Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks

Pan Zhang*[†]
Huazhong Univ. of Sci. & Tech., China
UC Riverside, USA
panzhanghust@gmail.com

Chengyu Song
UC Riverside, USA
csong@cs.ucr.edu

Heng Yin
UC Riverside, USA
heng@cs.ucr.edu

Deqing Zou[‡]
Huazhong Univ. of Sci. & Tech., China
deqingzou@hust.edu.cn

Elaine Shi
Cornell University, USA
elaine@cs.cornell.edu

Hai Jin*
Huazhong Univ. of Sci. & Tech., China
hjin@hust.edu.cn

Abstract

Intel *Software Guard eXtensions* (SGX) provides a hardware-based trusted execution environment for security-sensitive computations. A program running inside the trusted domain (an enclave) is protected against direct attacks from other software, including privileged software like the *operating system* (OS), the hypervisor, and low-level firmwares. However, recent research has shown that the SGX is vulnerable to a set of side-channel attacks that allow attackers to compromise the confidentiality of an enclave's execution, such as the controlled-channel attack. Unfortunately, existing defenses either provide an incomplete protection or impose too much performance overhead. In this work, we propose Klotski, an efficient obfuscated execution technique to defeat the controlled-channel attacks with a tunable trade-off between security and performance. From a high level, Klotski emulates a secure memory subsystem. It leverages an enhanced ORAM protocol to load code and data into two software caches with configurable size, which are re-randomized for after a configurable interval. More importantly, Klotski employs several optimizations to reduce the performance overhead caused by software-based address translation and software cache replacement. Evaluation results show that Klotski is secure against controlled-channel attacks and its performance overhead much lower than previous solutions.

CCS Concepts • **Security and privacy → Systems security**; **Side-channel analysis and countermeasures**; **Software and application security**.

Keywords Intel SGX; Oblivious Execution; Runtime Randomization; Page Fault Channel;

ACM Reference Format:
Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. 2020. Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378487

*Pan Zhang and Hai Jin are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

[†]The main work was done when Pan Zhang worked at University of California, Riverside as a Visiting Scholar under Prof. Heng Yin and Prof. Chengyu Song's supervision.

[‡]Deqing Zou is with Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, 518057, China.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378487

1 Introduction

A *trusted execution environment* (TEE) aims to safeguard the confidentiality and integrity of an application's execution against various security threats on potentially hostile platforms that are not physically controlled by the application developers (*e.g.*, public cloud). Such threats include malware, malicious or compromised *operating systems* (OS), rogue cloud administrators, etc. TEE can be provided by both software [16, 24, 32] and hardware [4, 21, 33]. Among these solutions, Intel *Software Guard eXtensions* (SGX) [33] is the most promising one for its availability in commodity Intel CPUs (since the Skylake microarchitecture) and strong, hardware-based security guarantees. Specifically, an application running inside an SGX-protected TEE (*a.k.a.* enclave) only needs to trust the processor, which is a much smaller *trusted computing base* (TCB) than software-based solutions. At the same time, Intel has also made significant efforts to formally verify the hardware specification of SGX and the implementation of its cryptography operations [26]. For these reasons, a variety of SGX-based applications have been developed, including data analytics [34, 40], machine learning [36], Tor [29], containers [6], and library OS to support legacy applications [7, 46, 52].

Unfortunately, SGX also has weaknesses. In particular, side-channel attacks are outside the threat model of its design and researchers have demonstrated the feasibility of several types of side-channel attacks against applications running inside enclaves, including page-fault-based attacks (*a.k.a.* controlled-channel attack) [45, 56], cache-based attacks [9, 19, 23, 42, 54], branch-prediction-based attack [30], and transient attacks [10, 14, 31, 41]. Among these attacks, we argue that controlled-channel attacks are the most critical because most other finer-grained attacks [10, 14, 23, 30, 31, 41, 53, 54] (except Meltdown [31]) are more costly. Thus, adversaries usually rely on controlled-channel attacks to pinpoint functions of interest and only launch fine-grained attacks when the target functions are executed.

Controlled-channel attacks are possible because (1) attackers can observe memory access patterns [51] and (2) applications' memory access patterns are input-dependent. Therefore, controlled-channel attacks can be defeated by addressing either one of the root causes. User-space page-fault detection methods [44, 45] aim to prevent attackers from acquiring page access patterns through deliberately injected page faults. Unfortunately, they are ineffective against controlled-channel attacks that do not rely on page-fault (*e.g.*, access-bit-based attack [54]). SGX-Shield [43] tries to obfuscate the memory access through memory layout randomization. But as it only randomizes once at load time, it can be defeated through online profiling [30]. Oblivious execution techniques [2, 3, 37, 39, 45] are secure against online profiling but impose much higher performance overhead. For example, deterministic multiplexing [45] imposes a runtime performance overhead of over 4000×, multiple program path execution [37] has an overhead of 9×, and OBFSCURO [2] has an overhead of 51× over *simple* benchmarks.

In this work, we aim to defeat controlled-channel attacks with a tunable trade-off between security guarantees and performance overhead. At a high level, our proposed system KLOTSKI acts as a memory subsystem. It consists of two software caches (one execution vCache and one data vCache), a software memory management unit (sMMU), and a virtual main memory. Similar to a physical CPU, all executed instructions are fetched from the execution vCache and all data is read from/write to the data vCache. sMMU translates compile-time virtual addresses, which we refer to as *logical addresses*, to runtime virtual addresses (*i.e.*, linear addresses to the hardware MMU). This mechanism allows us to load a block of memory into any slot of the vCache. To obfuscate the memory access pattern during the execution, sMMU uses the Ring ORAM protocol [38] to access the main memory. Moreover, because "client-side" ORAM operations, including accesses to the metadata (*e.g.*, stash and position map) are also vulnerable to side-channel attacks, KLOTSKI uses additional protections to make sure all such operations are oblivious under our threat model. Finally, vCaches are re-randomized through random replacement policy and forced flushing.

While the above design is secure and is similar to previous work [2, 3, 39], a straightforward implementation would impose a very high performance overhead. Another vital contribution of KLOTSKI is several optimization techniques. First, KLOTSKI reduces the number of address translations by caching the results. Similar to a hardware *translation lookaside buffer* (TLB), KLOTSKI leverages program locality to avoid redundant address translations. Second, KLOTSKI improves program locality to reduce the number of cache replacements, including aligning loops to avoid cross-cache-block loop bodies and relocating constants to the same code block. Finally, KLOTSKI provides a tunable trade-off between the performance and security through configurable parameters. On one end of the spectrum, KLOTSKI can guarantee oblivious execution when developers choose a small vCache size (*e.g.*, 4 KB), at the price of higher performance overhead (around 10×). On the other end, developers can choose to reduce the performance overhead with a larger vCache size (*e.g.*, enough for the working set), at the price of reducing security guarantees. However, in practice, with re-randomization of vCaches, even the reduced security guarantee is reasonable for most applications (see §6).

We have implemented KLOTSKI based on the Intel SDK for Linux, LLVM toolchain, and musl-libc. Our experimental evaluation demonstrates that (1) KLOTSKI is effective against known controlled-channel inference attacks against enclave programs, (2) KLOTSKI has good compatibility with enclave programs, and (3) the performance overhead imposed by KLOTSKI can be reduced to 1.3× on real programs while providing reasonable security guarantees.

In summary, this paper makes the following contributions:

- **New ORAM-based defense against controlled-channel attacks.** We designed and implemented a new obfuscated execution technique to protect enclave programs against controlled-channel attacks. Our security evaluation show our design is able to prevent all attacks under our threat model.
- **Optimization techniques.** We developed several optimization techniques to reduce the overhead. The evaluation also shows that our optimization techniques are very effective, which can improve the performance by as much as 6.7×. For real-world applications, KLOTSKI's performance is also acceptable, only 2.3× with a good balance for security.
- **Open-source implementation.** We implemented an end-to-end toolchain that supports a variety of enclave programs. The source code and documentation will be open to public upon the acceptance of this work (https://github.com/nczhang88pan/KlotskiSGX.git).

