Autarky: Closing controlled channels with self-paging enclaves

Meni Orenbach
Technion

Andrew Baumann
Microsoft Research

Mark Silberstein
Technion

Abstract

As the first widely-deployed secure enclave hardware, Intel SGX shows promise as a practical basis for confidential cloud computing. However, side channels remain SGX's greatest security weakness. In particular, the "controlled-channel attack" on enclave page faults exploits a longstanding *architectural* side channel and still lacks effective mitigation.

We propose *Autarky*: a set of minor, backward-compatible modifications to the SGX ISA that hide an enclave's page access trace from the host, and give the enclave full control over its page faults. A trusted library OS implements an enclave self-paging policy.

We prototype Autarky on current SGX hardware and the Graphene library OS, implementing three paging schemes: a fast software oblivious RAM system made practical by leveraging the proposed ISA, a novel *page cluster* abstraction for application-aware secure self-paging, and a rate-limiting paging mechanism for unmodified binaries. Overall, Autarky provides a comprehensive defense for controlled-channel attacks which supports efficient secure demand paging, and adds no overheads in page-fault free execution.

ACM Reference Format:
Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: Closing controlled channels with self-paging enclaves. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387541

1 Introduction

Enclave execution environments, and in particular Intel SGX [40], aim to make confidential cloud computing practical by removing trust from the cloud [6]. Major cloud providers have already deployed SGX [2, 34, 53], and another is developing a platform to support it [49].

However, side channels weaken the security of SGX. In this paper, we tackle the longstanding *controlled-channel attack* on enclave page tables [59, 67, 72, 76] that still lacks a general, practical mitigation. Controlled-channel attacks exploit the separation of concerns in SGX between enclave execution context, which is protected by the CPU, and resource management, which is delegated to the untrusted OS. In particular, the OS manages the enclave's address space and performs paging from and to its encrypted memory. Control over the enclave's page tables enables an OS-level adversary to trace the enclave's page access pattern in a noise-free manner by inducing page faults of her choice [76], or by monitoring page table access bits [67, 72]. As long as the enclave performs secret-dependent memory accesses to distinct pages, the attack can breach enclave confidentiality, extracting, for example, decompressed JPEG images or spell-checked text [72, 76].

SGX does not defend against controlled-channel attacks [28]; Intel's stance is that "preventing side-channel attacks is a matter for the enclave developer" [30]. However, it is not practical to avoid secret-dependent memory accesses for all but the simplest enclaves [59]. For example, the Opaque data analytics platform [78] requires an oblivious scratchpad memory, that SGX currently cannot provide. Moreover, existing software-only defenses [46, 58] suffer from significant practical limitations: they incur substantial performance overhead, prevent the use of demand paging and/or suffer from false positives in detecting the attack. Importantly, they require recompilation or even manual code changes, which limits their use in large enclaves running unmodified software [6, 50, 65]. On the other hand, proposed architectural defenses [1] require intrusive hardware modifications such as oblivious RAM-based paging. Thus, despite it being the earliest known SGX-specific side channel, the controlled-channel attack still poses a threat to practical enclave security.

Recent research revealed other side-channel attacks against SGX. These are primarily the consequence of sharing an internal CPU state across software trust domains [9, 44, 55, 72]. Coupled with speculative execution side channels (now mitigated by microcode updates and silicon fixes), these attacks enabled the extraction of attestation signing keys [11, 56], register values [70] and even full enclave memory [68]. While devastating, these *microarchitectural* attacks are highly sensitive to the (unpublished) properties of a specific CPU microarchitecture. Moreover, they are often noisy, as they exploit subtle timing fluctuations to infer the victim's access pattern to hardware resources shared with the adversary.

