPC or not are already present in the SGX. The MMU extension is a simple combination logic and incurs negligible area and timing overhead. Because there are only three logic gates in the path between its inputs and outputs.

7 COMPARISON WITH RELATED WORKS

We compare the security of SecPaging with other controlled channel defense mechanisms in Table 4. T-SGX [15], Déjà Vu [16], and Varys [17] employ software-based methods to detect malicious page faults during enclave execution. However, they cannot defend against attacks that involve snooping on the page table's Access/Dirty bits. The SGX-LAPD [18] aims to obfuscate page access via large pages but an attacker can still potentially infer information through access to large pages. The Heisenberg [19] suggests preloading all PTE into TLB in advance when enclave starts execution to prevent page faults. Nevertheless, it faces challenges as SGX2's memory capacity expands, making it difficult to store all PTEs in the TLB. Autarky [12] securely manages page tables and page faults through collaboration between the enclave and OS, but a malicious OS can still construct attacks based on page tables. Keystone [13] and InvisiPage [11] introduce the concept of a dedicated page table managed by the enclave itself, which allows a malicious enclave to access HostOS memory without restrictions.

This paper provides secure enclave paging with the following policy: ① SecPaging protects the enclave PTEs from being snooped and tampered with through hardware isolation. ② SecPaging prevents data leakage caused by page faults arising from stack allocation through the eager-allocation. ③ SecPaging prevents data leakage caused by page faults arising from page swap through the record-reload. ④ SecPaging enables OS to securely manage enclave memory through microcode, preventing the introduction of new attack surfaces caused by malicious enclaves.

8 DISCUSSION

Extend to protect Trusted Virtual Machines (TVM). TVM is also susceptible to controlled-channel attacks. In AMD SEV, the physical memory and nested page tables of a TVM is managed by untrusted hypervisor. An attacker who compromises the hypervisor can construct page faults within a TVM by tampering with the mapping in the nested page tables. SecPaging can be adapted to defend against controlled channel attacks on TVM with relatively simple modifications. We can introduce SecPaging microcode into SEV as well. Consequently, the nested page tables of TVM can be securely stored in the private memory of TVM, protected by hardware encryption and isolation mechanisms. Furthermore, the eager-allocation can be employed to protect the memory ballooning mechanism of TVM. Although the current SEV implementation lacks support for page swaps, and all SEV memory is currently pinned in memory, record-reload can be applied to SEV in future.

9 CONCLUSION

SecPaging proposes a secure enclave paging mechanism based on hardware-enforced and microcode-supported protection, preventing attackers from compromising enclave PTE. Furthermore, SecPaging prevents data leakage caused by page faults arising from stack allocation and page swap through eager-allocation and record-reload. We prototype SecPaging on SGX1 and SGX2 processors. SecPaging effectively defends against controlled channel attacks and introduces minimal performance overhead on enclave.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 62125208 and 62202467. The corresponding author is Peinan Li.

REFERENCES

[1] McKeen, Frank, et al. "Innovative instructions and software model for isolated execution." In HASP'13. 2013.
[2] McKeen, Frank, et al. "Intel® software guard extensions (Intel® sgx) support for dynamic memory management inside an enclave." In HASP'16. 2016, pp. 1-9.
[3] Kaplan, David, et al. "AMD memory encryption." White paper (2016): 13.
[4] Xu, Yuanzhong, et al. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems." In S&P'2015. IEEE, 2015, pp. 640-656.
[5] Shinde, Shweta, et al. "Preventing page faults from telling your secrets." In AsiaCCS'16. ACM, 2016, pp. 317-328.
[6] Van Bulck, Jo, et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution." In Security'17. 2017, pp. 1041-1056.
[7] Wang, Wenhao, et al. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX." In Security'17. 2017, pp. 2421-2434.
[8] Chen, Guoxing, et al. "SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution." In EuroS&P'19. IEEE, 2019, pp. 142-157.
[9] Van Bulck, Jo, et al. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution." In Security'18. 2018, pp. 991-1008.
[10] Li, Mengyuan, et al. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization." In Security'19. 2019, pp. 1257-1272.
[11] Aga, Shaizeen, et al. "InvisiPage: oblivious demand paging for secure enclaves." In ISCA'19. 2019, pp. 372-384.
[12] Orenbach, Meni, et al. "Autarky: Closing controlled channels with self-paging enclaves." In EuroSys'20. 2020, pp. 1-16.
[13] Lee, Dayeol, et al. "Keystone: An open framework for architecting trusted execution environments." In EuroSys. 2020, pp. 1-16.
[14] Evtyushkin, Dmitry, et al. "Iso-x: A flexible architecture for hardware-managed isolated execution." In MICRO'14. IEEE, 2014, pp. 190-202.
[15] Shih, Ming-Wei, et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In NDSS'17. 2017.
[16] Chen, Sanchuan, et al. "Detecting privileged side-channel attacks in shielded execution with Déjá Vu." In AsiaCCS'17. ACM, 2017, pp. 7-18.
[17] Oleksenko, Oleksii, et al. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks." In ATC'18. 2018, pp. 227-240.
[18] Fu, Yangchun, et al. "Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults." In RAID'17. Springer, 2017, pp.357-380.
[19] Strackx, Raoul, et al. "The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks." arXiv preprint arXiv:1712.08519 (2017).