2 Background

2.1 Intel SGX

SGX provides two security guarantees: confidentiality and integrity. First, it prevents code and data that belong to an enclave from being accessible outside the enclave, including privileged softwares, like an OS and a hypervisor. Second, it uses a memory encryption [22] to prevent memory attacks like snooping and cold boot. It also maintains integrity measures of enclave memory to prevent malicious tampering and replay attacks. When a hardware exception/interrupt occurs inside an enclave, the processor generates an *Asynchronous Enclave Exit* (AEX) before invoking system software's exception handler. SGX first saves the enclave's execution states to a *State Save Area* (SSA) and resets all registers to predefined values to avoid leaking secrets. For example, when a page fault occurs, SGX would clear the lowest 12 bits of the faulted address. Then the control is transferred to the exception handler. Finally, after finishing the process, the handler resumes the enclave program.

2.2 ORAM

Oblivious RAM (ORAM) [18], provides a formal and general model to prevent adversaries from learning anything about the input of a program from its memory access patterns. That is, given two inputs i and i', their memory access traces are computationally indistinguishable. ORAM achieves this goal by obfuscating the memory access patterns through adding noise, permutating and reshuffling randomly. There are numerous ways to construct an ORAM. A simple but extremely inefficient construction is that for every memory access, accesses the whole memory. Therefore, several different ORAM schemes [38, 49, 50, 55] have been proposed to improve ORAM's efficiency. In this subsection, we focus on explaining Ring ORAM [38], which is used in KLOTSKI. Ring ORAM is an optimization of Path ORAM [50]. It consists of three components:

- **ORAM Tree**: a complete binary-tree in an *untrusted* server to store encrypted memory blocks. The depth of this tree is typical $O(\log N)$ where N is the number of real memory blocks. Each node of this tree, also known as a bucket, has a fixed number of slots to hold blocks ($Z + S$, up to Z slots may contain real blocks, and at least S slots are filled with dummy blocks), and a small metadata containing basic information. Since all real blocks and dummy blocks are encrypted, an attacker is not able to distinguish them.
- **Position Map**: a lookup table that is reserved by the *trusted* client and used to record which path in the ORAM tree a real block maps to.
- **Stash**: a buffer in the *trusted* client, that stores the blocks which have not been evicted to the ORAM tree. A block is either in the ORAM tree or the stash.

Ring ORAM accesses a block in 4 steps:

1. **Position Map lookup**: The ORAM looks up the position map to learn in which path l the target block b currently resides and assigns a new path l' to the block.
2. **ReadPath**: The ORAM reads all buckets along the path l and stores b into the stash. Unlike prior tree-based schemes, only *one* block is read from each bucket along the path. Except for the bucket where b resides, a random dummy block is read.
3. **EvictPath**: To keep the stash occupancy low, after every A `ReadPath` accesses, `EvictPath` selects a path, reads Z blocks (all the remaining real blocks and potentially dummy blocks) from each bucket into the stash and then fill the path with blocks in the stash in the *reverse lexicographical order* [38].
4. **Early Reshuffles**: To avoid a bucket from being read $> S$ times before `EvictPath` reshuffles the bucket. For each `ReadPath` operation, `EarlyReshuffle` is performed on the bucket(s) that have been accessed more than S times. Similar to `EvictPath`, `EarlyReshuffle` reads Z blocks and writes back $Z + S$ permuted blocks.

3 Related Work and Motivation

3.1 Side-channel Attacks against SGX

Table 1 summarizes and compares the existing side-channel attacks on SGX from various aspects. Xu *et al.* [56] first demonstrated a malicious OS can extract a program's secrets by observing its memory accesses at page granularity. In this attack, they used page faults as a noise-free controlled-channel. In order to reduce the extraordinary overhead introduced by frequent page faults, they used several sets of page-fault sequences to identify the start and the end of functions of interest, and only perform heavy page fault tracking for these functions. Even with this optimization, their attacks resulted in an overheads of 209.6× to 354.9×. J. Bulck *et al.* [12] then presented a page-table-based attack that generates fewer page faults. They carefully picked some *trigger page(s)* and collected the set of accessed pages between two successive accesses to trigger pages using the A(ccessed)/D(irty) flags in page table entries. Using this method, the victim enclave is only interrupted when a triggering page is accessed. TLBleed [20] measured dTLB latencies to gather page-granular signals for data pages. This attack collects the activities in the TLB for 2 ms during a single signing operation.

Page-table-based attacks face a fundamental limit on the temporal and spatial resolution, attackers cannot observe memory accesses within a single 4KB page. So more fine-grained side-channel attacks are proposed. Cache-based attacks [9, 19, 23, 35, 48] demonstrated that Prime+Probe can still be leveraged to observe cache-line-level memory accesses of an enclave. Branch shadowing [30] leveraged the

Table 1. Related SGX Side-channel attacks

	Resolution	Attack Target	Side-channel	Side-channel pattern	AEX frequency	Comments
Xu et al. [56]	Page	accessed pages	page fault exception	page fault sequences	every page fault	~209.6x to 354.9x
J. Bulck et al. [12]		specified page sets	A/D bits in PTEs	accessed pages in between the access of a trigger page	each access of the trigger page	
TLBleed [20]		specified data pages	dTLB latency	accessed data page sequences	N/A	collect the activities in the TLB for only 2 ms
F. Brasser et al. [9]	Cache	lookup tables	L1 cache latency	cache access footprint	N/A	Muti-iterations
A. Moghimi et al. [35]					every ~40 cycles	
J. GÃűtzfried et al. [19]					N/A	
HÃďhnel et al. [23]		instructions with memory operands			single stepping	~3532x
Skarlatos et al. [48]						No Noise
branch shadowing [30]	Branch	branches	branch-prediction	branch misprediction penalty	every ~50 CPU cycles	
SGX-Step [11]	Misc	N/A	N/A	N/A	single stepping	
Nemesis [53]		instructions	IRQ latency	Interrupt arrival timing	single stepping	

branch prediction latency to gather precise control-flow information. Similar to traditional side-channel attacks, the key challenge for these attacks is *noises* caused by other uninterested accesses at probing phase. Most of the attacks mentioned above mitigated this issue by exploiting the x86 local APIC timer to trigger interrupts as frequent as possible to approximate *single stepping* [11], so the malicious OS can break into an SGX application right before and after the access of interested memory to reduce pollution. However, this also makes these attacks extremely slow. Even after using page-level side-channel to identify target function of interest, Hähnel *et al.* [23] reported a 3532× slow down on observing lookups of a single array.

Finally, Foreshadow [10] demonstrated a new transient execution attack, which can dump the entire contents of a victim enclave even without any cooperation within it.

Observation. Most SGX side-channel attacks rely on frequent AEXs to pause the execution of an enclave to reduce noises, which also increases the program's overall execution time. This problem becomes more severe in fine-grained side-channel attacks. To make them practical, attackers only launch fine-grained attacks after leveraging coarse-grain side-channel to pinpoint function(s) of interest. Therefore, we can hinder these fine-grained attacks by preventing coarse-grained side-channels.