The controlled-channel attack is an *architectural* attack that does not suffer from these limitations: OS tracing of enclave page accesses is guaranteed by the Intel architecture specification [29]. Thus, the attack is noise-free, deterministic, and portable across hardware generations. Moreover, the

EuroSys '20, April 27–30, 2020, Heraklion, Greece

same mechanism helps remove the noise from microarchitectural attacks, dramatically increasing their precision [61]. The power of the controlled channel was unequivocally demonstrated in real-world attacks: SgxPectre, Foreshadow, RIDL and LVI all used it as a precursor [11, 68–70].

Devising a *practical* mitigation for the controlled-channel attack poses unique challenges. Grounded in the SGX memory management architecture, it can only be fully removed via architectural change. However, SGX's goal of compatibility with existing x86 software (the OS in particular), and its implementation within an existing microarchitecture [14] constrains the possible solutions. In particular, the backward compatibility requirement would almost certainly preclude hardware designs relying on separate enclave page tables [15, 17, 37]. On the other hand, software-only defenses restricting demand paging of enclave memory [46, 58] not only harm the usability of SGX, but still leak enclave accesses via page table accessed and dirty bits [67, 72].

We propose *Autarky*, a hardware/software co-design that takes a pragmatic and practical approach to closing page-fault side channels in SGX. The key principle is simple: *we revoke exclusive control of enclave page faults from the OS and enable trusted enclave software to enforce a secure paging policy.* As a result, the enclave can detect the occurrence of OS-induced page faults and block attacks, while permitting demand paging that complies with the enclave-enforced paging policy.

Autarky introduces minor backward-compatible modifications to SGX to enforce the new page fault handling flow, and designs a new secure paging mechanism in software.

Modified ISA for secure paging. We prevent the OS from silently resuming the enclave after a page fault, without first invoking a trusted in-enclave page fault handler. In addition, we stop leaks via page table accessed/dirty bits. These changes are non-intrusive in that they do not affect core hardware page-management mechanisms. Further, they are lightweight as they add only a few conditional checks to the *existing* SGX flows. We believe that these properties increase the chances that our proposal can be implemented in trusted firmware alone [14], facilitating adoption.

Self-paging runtime. Autarky's runtime manages an enclave's memory and handles its page faults thereby implementing *secure self-paging* [21, 32]. As we show in §6, the runtime is a good fit for SGX library OSes [6, 50, 60, 65] which enable in-enclave execution of unmodified binaries.

Secure paging policies. While Autarky's architectural changes remove an attacker's ability to target particular pages, leaks remain through legitimate enclave paging activity. To this end, the secure runtime may implement a range of secure paging schemes. We propose three such schemes demonstrating different security-vs-usability tradeoffs: *(i)* a software oblivious RAM (ORAM) providing provably secure general paging that builds on Autarky to speed up prior SGX ORAM systems [48] by orders of magnitude, yet requires recompilation; *(ii)* a new *page clusters* mechanism that provides strong security without ORAM overheads for enlightened applications; and *(iii)* a rate-limited demand paging mechanism that provides a weaker guarantee of bounded leakage (similar to prior work [46]) yet incurs minimum overhead and works with unmodified application binaries.

We evaluate Autarky's performance on the nbench [39], Phoenix [52] and PARSEC [7] benchmark suites observing zero overhead in the absence of paging activity, and better performance with rate-limited demand paging compared to that reported for software-only defenses [46]. Furthermore, Autarky successfully mitigates published attacks on known-vulnerable workloads with a small performance impact only under paging. Finally, we quantify the tradeoffs between page clusters and ORAM secure paging on paging-intensive workloads: Memcached [42] and uthash [22].

2 Background

2.1 Intel SGX

Intel SGX is an x86 architecture extension that supports isolated user-mode *enclaves*, protecting their confidentiality and integrity against all other software on the platform (including the OS, hypervisor, and other enclaves), and most hardware attacks. We summarize here aspects of SGX that relate to Autarky; full details are documented elsewhere [14, 29, 40, 41].