3.2 Existing Defenses

T-SGX [44] used Intel *Transactional Synchronization eXtensions* (TSX) to limit the generating of an enclave's AEXs while the critical code is executing. Chen *et al.* [15] also leveraged the TSX to implement an execution-based reference clock for detecting the interruption of critical executions. The problem of TSX-based solutions is that some attacks [9, 19, 20, 54] acquire access pattern without triggering AEXs. SGX-Shield [43] introduced a code block-level randomization at the load time of an enclave program. However, a malicious OS can still infer an enclave's memory layout by observing its memory access patterns on an input from the OS [30]. Zigzagger [30] obfuscates a set of branch instructions into a single indirect branch to prevent branch-based side channel attacks but it only protects code execution and have been defeated by some fine-grained attacks [11, 53]. ENCLANG [47] obfuscates leaf functions which do not call other functions. DR.SGX [8] continuously re-randomizes all enclave data at the granularity of cache lines during an enclave's execution. But it only focuses on data accesses and does not obfuscate code execution. Shinde *et al.* [45] introduced an oblivious execution approach called deterministic multiplexing that places sensitive code and data to one page to hide page access patterns. Their scheme imposes a very high performance overhead (over 4000×) and requires manual optimization. OBLIVIATE [3], ZeroTrace [39], and OBFSCURO [2] use ORAM protocol to obfuscate memory accesses. ZeroTrace only considers data obliviousness. OBLIVIATE provides an obfuscated file system for an enclave but does not eradicate the controlled-channel for its execution. OBFSCURO obfuscates accesses of both code and data. While it provides strong protection against both cache and timing side-channels, it can only support programs of small code and data size (8KB); hence is not practical for real programs. Besides, its performance overhead is significantly higher: 55× on their customized small benchmarks.

InvisiPage [1] proposes a new hardware design that is similar to Klotski, where a program inside enclave can populate its own page table and its page access pattern is protected by an ORAM protocol. While a hardware implementation could provide better performance, Klotski can be applied to existing hardware and all of Klotski's compile-time optimizations are also applicable to InvisiPage.

Observation. Existing defenses suffer from three limitations: (1) incomplete protection against even coarse-grained side-channel attacks, (2) incomplete protection of both code and data, or (3) high performance overhead. Klotski aims to address these limitations by providing a subpage-level runtime randomization scheme for both code and data. This scheme breaks the fixed relationships of a program's addresses and contents. In addition, Klotski uses an ORAM-based shuffling approach to prevent possible information leakages during runtime re-randomization. As a result, Klotski can prevent attackers from locating the access of code/data of interest via coarse-grained side-channel attacks. Although Klotski does not directly mitigate fine-grained side-channel

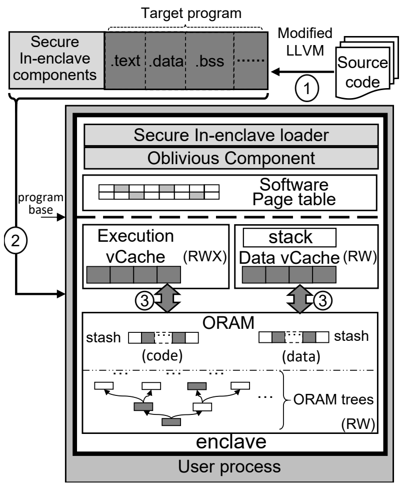

Figure 1. Overview of Klotski

attacks, it forces attackers to monitor code/data access all the time, which would make the attack extremely slow and impractical.

4 Design

4.1 Threat Model

We assume the same threat model as previous controlled-channel works [44, 45, 56]. First, except the enclave itself, all other components in the software stack are not trusted, so the page access pattern is visible to the adversary. Second, we assume the code of a target program, including its source code and binary code, is known to the adversary; so an attacker can perform any automatic or manual analysis on the program to extract necessary information. Moreover, we assume that an attacker can feed attacker-controlled inputs to the target program, so as to perform online training to learn the memory access pattern. Finally, since our focus is on preventing side-channel attacks, we deem any attacks exploiting software vulnerabilities (*e.g.*, buffer overflow) within the target program out of scope. Any defense that mitigates these exploits is complementary to our work.

While our solution leverages ORAM to mitigate side-channel attacks, our threat model is much stronger. Existing ORAM solutions assume a client-server model. In this model, the server is untrusted, but the client is trusted. This means all client-side operations are not subject to side-channel attacks. In contrast, in our scenario, all components are running inside the same enclave thus are subject to side-channel attacks.

4.2 Overview

In this work, we mitigate side-channel attacks using a runtime re-randomization-based obfuscation approach. Figure 1 illustrates an overview of Klotski, which includes two main components: a compiler extension and an ORAM-based runtime. The compiler extension performs two transformations.

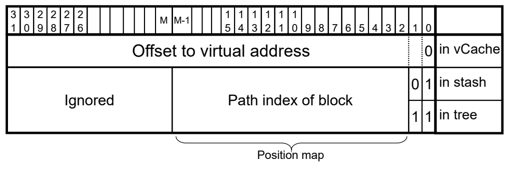

Figure 2. Format of a 32-Bit Page Table Entry in Klotski

First, it instruments all memory access instructions, including control-transfer (code access) instructions of an enclave program to go through the runtime. Second, it divides the program's code and data into small chunks that can be relocated to any virtual address. The runtime acts as a memory subsystem which obfuscates memory accesses using an enhanced Ring ORM protocol [38]. To speed-up the execution, the runtime also consists of two memory buffers of configurable size, which serve as code cache and data cache. Note, to avoid confusing between Klotski's software cache and architecture/hardware cache, we will use vCache to denote Klotski's cache. Code and data chunks are fetched into the vCaches using the ORAM protocol. vCache re-randomization is done through either nature replacement (with a random replacement policy) or forced flush.

4.3 Software MMU

Klotski builds a target program's source code into a Klotski-enhanced version with a modified compiler. The overall goal is to instrument memory access operations to consulting the software MMU to translate an original logical address into a real virtual address in the vCache. This stage is performed at a trusted machine.

Translation Data Structure. To reduce the overhead of each address translation, Klotski's software MMU only uses one level page table. Each *page table entry* (PTE) is 4 bytes long. During translation, an input address (64-bit) is split into two parts: page index (against the base address of the enclave) and page offset. This split depends on the ORAM block's size. **Note**, to avoid confusing between a *basic block* of the program and an *ORAM block*, we will use *mini-page* interchangeably with ORAM block. For instance, in our prototype implementation, the mini-page size is 2 KB. The software MMU uses the mini-page index to look up the corresponding PTE. A valid PTE records the *offset* between the compile-time logical address and the runtime virtual address. Hence, the software MMU can efficiently calculate the runtime virtual address by just adding the offset (*i.e.*, PTE's value) to the logical address. Klotski guarantees that all offsets are within the valid range of a 32-bit PTE.

Since only mini-pages that have already been loaded into the vCache have valid PTEs, we also use the page table as a virtual position map, similar to how OS kernels implement virtual memory (in Figure 2). If a mini-page is in the vCache (valid, `00`), then its PTE records the offset, as described above. If a mini-page is in the stash (`01`), then its PTE records its new path in the ORAM tree. If a mini-page is in the ORAM tree (11), then its PTE records its current path in the tree.

Explicit Memory Accesses. Explicit memory accesses include branches (`jmp, call, ret`) and data accesses (`load, store`). Since a mini-page's effective virtual address changes at runtime, in addition to instrumenting memory access operations, we need to ensure that all the addresses (pointers) stored in the memory are original logical addresses instead of virtual addresses. As a result, we need to handle the `call`, `ret`, `jcc`, and `lea` instructions specially. A `call` instruction pushes the effective virtual address as the return address onto the stack. To fix this issue, during instrumentation, KLOTSKI will replace each `call` instruction with a `push` instruction followed by a `jmp` instruction. In doing so, the logic address is pushed on the stack, and the `jmp` instruction takes the effective virtual address as an operand, and transfers execution to the target function. To fix a `ret` instruction, KLOTSKI replaces each `ret` instruction with a `pop` and a `jmp` instruction and inserts the address translation logic in between. Conditional jump instructions are more complicated because they use relative addresses (*i.e.*, the operand is the offset between the current address and the target address). To handle them, KLOTSKI replaces the target with a trampoline to perform address translation and control transfer.