Overview. Enclaves are backed by a dedicated region of physical memory known as *enclave page cache* (EPC). Memory within this region may only be used by SGX enclaves and their associated metadata; it is encrypted and integrity-protected in the processor before it leaves the last-level cache. While the dedicated physical memory for enclaves is limited today (256 MB), it has already doubled in size and is expected to grow to be better suited for cloud computing [31].

Architecturally, an enclave occupies a contiguous region of virtual address space, the initial layout, and contents of which are guaranteed through remote attestation, and which is accessible only to software executing inside the enclave.

Enclave execution and state save areas. To run an enclave, software uses the EENTER instruction, which transfers control to a pre-defined (and attested) *entry point* address. Execution stays in the enclave until either control leaves it via the EEXIT instruction, or an exception occurs, such as a hardware interrupt or page fault. In the latter case, known as an *asynchronous enclave exit* (AEX), the processor saves the register context and the exception cause inside the enclave before replacing the context with a synthetic one, and finally leaving the enclave to invoke the untrusted OS exception handler.

After an asynchronous exit, the OS may re-enter the enclave via its entry point, or resume it using ERESUME which restores the context saved by AEX. The ability of a malicious

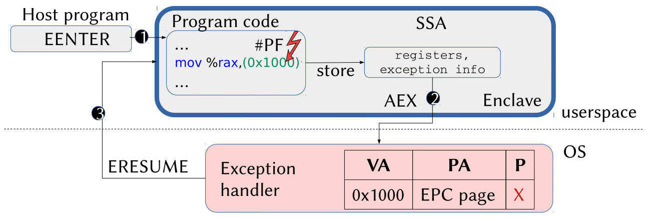

Figure 1. SGX page fault flow. ❶ The enclave runs. ❷ On a page fault (e.g., present bit clear), an AEX occurs, saving register context and exception details. ❸ After resolving the fault, ERESUME restores enclave context and replays the faulting instruction.

OS to transparently resolve an exception and resume the enclave, effectively hiding exceptions from in-enclave logic, is a key ingredient in the controlled-channel attack.

Each enclave contains at least one thread control structure (TCS), and each logical core entering an enclave must do so on an exclusive TCS. The TCS, in turn, points to a region of state save area (SSA) frames used to save enclave context and exception information on asynchronous exits, and from which state is resumed by ERESUME. This area is managed as a stack to allow re-entering the enclave on AEX; conceptually, AEX pushes a new SSA frame, and ERESUME pops it.

Memory management. Enclaves execute in a user-level process, and their address space is managed by the OS via the same page table. To protect enclaves from an untrusted page table, SGX consults an additional trusted *EPC map* (EPCM) structure, which is stored in secure memory inaccessible to software. After walking the page table, hardware uses the EPCM to check that enclave pages are mapped correctly, and that any changes are coordinated with the enclave code.

The EPCM is updated by SGX instructions. Prior to enclave launch, EADD populates an enclave's initial (attested) pages. After launch, an enclave's virtual memory can be modified dynamically (in SGX version 2); adding enclave pages or invalidating mappings requires the OS to coordinate changes with the enclave. Specifically, the OS uses instructions such as EAUG to add a page, EMODT to "trim" (deallocate) a page, and EMODPR to reduce permissions and, if necessary, performs a TLB shootdown. Then, for the change to take effect, trusted unprivileged code in the enclave confirms and commits the desired change using EACCEPT or EACCEPTCOPY.

Finally, SGX supports OS-driven demand paging, enabling enclaves to oversubscribe EPC. Since the OS cannot be trusted to directly swap EPC pages, the privileged EWB instruction evicts a page from EPC, storing it as encrypted data in regular memory, and its counterpart ELDU restores this data in the EPC. The instructions guarantee the integrity of the swapped out contents, and protect against replay attacks.

Access control and page faults. SGX implements its protection by modifying the processor's TLB miss handler [14]. The TLB is flushed when entering/exiting an enclave, and while executing in enclave mode, extra access control checks apply [29, §37.3]. When a TLB miss occurs in enclave mode, the processor walks the page table as usual. If it results in a valid page table entry (PTE) with sufficient permissions, extra SGX-specific checks occur. First, the enclave region (and only that region) may only map EPC pages. Second, the EPCM is consulted to check that the mapping is correct.