A `lea` instruction may load the virtual address into a register. In KLOTSKI, since the code is not compiled as position independent, code pointers are generated statically instead of with `lea`. So the only special case is to obtain a stack address. In this case, we need to translate the loaded virtual address back to logical address before it is stored to memory or passed to another function. Because stack addresses are not determined at compile time, the conversion is done by (1) preserving a region (*i.e.*, a set of PTEs) for stacks, (2) assigning a unique PTE as the logical base of the stack, and (3) storing the logical base of the stack in the thread local storage for future conversion.

Implicit Memory Accesses. An implicit memory access happens when the processor attempts to fetch the next instruction. If the next instruction is in the same mini-page, no instrumentation is needed; otherwise, an explicit control transfer should be inserted. This is done in two steps. First, when emitting machine code, KLOTSKI ensures that no basic block will across the boundary of a mini-page. Second, similar to the conditional jump, KLOTSKI inserts a pair of `mov` + `jmp` instructions at the end of each basic block that has an implicit fall-through. During target code generation, if both the source and target basic block are in the same mini-page, KLOTSKI discards the explicit fall-through instructions to avoid unnecessary translations.

OCalls. External Library functions, system calls, and I/O instructions such as `sendto()` and `recvfrom()` are not supported within an enclave thus need to be wrapped as `Ocalls`. Because these wrappers may access data objects that cross multiple non-consecutive mini-pages, we also need to modify them to be compatible with KLOTSKI. Specifically, for `Ocalls` that transfer data from the enclave to outside (*e.g.*, `sendto`), they will first allocate a consecutive buffer and then copy the data object from KLOTSKI's vCache to the buffer. For `Ocalls` that transfer data from outside into the enclave (*e.g.*, `recvfrom`), they will invoke the software MMU to write the data back to data vCache carefully.

Bootstrapping. To construct the desired memory layout, KLOTSKI initialized the enclave in two steps: (1) enclave bootstrapping deployed by the untrusted OS, creates and initializes an enclave for the SGX program, and (2) secure in-enclave bootstrapping initializes both the SGX program's memory layout and KLOTSKI's data structures in the enclave. During enclave bootstrapping, two regions of consecutive pages are allocated with read and write permissions (`RW`). One is for in-enclave program's data (*i.e.*, data vCache and stack) and the other is for the ORAM (*i.e.*, stash and ORAM tree). The ORAM tree will be used to store all data and code sections of the enclave program, as well as the heap. A fixed number of pages with the permission of read, write, and execution (`RWX`) are allocated as the execution vCache. The in-enclave loader, the software MMU, and the ORAM module will be mapped as parts of the program. Since they will be invoked frequently, they will remain at their initial loading addresses.

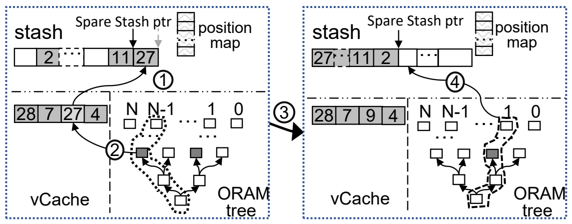

Figure 3. Requesting a mini-page from ORAM: This example requests a mini-page (index 9) from the ORAM tree and does a PathEvict to refresh the `Path 1`. The processing contains 4 steps: ①: `EvictCache` randomly picks a vCache slot and swaps its mini-page to stash. ②: `ReadPath` loads the target mini-page (Index 9) into the reserved vCache slot. ③: the `Spare Stash ptr` occurs the stash boundary, so `Oshuffle` permutes the order of mini-pages in stash to reset the pointer. ④: `EvictPath` reads mini-pages from the `Path 1` and then refreshes the buckets along it.

4.4 ORAM Access

When a mini-page is not found in the vCache, the control flow is transferred to the ORAM component to load the mini-page into the vCache. The overall process is similar to the Ring ORAM protocol described in §2. However, a big difference between traditional ORAM's threat model and KLOTSKI's threat model is that we do not have a trusted "client"; so accesses to stash and position map are also subject to side-channel attacks. To mitigate these threats, KLOTSKI adds additional oblivious operations and enhances some existing operations.

- **ORead** is a primitive operation to read from a target mini-page without revealing which mini-page is the target. Similar to `cmov` operation, in a single iteration, all the related mini-pages are accessed, but only data in real mini-page is read into a 256-bit `ymm` register while data from other mini-pages is read into a dummy register; then it iterates again until all available `ymm` registers are filled.
- **OShuffle** permutes the order of a list of mini-pages without revealing the new order.
- **WriteStash** `ReadPath`, `EvictPath`, and `EvictCache` all need to write mini-pages to the stash. Two types of information could be leaked during this operation: (1) writing a mini-page to a slot means the slot is empty, so it must have been evicted previously; and (2) if a slot has been written more than once without being read, then its previously stored content must be dummy. To mitigate this threat, Klotski re-randomizes the stash to obfuscate the position of the real, dummy, and empty slots. Specifically, Klotski remembers the index of the last written slot (*Spare Stash Pointer* in Figure 3). When trying to find the next empty slot, Klotski scans from this location to the end of the stash. If an empty slot is found, then it will be used to store the mini-page and the pointer will be updated. If the pointer reaches the end of the stash, an `OShuffle` operation is performed and then the pointer is reset to the beginning of the stash.
- **ReadPath** reads one mini-page from each bucket along with a selected path and then only stores the mini-page of interest to the stash. In traditional ORAM threat model, the store operation is not observable to the adversaries so they cannot know which bucket contains the target mini-page. However, in Klotski, directly copying the target mini-page to the stash will leak the information that current bucket contains the real mini-page. To mitigate this kind of risk, Klotski uses the `ORead` operation to read all buckets along the path and then uses `WriteStash` to write the target mini-page to stash.
- **EvictPath** deterministically reads Z mini-pages from every bucket along the path into the stash, so it does not require `ORead`. However, it does use `WriteStash` to write to the stash. After shuffling, to avoid leaking which stash slots are evicted, `EvictPath` uses `ORead` to load the mini-page to be evicted and write it back to the target bucket.
- **LoadCache** uses `ORead` to "scan" the stash and load the target mini-page into an empty vCache slot if the target mini-page is in stash; otherwise, it invokes `ReadPath` to directly load the target mini-page into an empty vCache slot. Finally, `LoadCache` updates the mini-page's PTE to replace its path information with the offset information (see §4.3).
- **EvictCache** *randomly* picks a mini-page and uses `WriteStash` to evict it to stash when the vCache is full and a new mini-page needs to be loaded. We use the *random replacement* (RR) instead of more commonly used policies like *least recent used* (LRU) or Pseudo-LRU because our (a) RR can avoid leaking information and (b) Klotski's vCache is relatively small. Since the path information is lost during `LoadCache`, a new path is assigned when the block is evicted from the cache.
- **ReadPTE** reads a PTE from the software page table. Because the page table (virtual position map) is mapped at a fixed address, attackers may figure out which logical address is accessed through the access to the position map. To mitigate this threat, Klotski uses an oblivious access operation based on the `Oget` function from [?] which iteratively applies the `vpgatherdd` instruction to read an element from an aligned 512-byte array obliviously.
- **WritePTE** leverages the non-temporal write instructions [8] to update a PTE. Non-temporal write instructions immediately affect the DRAM and do not buffer the data into the cache hierarchy. Therefore the updating is invisible to an attacker.

5 Optimization

The baseline design of Klotski incurs a high runtime overhead, which mainly comes from two sources: (1) additional software address translation logic and (2) vCache misses (*i.e.*, ORAM accesses). In this section, we present several optimization techniques to reduce the performance overhead of Klotski.

Note that these techniques only optimize accesses to the vCaches, instead of the ORAM protocol itself (*e.g.*, making access to ORAM faster), so they would not affect the security guarantees of the ORAM protocol.

5.1 Address Translation Reduction

The baseline implementation of Klotski inserts the translation logic before every memory access. Klotski reduces the number of address translations by eliminating redundant ones: when a new pointer is derived (adding or subtracting an offset) from an older pointer whose address has already been translated, we can reuse the already translated address as long as (1) the new pointer is guaranteed to be within the same mini-page, and (2) the corresponding mini-page has not been evicted from vCache. Based on this observation, Klotski performs the following optimizations.

Code. To optimize code accesses, Klotski first tries to place an entire function into a single mini-page. Next, for each control transfer, Klotski checks if its target address is inside the same mini-page; if so, it eliminates the address translation.

Data. Optimizing data accesses are more complicated. The first challenge is the contiguous policy. In particular, while it is possible to enforce that small objects will never across the

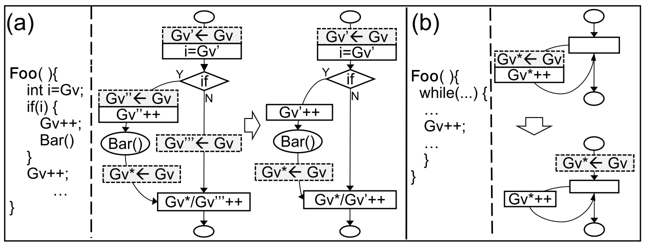

Figure 4. Examples of how Klotski caches translation results. Typtically `translation` instructions (filled) are inserted before every memory access instruction (right side). In example (a), if condition is not met (`i==0`), `Gv'` is still valid at ④; so there is no need to translate the address. If `i != 0`, calling function `Bar` kills all previous translations and needs to translate `Gv`'s address again before accessing ④. The right side is the optimized one. In example (b), Klotski moves the translating of `Gv's` address out of the loop's body.

boundary of a mini-page, objects that are larger than the size of a mini-page will inevitably occupy multiple mini-pages. So the first step to optimizing data access is to identify which memory accesses can be optimized.

Large objects. To identify memory accesses to large objects, we used a conservative inter-procedural field-sensitive data-flow analysis based on the *type-based alias analysis* (TBAA) from LLVM. For each heap allocation site, we mark the return value (pointer to the allocated object) as "tainted" if the allocation size is larger than the size of a mini-page or if the allocation size is unknown at compile time. We then use the data-flow analysis to find all memory access instructions that may access large objects (*i.e.*, dereferencing a tainted pointer). The rest memory accesses are thus safe to be optimized. We also modified common data accessing library functions like `memset, memcpy, memmove` to support going beyond the boundary of a mini-page.

Small objects. Optimizing access to small heap or global objects is similar to the code: once translated the base address of an object, future accesses to the same object do not require additional address translations. Klotski uses a classic liveness analysis to track logical addresses that have already been translated. When instrumenting the code, for each memory access (*i.e.*`load, store`), Klotski first checks if the target address derives from a virtual address from the liveness set. If the virtual address is not found, Klotski inserts a translating instruction pair and adds the result to the set (Figure 4).

The second challenge for data access is that a translated address will become invalid when the corresponding mini-page is evicted. Since it requires very sophisticated analysis to predict when a mini-page would be evicted. To solve this, Klotski takes a simple way: (1) it invalidates all translated addresses when calling another function and (2) Klotski will only evict data vCache at function call and return.

Finally, Klotski further minimizes the number of translations by promoting a translation logic closer to the entry of a function, as long as it will not be killed. This is especially helpful to avoid translating the address repeatedly inside a loop (Figure 4 (b)).

5.2 Loop Alignment

Since instructions inside loops execute repeatedly, they account for a large portion of a programś execution time. Thus, optimizing loops is critical for improving the program's performance. In Klotski, if a basic block and its successor(s) do not reside in the same mini-page, then address translation will be inserted at the end of this basic block (§4.3). If this happens inside a loop body, then it can incur very high performance overheads. To avoid such behavior and improve the locality of the program, Klotski tries to align loops such that the body of most loops will not across a mini-page boundary. If a loop body is too large to fit into a single mini-page, then Klotski will still leave its basic block split into two mini-pages. Fortunately, such large loops are rare.

5.3 Constant Embedding

To reduce the size of a program and improve performance, a common compiler optimization is to merge and move constant values (especially floating-point constants) to the program's data section; thereafter loads these constants from the memory into registers during a runtime. While this is an effective optimization strategy for regular programs, it causes troubles for Klotski-enhanced programs. First, access to each of these constants would cause an address translation, which is expensive. Second, these constants are stored in the data section, an access to a constant value would require loading it from the data section into the data vCache. Due to the limited size of Klotski's data vCache, this also results in additional vCache load and eviction.

To minimize this overhead, Klotski employs another optimization: it embeds constants that aught to be accessed by a basic block into a spare space of the corresponding mini-page. This optimization allows a basic block to use PC-relative addressing to access constants without translating the address and affecting the data vCache.

5.4 Configurable Mini-page and vCache Size

Our design of Klotski is generic, so the sizes of the mini-page and the vCache are configurable. However, when configuring these parameters, one must carefully consider the trade-off between security and performance. Specifically, a smaller mini-page size may improve the security [2] and the vCache hit rate; but it will also increase the number of address translations as well as the cost of each translation: smaller mini-page size requires more PTEs to index, and the cost of `ReadPTE` (§4.4) is proportional to the size of the page table. Similarly, from the security perspective, the ideal size of the vCache is one physical page (*i.e.*, 4 KB) under our threat model. However, it is well-known that when the vCache size is smaller than the working set of the program, it would end up with *thrashing* [17], *i.e.*, the system will spend most of the time copying data to/from ORAM. Because thrashing cannot be avoided through vCache replacement algorithms, from the performance perspective, a larger vCache size is preferred. While smaller vCache is implicitly re-randomized via frequent replacement, larger vCaches require explicit re-randomize (by flushing). Klotski allows user to configure the flushing frequency (*e.g.*, after a fixed number of code address dereferences).

6 Security Analysis

In this section, we analyze the security of Klotski under our threat model (*i.e.*, controlled-channel attacks).

6.1 Baseline Design

We first analyze the baseline design of Klotski (§4) with 4 KB vCache size. We start the analysis with each important step of Klotski.

Claim 1. Accessing blocks from the ORAM tree leaks no information. The algorithm of Ring ORAM ensures that even though its subroutines, including `ReadPath`, `EvictPath`, and `EarlyShuffle` may have externally observable behaviors, they leak no useful information. Moreover, Klotski uses the `ORead` operation (see §4.4) to prevent attackers from distinguishing real/dummy blocks and inferring the remaining blocks inside a bucket.

Claim 2. Data exchanges between the stash and vCache leak no information. Klotski utilizes `ORead` operation to load blocks from the stash. Because `ORead` will access every block in the stash at each time, the pattern is oblivious under our threat model. When evicting a vCache block to the stash, Klotski uses the random replacement policy, so the evicting block is independent of the program's past access pattern. Furthermore, Klotski uses `WriteStash` operation to store blocks into the stash. This operation also reveals no access information as the distribution of empty slots is re-randomized during the shuffling. Thus, attackers cannot get useful information from the data exchange steps.

Claim 3. Accessing the page table (and the virtual position map) leaks no information. Klotski uses the `ReadPTE` and `WritePTE` operations to access the position map. Since they are oblivious operations, this step leaks no useful information (*i.e.*, attackers cannot find out which logical block is being accessed).

Claim 4. Execution inside the vCache leaks no information. When the size of both the execution and the data vCache is set to one physical page, the execution exhibits the same memory access pattern to an attacker, thus no executing information leaks under our threat model. This is identical to the claim made by Shinde *et al.* [45].