If any of these checks fail, a page fault and an AEX occur as shown in Figure 1. The faulting address and error information are saved in the SSA frame, and the page offset portion of the faulting address is zeroed prior to delivering the fault to the OS. Thus, the OS fault handler is given the page and the reason for the fault, but not the exact virtual address.

If all the checks pass, the TLB entry is installed and the translation proceeds. The PTE is also updated to set the accessed and dirty bits appropriately, as in regular paging.

The details above illustrate how the SGX protection mechanisms fit into the existing x86 architecture. In Autarky, we seek to change only the same paths as the current SGX implementation; this is a particular challenge for preventing leaks via accessed and dirty bits, as we will see in §5.1.

2.2 Controlled-channel attacks

Different variants of controlled-channel attacks have been described to leak enclave page accesses. The original work of Xu et al. [76] unmapped pages to trigger a page fault when they were accessed by the enclave before silently restoring the mapping. In the limit, this provides a page-granularity trace of every memory access by every enclave instruction.

At its core, a controlled channel consists of two components: first, a noise-free side-channel that leaks EPC page accesses, and, second, a mechanism for the attacker to stealthily control enclave execution, making the channel deterministic.

The first component is provided by one of the architectural mechanisms (§2.1) through which the OS may observe an enclave's page accesses. For example, the OS may invalidate the PTE (as in the original attack), reduce its permissions (e.g., making a code page non-executable) [74], or simply map the wrong page [68]. All trigger a page fault that the OS intercepts. Alternatively, it may clear the PTE's accessed or dirty bits, and observe when they are set [67, 72].

The second component, control, comes from SGX allowing the OS to resume (ERESUME) an enclave after an OS-injected page fault, which hides the attack from enclave software.

Since the attacker knows the program's expected behavior, she may use the access trace to recover secrets used in control- or data-dependent accesses. Xu et al. [76] demonstrated attacks that leaked images, text and characters processed by the libjpeg, Hunspell, and FreeType libraries respectively.

2.3 ORAM

Oblivious RAM [20] obfuscates memory accesses performed by a client to untrusted storage such that an attacker cannot learn any information about the actual accesses. Many ORAM algorithms have been proposed [62, 73, e.g.] that obfuscate program access patterns by dynamically re-shuffling and re-encrypting the accessed memory regions.

ORAM relies on a trusted memory region to store metadata corresponding to the un-shuffled addresses in the untrusted storage. For example, PathORAM [62] (a popular and efficient ORAM algorithm) arranges untrusted storage as a complete binary tree containing both real (valid client data) and dummy blocks. It maintains a *position map* that maps client data to blocks in the tree, and a *stash* that stores fetched blocks.

Recently, ORAM was proposed as a generic mechanism to mitigate controlled-channel attacks [48, 51, 54]. The key idea is to invoke ORAM for every normal memory access while also making oblivious accesses to the ORAM's own metadata. The latter is achieved by linearly scanning metadata with the CMOVZ instruction.

Unfortunately, ORAM typically incurs high performance overheads [51]. With Autarky, however, ORAM implementations in SGX may run orders of magnitude faster.

3 Threat Model and Out-of-scope Attacks

We consider a typical enclave threat model with a privileged adversary who controls the OS and/or hypervisor. The adversary manages the enclave's address space, and has full access to the page table. We assume that the enclave's code is public and free of vulnerabilities [74].

We disregard leaks via microarchitectural side channels, including cache timing and speculative execution attacks [9, 44, 55, 56, 68–70]. While these may leak enclave memory accesses (including PTE fetches [67, 72]), they are orthogonal to our work and we mitigate them to the extent possible by disabling hyperthreading and using the latest CPU microcode. We note, however, that the *microarchitectural replay attack* [61] relies on OS-induced page faults that Autarky prevents.