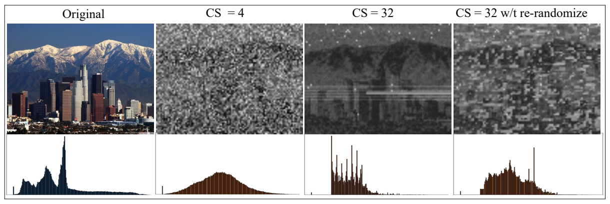

Figure 5. Controlled-channel attacks against Klotski-enhanced `djpeg` under performance-oriented configurations. CS denotes the vCache size. The second row shows the pixel distribution of each image in the same column. Horizontal axis represents pixels. Vertical axis expresses pixel numbers.

Because every step of Klotski's execution leaks no information, we conclude that Klotski is secure (*i.e.*, page-access-oblivious) under our threat model.

6.2 Performance-Oriented Configuration

While the baseline design of Klotski is secure, it also has a high performance overhead, especially when the working set is larger than the vCache size (see §7). In this subsection, we analyze the security of Klotski when developers opt for better performance by increasing the vCache size.

Because vCache size is irrelevant to Claim 1, 2, 3, they still hold under this configuration. The only problematic one is Claim 4—when the vCache size is larger than one physical page, the execution could exhibit distinct page access pattern. This distinction is especially dangerous when the vCache contains the whole working set. However, we argue that even with configuration, launching controlled-channel attacks against Klotski is still very difficult in practice. First, the random vCache eviction policy means the same block can be loaded into any random slot of the vCache. Second, as the mini-page size in Klotski is usually less than a physical page (*e.g.*, 2 KB in the prototype implementation), the combination of two arbitrary blocks will generate new access patterns which introduce additional noise thus make it more difficult to attack. Finally, when Klotski re-randomizes vCache (through vCache flushing), any information acquired through previous profiling will be invalidated. So even an infrequent rate of re-randomization would make it very difficult to carry information learned from previous (*e.g.*, attacker-controlled) input to the next input.

Empirical Evaluation. We replayed the JPEG and the FreeType attacks [56] with various vCache size configurations. One major difference is that because Klotski prevents us from using the way in the original attacks to infer the start and end of the interested function, we had to manually modify the two programs' source code to insert a special OCall to notify us the start and end of the function of interest.

Figure 5 demonstrates part of the results of our JPEG attack. With a vCache size of 16 KB (4 + 4 mini-pages), it is very difficult to recognize the content from the recovered image; and the distribution of pixel values of the restored image is very smooth. When the vCache size is increased to 128 KB

(32 + 32 mini-pages), the working set of the vulnerable functions can fit into the vCache. Without re-randomization, the attack sucessfully recovers many pixels of the original image. However, even with a low re-randomization frequency (once after every 1280 address translations), most details become unrecognizable and its pixel distribution is relatively smoother than the one without re-randomization. For the FreeType attack, the accuracy of predicting a word was only 0.04%, 0.23%, and 8.42% when the vCache size was set to 16, 32, and 64 KB and without a random flush.

We want to emphasize again that, in these two attacks, we had given the adversary an advantage to precisely distinguish the boundaries of the target functions. In practice, because with vCaches being repeatedly re-randomized, it is not easy to distinguish which mini-pages are in the vCache. Based on these empirical results, we recommend a 2.5% evicting rate of the execution vCache to reach a balance between the security (see Figure 5) and the performance (see §7.2).

6.3 Other Side-channel Attacks

Cache-based side-channel attacks [9, 10, 19, 23, 42] and timing-based attacks [11, 53] have been successfully demonstrated against SGX enclaves to infer fine-grained access pattern. Although Klotski is not designed to fully mitigate these attacks, it still makes them very difficult to succeed. Specifically, all input inference attacks require to know when the input-dependent functions/instructions are executed, which is especially important for fine-grained attacks because a single inference will require significant amount of time. To accelerate the attack, they usually start the fine-grained attack after detecting the execution of some specific functions of interest, by leveraging enclave preemption via page faults [56] or a dedicated spy thread [12]. By performing regular vCache re-randomization and using random vCache eviction policy, mini-pages in Klotski's vCache are continuously changing; so, it is hard to know when the functions of interest will be swapped into or out from the vCache. Moreover, because every 4 KB physical page in Klotski's vCache is formed by combining two arbitrary mini-pages at the runtime, attackers cannot use a fixed page access pattern to locate the functions of interest. Therefore, to launch a successful attack, attackers need to find good way to overcome these challenges. More importantly, if the vCache are re-randomized before the attacks can pinpoint the function of interest, it would be almost impossible to launch the attack. Recall that in our empirical evaluation we had to modify the target programs to notify us when the functions of interest were executed, which is unrealistic in practice.

7 Performance Evaluation

Environment Setup. All experiments were carried out on a machine with an Intel Skylake i7-6700 @ 3.4GHz, 16 GB RAM and 128 MB PRM. The machine is running with an operating system of Ubuntu 14.04 64-bit with Linux kernel 4.4.0 and the Intel SGX SDK (version 1.9.100) [28] and Intel-SGX-driver (version 1.8) [27].

We run Klotski on both benchmark suites and real applications with a general Ring ORAM configuration: each bucket contains six slots which up to two slots are reserved for real mini-page ($Z = 2$), and the remains are for dummy mini-page ($S = 4$). After every four oblivious accesses ($A = 4$), a path eviction operation would be taken. A mini-page size is set at 2 KB. All programs were compiled with optimization level 2 (`-O2`). Unless otherwise specified, the size of the execution vCache and the size of data vCache are equal.

7.1 Nbench Benchmark

We chose the Nbench benchmark suites [13] to evaluate the performance overhead, which is also used in SGX-Shield [43] and T-SGX [44].

In Klotski, two primary sources of the overhead are software address translation and ORAM access. To demonstrate the overhead of these two sources and the benefits of different optimization techniques, we evaluated Klotski with multiple configurations. For each configuration, the result was measured over 100 runs. For each test case, an iteration number was selected to yield a total time greater than 5 seconds.

When obfuscated execution is enabled (Table 2), performance overhead varies greatly across benchmarks and configurations. Generally, benchmarks with lower vCache hit rates exhibited higher performance overheads. When the vCache sizes are increased, the relative overheads decreased from 10.22× to 3.46×. After examining the benchmark source code, we found that: almost every benchmark uses a customized heap allocator that allocates smaller chunk of data from a large memory buffer. Unaware of this characteristic, Klotski treated the large buffer as a single memory object and disabled address translation optimization. Therefore, this result can be considered as one of the worse case scenarios for Klotski.

To estimate the performance overhead under normal circumstances (*i.e.*, when small objects are directly allocated from our modified heap allocator), we disabled the detection of large objects during compilation (§5.1) and evaluated again (the right half of Table 2). For benchmarks with a small working set (`Num sort`, `String sort`, `Assignment`, `Huffman`, and `Lu.decomp.`), their EC hit rates are above 98% when the vCache size is 4 KB, so their performance overhead is relatively the same with larger vCache size. For benchmarks with a moderate working set (`Fp.emu`, `Idea` and `Neural net`), their EC hit rate improved significantly when the vCache size is increased from 4 KB (2 mini-pages) to 8 KB (4 mini-pages), so their performance overhead also dropped significantly. For benchmarks with a large working set (`Fourier`), the overhead only drops significantly when the vCache size is increased to 16 KB (8 mini-pages). We dived into this case and found

Table 2. The performance overheads of KLOTSKI with different vCache size. The baseline is the native run without KLOTSKI in an enclave and shown in microseconds. *EC* = and *DC* = represent the maximum number of mini-page in the execution vCache and data vCache. *RO* represents the relative overheads compared to the baseline in a multiple (×) or percentage (%). Columns under *EChit*/*DChit* are the hit rate of execution/data vCache. The average standard deviation is 0.075% (the maximum is 0.53%)