We consider *restart* attacks that require recreating an enclave to be out of scope, as there are known ways to defend against such attacks. For example, the enclave could perform remote attestation at startup [6, 50], or a local parent enclave (as in Graphene-SGX's multi-process mode [65]) could manage its children's lifecycle. In either case, users or trusted services could detect unusually frequent restarts.

4 Design Considerations

In this section we analyze the limitations of pure software or clean-slate hardware solutions, and discuss the design tradeoffs of a practical defense mechanism.

Software-only mitigations are limited. Defending against controlled-channel attacks on current SGX hardware is hard because the architecture exposes an enclave's page tables and page faults to the OS by design. Enclave software is left with three options: obfuscate the memory access trace [8, 59], prevent page faults altogether [58], or detect excessive enclave exits associated with page faults while disabling the sibling hyperthread [12, 46]. Unfortunately, all three suffer from high overheads. More crucially, since benign page faults are indistinguishable from an attack, these defenses curtail any fault-driven mechanisms, such as demand paging or lazy allocation. This hinders their applicability in real systems with limited EPC memory. Furthermore, all three require recompilation, preventing the use of existing binaries (supported by enclave libOSes [6, 50, 65]) or JIT compilers.

We conclude that a practical, general solution to these attacks will require changes to the SGX architecture.

Hardware solutions require intrusive changes. In clean-slate designs for hardware-based enclaves, controlled channels are avoided through separate enclave page tables [1, 15, 37]. Since the enclave has a private page table that the OS cannot observe or tamper with, the channel does not exist. However, retrofitting such a design to SGX would require substantial changes to the x86 implementation with which SGX is entangled. First, a new mechanism for collaborative management of encrypted pages between host OS and enclaves would be required. Second, new protections would be necessary to prevent an enclave mapping arbitrary host memory. Finally, performance-critical portions of the MMU would require substantial changes to support this design. Furthermore, private page tables may still leak when legitimate demand-paging occurs since the swapped-out pages are still under the OS's control and are therefore vulnerable.

Security vs. compatibility. When considering design options, we face an inherent tradeoff between backward compatibility and strong, general security.

We assert that any solution in which enclave page mappings are visible to the OS cannot provide strong ORAM-like guarantees for applications that use demand paging, unless it involves intrusive hardware changes. The reason is simple: if page mappings are visible to the OS, it can always determine which enclave pages are mapped. Given knowledge of application code, this leaks portions of the application's working set. We note that this *demand paging* side channel is strictly weaker than the controlled channel because the OS cannot trace arbitrary pages, yet it still compromises secrecy.

We see two solutions to hide page mappings, neither of which is fully satisfactory: *(i)* enclave-private page tables that are difficult to support in SGX as discussed above; and *(ii)* ORAM, which breaks the link between addresses and secrets, but either requires intrusive hardware changes [1] or else forces software recompilation and is slow [48]. With Autarky,

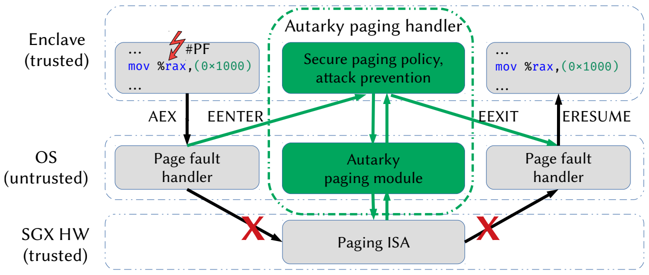

Figure 2. Autarky enforces invocation of an enclave's self-paging handler on each page fault.

we therefore favor a pragmatic solution that is compatible with the existing SGX design and empowers trusted software to make this tradeoff, at the expense of *either* application changes to achieve security and performance, *or* weakened (yet improved) security without any application changes.

Conservative approach to SGX architecture changes. We justify our approach of minimal architecture changes as follows. First, we hope that it will facilitate Autarky's rapid adoption. Given the slow evolution of SGX [5] and the urgency to eliminate controlled channels, time-to-deployment matters. Second, it minimizes assumptions about the SGX implementation, reducing our risk of overlooking broader side-effects of any changes. Last, it maintains the SGX design philosophy of embedding trusted execution in x86.

5 Design

SGX gives an OS control over enclave resource management while monitoring the OS's actions (e.g., page mappings) to ensure correctness. The problem is that the basic mechanisms of demand paging and fault handling leak an enclave's internal state without its control. Autarky revokes from the OS control of EPC paging policy, and delegates it to a *trusted self-paging* runtime. Figure 2 shows our modifications to the SGX page fault flow. Hardware now *enforces* invocation of the trusted page fault handler, and hides enclave page fault information and access/dirty bits from the OS.

Thus, Autarky establishes a protocol for cooperation between the OS and an enclave runtime, much like SGX does to protect against direct attacks, but instead to close controlled channels. To enable this new separation of concerns, Autarky relies on a trusted runtime layer, such as an SDK [27, 43, 49] or library OS [4, 6, 50, 65, 65], which manages an enclave's memory and handles its page faults.

5.1 SGX architecture modifications

5.1.1 New enclave attribute

Following the existing scheme for optional features, we define a new enclave attribute bit for a self-paging enclave. This bit is an *attested* flag for an enclave to enable all the changes described below. If the OS or CPU does not support it, the attribute mechanism allows the enclave to either continue in legacy (insecure) mode, or fail to start.

5.1.2 Closing the channel: hiding faults from the OS

We propose to extend the original SGX exception handling mechanism as follows. For any enclave-mode page-fault occurring inside the enclave region, the processor already saves full exception information in the SSA frame and masks the page offset visible to the host. Our modified mechanism hides the entire address and access type in the page-fault error code before delivering it to the OS's exception handler. To avoid ambiguity, all enclave faults should be reported at some consistent address within the enclave region; for example, as a read fault at the enclave's base address. In this way, the OS learns only that some enclave fault has occurred.

5.1.3 Removing attacker control: reporting all faults

This change prevents the OS from silently resuming an enclave after a fault. We extend the per-thread TCS with a new *pending exception flag* and modify the AEX procedure so that on any page fault, the processor sets the pending exception flag. We also modify EENTER to clear the flag on entry, and ERESUME to fail if the flag is set. Thus, the OS is forced to re-enter the enclave after an exception, at which point the trusted runtime can reliably determine that an exception occurred (using the SSA frame), and run the handler.[1]

This, in turn, allows the enclave's exception handler to detect *unexpected page faults*, to which it may apply a policy. For example, it may simply ask the OS to restore the faulting page and continue execution, perhaps applying a heuristic to detect excessive fault rates. Alternatively, it may implement secure self-paging, and treat any unexpected page fault as an attack, in which case it would terminate to prevent leaks. We describe a range of possible software designs in §5.2.

Eliding AEX. We note that this approach adds overhead to benign exceptions. For every page fault, the CPU saves the full context, exits the enclave, and invokes the OS handler, which can do nothing useful since it lacks knowledge of the fault, but must return to user-mode and re-enter the enclave. Therefore, as an optimization, we propose staying in enclave mode: after saving the exception information in the SSA, the hardware would immediately simulate a nested enclave re-entry by incrementing the SSA and jumping to the enclave entry point, whereupon the exception handler can run. This optimization appears practical and elides costly enclave transitions [4, 47], which besides their high direct cost also flush TLB, and L1 caches—previous work showed the latency of invoking an enclave exception handler is more than 6× that of a signal handler [48]. We evaluate performance both with and without this optimization, in §7.

[1]Like an OS, the enclave runtime must take care to avoid nested faults that would exhaust the SSA stack and render the enclave un-executable.