	Baseline	**full features**									**disabled big objects**					
		EC=2,DC=2			**EC=4,DC=4**			**EC=8,DC=8**			**EC=2,DC=2**		**EC=4,DC=4**		**EC=8,DC=8**	
	(μ)	**RO** (×)	**EC hit** (%)	**DC hit** (%)	**RO** (×)	**EC hit** (%)	**DC hit** (%)	**RO** (×)	**EC hit** (%)	**DC hit** (%)	**RO** (%)	**EC hit** (%)	**RO** (%)	**EC hit** (%)	**RO** (%)	**EC hit** (%)
Num sort	329	58.59×	81.78	76.38	23.75×	91.75	92.86	13.88×	93.93	96.29	1.46	99.97	2.19	99.98	1.04	99.99
String sort	4383	2.06×	94.35	93.37	1.84×	97.82	98.03	1.41×	99.99	99.99	7.83	99.94	8.83	99.95	4.98	99.98
Bitfield	0.0013	3.12×	87.36	99.99	3.10×	89.47	99.99	3.09×	93.68	99.99	58.59	99.99	58.09	99.99	58.04	99.99
Fp emu	2555	14.25×	62.96	97.79	7.38×	99.79	99.99	6.83×	99.99	99.99	329.95	89.91	94.77	99.97	92.57	99.99
Fourier	18	52.72×	67.57	36.54	30.02×	80.74	99.99	1.16×	99.99	99.99	5339.66	78.89	3015.65	85.41	14.16	99.99
Assignment	14600	3.08×	99.03	98.95	2.78×	99.95	99.95	2.77×	99.05	99.95	2.50	99.94	3.61	99.96	1.09	99.97
idea	84	3.23×	99.99	99.99	3.16×	99.96	99.98	3.12×	99.99	99.99	1215.58	33.33	13.23	99.99	11.16	99.99
Huffman	227	22.75×	99.67	88.29	3.20×	90.47	99.66	2.90×	99.99	99.99	73.53	99.15	70.63	99.94	69.08	99.96
Neural	8581	41.45×	32.17	30.40	22.82×	89.90	99.99	4.90×	99.99	99.99	2010.82	68.41	151.24	93.99	25.94	99.99
Lu decomp	319	4.67×	98.59	97.19	4.14×	99.98	99.98	4.17×	98.59	99.99	64.58	98.55	11.65	99.99	10.13	99.99
GEOMEAN		10.22×	78.58	76.17	6.17×	93.43	99.02	3.46×	98.49	99.61	256.32	83.14	89.56	97.80	25.52	99.98

the root cause: `Fourier` calls `pow`,`cos`, and `sin` functions in its most inner loop. Because the total size of these three functions is more than 4 mini-pages, thrashing happened when the size of the vCache is smaller than 4 mini-pages.

Effectiveness of Optimizations. For KLOTSKI's optimizations, Table 3 shows their impact on the number of address translations and overall performance, with the large object detection disabled. Overall, *translation reduction* is the most general and effective one. When it is enabled, all benchmarks showed some data translating reduction (ranging from 4.50% to 99.96%, 72.08% on geometric average) and performance improvement (2.09× speedup on geometric average), compared to the baseline. Because the *translation reduction* eliminates most of the data translation instructions, it also decreases the code's size and changes the code's layout. Some of the benchmarks (*e.g.*, `Num sort`) benefit from the decreased code size because more functions can be put into the same mini-page. However, the numbers of code translating increased for other benchmarks (`Fp emu`, `Assign`, and `Idea`). This is because the changes to their code layouts caused some functions/loops to go across mini-page boundaries and resulted in extra inter-mini-pages control flow transfers and additional overheads. *Constant embedding* decreases the data translating of `Fourier` (90.94%) and `Neural net` (68.89%) significantly, because there are a lot of accesses to constants in the loops. However, due to the change of code layout, the number of code address translations of `Huffman` went up significantly (2468×) after this optimization; while the number of translations of previously affected three benchmarks (`Fp emu`, `Assign`, and `Idea`) dropped back to normal. After applying the *loop alignment*, the number of code address translations all decreased to normal. This result highlighted the importance of combining all three optimizations.

Table 3. The effectiveness of address translation reduction. All results are run without ORAM access and big object detection. The *Baseline* is the number of address translations executed per-iteration without any optimization. We add different optimization schemes step-by-step. *C* and *D* denote code and data separately. *TN* denotes the Total Number of address translations in a case. *TR* denotes the Translation Reduction rate comparing to the previous result (at the left side). *SU* represents the speed-up.

		Baseline	**Translation reduction**		**Constant embedding**		**Loop alignment**	
		TN	**TR** (%)	**SU** (×)	**TR** (%)	**SU** (×)	**TR** (%)	**SU** (×)
Num sort	D	673 K	-94.58	3.53	0.00	1.00	0.00	1.01
	C	15 K	-99.96		0.00		0.00	
String sort	D	1391 K	-93.21	1.50	0.00	0.99	0.00	0.98
	C	34 K	1.76		0.00		-0.01	
Bitfield	D	3	-33.33	1.80	0.00	1.01	0.00	0.96
	C	1	0.00		0.00		0.00	
Fp emu.	D	7640 K	-75.41	2.55	0.00	1.02	0.00	0.99
	C	20 K	579.64		-86.87		-8.26	
Fourier	D	9 K	-4.50	1.00	-90.94	3.95	0.00	1.04
	C	3 K	-6.90		7.41		-15.39	
Assign.	D	17775 K	-99.88	2.78	0.00	1.05	0.00	1.01
	C	0.203 K	482×		-99.79		0.00	
Idea	D	60 K	-38.33	1.16	0.00	1.23	0.00	0.98
	C	3 K	466.66		-88.24		0.00	
Huffman	D	258 K	-52.74	1.71	0.00	0.97	0.00	1.06
	C	0.004 K	-0.01		2468×		-99.96	
Neural net	D	35486 K	-93.27	2.86	-68.89	2.42	0.00	0.96
	C	494 K	1.14		2.00		0.00	
Lu decomp.	D	809 K	-96.75	4.17	-0.02	1.04	0.00	0.98
	C	0.207 K	0.00		-96.62		0.00	
GEOMEAN				2.09×		1.29×		1.00×

Table 4 shows the impact of KLOTSKI's optimization on code vCache hit rate, when the vCache size is set to 4 mini-pages. Data vCache hit rate is very high, so we omitted the results on data vCache. Here, we mainly focus on *constant embedding* and *loop alignment*. These two optimization techniques aim to improve the locality of the execution, so their effectiveness depends on the target program. Constant embedding remarkably improves the performance of `Fourier` and `Neural net` as they contain lots of constant accesses. However, constants inserted at the begin of each mini-page also increased the code size and the possibility of thrashing, so the overhead for `Neural net` is increased. Loop alignment notably improved the performance of `Neural net` (2.3× speedup). That is because of the improvement of its locality (code vCache hit rate from 92.26% to 99.99%). Nonetheless, our loop alignment algorithm also increased the code

Table 4. The effectiveness of KLOTSKI's optimization on vCache hit rate. The results are measured with ORAM enabled, and the size of execution/data vCache is set as 4 mini-pages. The *Baseline* columns run with *Translation reduction* enabled. Its left column is the overhead measured in microseconds. Optimization schemes are added step by step to measure the results. *SU* denotes the relative speed up compared to the previous column. *EChit* donates the execution vCache hit rate.