Resuming from exceptions. Another source of exception overhead arises when resuming execution. As previously observed [6], the handler must EEXIT to a stub that merely ERESUMEs the original SSA frame, incurring another costly enclave transition for the sole purpose of restoring context and popping the SSA stack. This could be avoided with an in-enclave variant of ERESUME, but we do not assume it.

5.1.4 Blocking the use of accessed and dirty bits

With the changes described above, we have curtailed the OS's ability to inject or observe enclave page faults. However, it may still infer access patterns via PTE accessed and dirty bits. This is an area where changing hardware carries the most risk since this functionality is performance-sensitive, and we cannot afford to degrade the use of these bits for non-enclave PTEs (or indeed, for non-self-paging enclaves, since the legacy paging mechanism relies on it).

A straightforward fix would be to prevent updates of accessed and dirty bits in enclave PTEs. However, this may prove impractical to implement: it requires that core MMU paths be modified with SGX-specific logic. For example, the page table walk would need to skip updating access bits for addresses within the range of a self-paging enclave, and TLB entries would need to be flagged as holding enclave translations to avoid writing back dirty bits.

To simplify the hardware, we therefore propose to add a check for fetched PTEs, only in enclave mode for a self-paging enclave, and only after fetching a PTE in the enclave region (i.e., in conjunction with the EPCM lookup, which is clearly already SGX-specific): the accessed and dirty bits for the fetched PTE must already be set; if either is clear, the PTE is treated as invalid, and a fault occurs. This blocks the OS use of accessed and dirty bits for self-paging enclaves, and allows the enclave to detect such leaks in its fault handler.

Our proposal assumes that (a) the fetched PTE's accessed and dirty bit state is available at the time of the SGX-specific checks, and (b) the MMU will never perform writeback of accessed/dirty bits if they were already set at the time of the TLB miss. The latter assumption, which is consistent with, but not guaranteed by, the specification [29, §4.8] is necessary to prevent a time-of-check to time-of-use attack, whereby the accessed or dirty bit is set at the time of the TLB fetch yet later cleared, effectively defeating the defense.

This change may degrade eviction performance for self-paging enclaves, because access/dirty bits cannot be used to guide eviction policy (e.g., to implement the common "clock algorithm"). However, since (as we will see in §5.2) eviction policy will be devolved from the OS to the enclave runtime, any impact on core OS paging decisions is minimal. One option for the enclave runtime is to use a more coarse-grain frequency-based algorithm that counts the frequency of page faults for each page, and eventually learns to keep "hot" pages paged in (similarly to Linux NUMA page migration).

Summary. The combined effect of our architecture changes is that the OS cannot evict EPC pages nor observe accesses without enclave cooperation. More precisely, it can still unmap pages or even evict them using the EWB instruction, but if the enclave ever attempts to access such a page, it will detect that it was missing. As a result, the OS is reduced either to swapping in/out entire enclaves or managing an enclave's memory with its consent and cooperation.

5.2 Software design

The SGX changes described in §5.1 suffice to block the attack by simply keeping an entire enclave resident in EPC. Thus, any fault is regarded as an attack, terminating execution. Prior software mitigations for the controlled-channel attack [46, 58] take essentially this approach. However, it leads to severe practical limitations: first, memory requirements for even one enclave may easily exceed EPC, rendering such enclaves un-runnable. Second, dynamic memory allocation would be impractical even for smaller enclaves: in the presence of multiple dynamically-sized enclaves, the OS would be unable to balance competing memory demands. Instead, it could merely swap entire enclaves, which is inefficient.

On the other hand, reimplementing demand paging inside an enclave runtime risks information leaks to the OS via the demand paging side channel (discussed earlier in §4). If the enclave runtime responds to each page fault by mapping the accessed page, the OS may easily infer it. Likewise, when the enclave evicts a page, the OS may infer access information. In fact, the system's overall security depends on the eviction and fetching *policy* implemented by the self-paging runtime.