	Baseline		**+ Constant embedding**		**+ Loop alignment**	
	(μs)	**EC hit**	**SU**	**EC hit**	**SU**	**EC hit**
Num sort	375	99.96%	1.13×	99.98%	0.99×	99.99%
String sort	4769	99.95%	0.99×	99.95%	1.01×	99.95%
Bitfield	0.002	99.99%	0.99×	99.99%	1.00×	99.99%
Fp emu.	6511	99.61%	**1.33**×	99.97%	1.00×	99.97%
Fourier	822	57.01%	**1.83**×	76.04%	0.81×	60.41%
Assignment	15204	99.96%	1.02×	99.95%	1.00×	99.96%
Idea	94	100.00%	1.00×	100.00%	1.00×	99.99%
Huffman	376	99.85%	0.97×	99.94%	0.99×	99.94%
Neural net	38463	98.99%	0.78×	92.26%	**2.30**×	99.99%
Lu decomp.	358	100.00%	1.00×	100.00%	1.00×	100.00%
GEOMEAN		94.37%	1.08×	96.49%	1.06×	94.47%

Table 5. The performance overheads in running an HTTPs sever when handling a single request. Columns of `with vCache shuffle` are measured with re-randomizing both vCaches after every 1280 code translations. The mean time of the native server to handle a single request is 75.76*ms*.

				with vCache shuffle		
	Overheads	**DC hit**	**EC hit**	**Overheads**	**DC hit**	**EC hit**
EC,DC=2	1542.42%	98.66%	53.30%	1667.87%	98.66%	53.27%
EC,DC=4	1062.37%	99.73%	66.96%	1066.28%	99.72%	66.92%
EC,DC=8	657.81%	99.98%	80.95%	673.79%	99.96%	80.46%
EC,DC=16	222.12%	99.99%	95.59%	295.26%	99.99%	93.66%
EC,DC=32	48.02%	99.99%	99.89%	153.10%	99.99%	97.52%

size, so the performance of `Fourier` is decreased (0.81x comparing with the non-loop-alignment one).

Comparing with OBFSCURO. We also tested all the benchmarks used in [2]. Since the benchmark programs are very simple, we used 4 KB vCache size. The geometric average slowdown of KLOTSKI is 88.08% (over 1 billion executions), which is much lower than that of OBFSCURO (51×).

7.2 Real-World Applications

The first real application is an HTTPS sever `mbedTLS` [5], which is an open source *Transport Layer Security* (TLS) library. This project includes a sample HTTPS sever. The size of this HTML file is 30 KB in our test. To measure the performance, we used `ab` command to request the files from this HTTPs server 20,000 times and reported the mean of the total elapsed time to handle a single request. Table 5 displays the results. Overall, KLOTSKI imposed about 15.4× slowdown than the native one while the vCache size is set to 4 KB. The performance overhead reduced exponentially as the vCache size increased (from 15.4× to 48.02%). We also measured the larger vCache configurations with re-randomization, which increased the overhead to 153.10%.

Table 6. Performance overheads in running `cjpeg` and `djpeg` when process a single image in an enclave. The native `cjpeg` is 3.21*ms* and `djpeg` is 7.94*ms*. The frequency of vCache re-randomization is once per 1280 code translations.

			with vCache shuffle	
	djpeg	**cjpeg**	**djpeg**	**cjpeg**
EC,DC = 2	271.30%	552.19%	292.60%	550.20%
EC,DC = 4	171.91%	367.74%	200.48%	335.95%
EC,DC = 8	112.35%	238.13%	162.77%	235.73%
EC,DC = 16	77.61%	108.42%	126.92%	168.25%
EC,DC = 32	57.68%	67.42%	113.05%	137.30%

The next two applications are from `libjpeg` [25], which is a widely used C library for reading and writing JPEG image files and is attacked by Xu *et al.* [56]. `cjpeg` and `djpeg` are two examples in this project to show the processing of compressing a BMP or PPM file to a JPEG file and decompressing a JPEG file to another type of file. To measure the performance of `cjpeg`, we randomly downloaded ten PPM images with a fixed pair of width and height (227*149 pixels). The `cjpeg` enclave program compresses all these ten files to PPM files at one time and outputs the average time of processing an image. We ran the program for 100 times for each image and computed its mean. For the `djpeg`, we followed the same procedure for `cjpeg`, except that the inputs were ten JPEG images and its outputs were BMP file. Table 6 displays the performance overheads. When the vCache size is set to 4 KB, we see that KLOTSKI imposed 2.7× and 5.5× slowdown, respectively. Similar to the results of `nbench` and `mbedtls`, the performance overhead is reduced exponentially as the vCache size increases.

8 Conclusion

In this work, we present KLOTSKI, an efficient oblivious execution technique against controlled-channel attacks with tunable trade-off between security and performance. KLOTSKI essentially emulates a secure processor by performing the execution inside two in-memory vCaches in a page-access-oblivious way and leveraging ORAM protocol to load/evict content into/from the vCaches. We also designed and implemented several optimization techniques to reduce the overhead of KLOTSKI. Our security analysis shows that KLOTSKI is secure against controlled-channel attacks under our threat model. Performance evaluation over our prototype implementation shows that KLOTSKI has much lower performance overhead than previous solutions and our optimization techniques are very effective.

Acknowledgments

We would like to thank all the anonymous reviewers for their valuable comments and suggestions. The work is in part supported by the National Key Research and Development Plan of China under Grant No. 2017YFB0802205, by the Shenzhen Fundamental Research Program under Grant No. JCYJ20170413114215614, by the Guangdong Provincial

Science and Technology Plan Project under Grant No. 2017B0 10124001 and the Guangdong Provincial Key R&D Plan Project under Grant No. 2019B010139001, National Science Foundation under Award No. 1718997, Office of Naval Research under Award No. N000141712893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Shaizeen Aga and Satish Narayanasamy. 2019. InvisiPage: Oblivious Demand Paging for Secure Enclaves. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. New York, NY, USA.

[2] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[4] ARM. 2014. *ARM TrustZone.* https://developer.arm.com/ip-products/security-ip/trustzone

[5] ARMmbed. 2019. *Mbed TLS.* https://tls.mbed.org/

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *CoRR* abs/1709.09917 (2017). arXiv:1709.09917

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the USENIX Security Symposium (Security)*.

[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the USENIX Security Symposium (Security)*.

[13] BYTE Magazine. 2017. *Linux/unix nbench.* http://www.tux.org/~mayer/linux/bmark.html

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[15] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[16] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[17] Peter J. Denning. 1968. Thrashing: its causes and prevention. In *Proceedings of the American Federation of Information Processing Societies (AFIPS)*.

[18] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* (1996).

[19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security*.

[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of the USENIX Security Symposium (Security)*.

[21] James Greene. 2012. Intel trusted execution technology. *Intel Technology White Paper* (2012).

[22] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive* (2016).

[23] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[24] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] IJG. 2019. *libjpeg.* http://libjpeg.sourceforge.net/

[26] Intel. 2015. SGX Tutorial, ISCA 2015. http://sgxisca.weebly.com/.

[27] Intel. 2019. *Intel(R) Software Guard Extensions for Linux OS linux-sgx-driver.* https://github.com/01org/linux-sgx-driver

[28] Intel. 2019. *Intel(R) Software Guard Extensions for Linux OS SDK.* https://github.com/01org/linux-sgx

[29] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2017. Enhancing Security and Privacy of Tor's Ecosystem by using Trusted Execution Environments. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[30] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the USENIX Security Symposium (Security)*.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[32] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[33] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*.

[34] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems*.

[36] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the USENIX Security Symposium (Security)*.

[37] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the USENIX Security Symposium (Security)*.

[38] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *Proceedings of the USENIX Security Symposium (Security)*.

[39] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2017. Zero-Trace : Oblivious Memory Primitives from Intel SGX. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[40] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[41] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[42] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*.

[43] Jaebaek Seo, Byoungyoung Lee, Sungmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[44] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[45] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[46] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[47] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. 2017. A Compiler and Verifier for Page Access Oblivious Computation. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.

[48] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: enabling microarchitectural replay attacks. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[49] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[50] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[51] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[52] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[53] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[54] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[55] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[56] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.