In this section, we describe three alternative self-paging policies that minimize such leaks: an efficient software-based oblivious RAM system (§5.2.2), a new *page clusters* abstraction (§5.2.3) that provides a meaningful security guarantee to applications modified to make use of it, and finally a rate-limited demand paging mechanism (§5.2.4) that provides a weaker bounded leakage guarantee for unmodified application binaries. We begin by describing the OS interface on which all three depend to enable enclave page management.

5.2.1 OS interface

We need a flexible mechanism to balance the number of EPC pages available to each enclave, that adjusts to the available EPC and memory pressure from other enclaves. This is the role of the interface between the enclave runtime and the OS: restricting the use of OS-level paging on sensitive pages, and supporting self-paging of those pages by the enclave.

Enclave-managed pages. We take a two-level approach to enclave page management. We partition the set of EPC pages used by an enclave into those managed by the OS, and those managed by the enclave. For pages that cannot be exploited to mount the controlled-channel attack (e.g., a buffer to which the access pattern is independent of secrets, as in a matrix product), the existing OS-level paging mechanism is more flexible because it permits the OS to evict and fetch pages at any time without enclave interaction.

Other pages with potentially-sensitive access patterns cannot be silently evicted by the OS. For these *enclave-managed pages*, the trusted runtime tracks the residence status of each page and treats any unexpected fault on a purportedly-resident page as an attack to which it responds by terminating the enclave. The sensitivity of a page may change over the lifetime of the enclave, so the OS must be aware of which pages it manages. To do so, we add two system calls: ay_set_os_managed yields management of a set of pages to the OS and ay_set_enclave_managed claims it for the enclave. The latter call also returns the current residence status of each page (i.e., whether it is currently paged out) so that the enclave can update its state and initiate page-in if desired.

From the OS perspective, the contract on enclave-managed pages is as follows: each resident enclave-managed page is effectively pinned in EPC whenever the enclave is runnable. EPC is a limited resource, and the OS may enforce a limit on its use to prevent one enclave from monopolizing EPC.

If the OS wishes to reclaim memory dynamically, it has three options: it may evict any OS-managed page at will, it can upcall the enclave and ask it to reduce its memory use, or it can swap out the enclave. The first option is straightforward; we discuss the latter two in detail here.

Similar to memory ballooning in virtual machines [71], memory management upcalls from OS to enclave imply a series of difficult tradeoffs. First, the enclave must be given time to reduce its memory allocation. Second, the enclave runtime must take care that its eviction policy does not leak sensitive information. Third, the enclave may not cooperate; for example, it may refuse to evict sensitive pages to prevent leaks. For these reasons, we did not pursue such upcalls, deferring their investigation to future work. Their absence leaves the OS with one final option: evicting enclave-managed pages. To respect the contract, it can only do so by suspending the enclave, at which point it can evict all enclave pages (i.e., swap the entire enclave out), but it must first restore them all before resuming enclave execution.

Supporting self-paging. As described above, the enclave may decide to evict or fetch one or more enclave-managed pages at any point in its execution including, but not limited to, the page fault handler. This process involves modifying the page table and allocating/freeing EPC pages, so it necessitates new system calls: ay_fetch_pages to securely bring pages into EPC from a backing store, i.e., untrusted memory, and ay_evict_pages to securely write pages out to the backing store. The calls explicitly support batching to minimize system calls and enclave crossing overhead: each takes an array of page base addresses to fetch/evict. Note that the

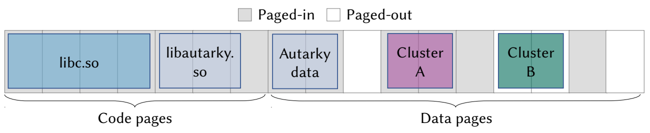

Figure 3. Sample enclave with page clusters shown as different colors. Pages in a cluster are fetched and evicted together.

enclave runtime only manages pages using their virtual address; the mapping of virtual to physical EPC frames is under the control of the OS, and invisible to the enclave.

To prevent the OS from interfering with page contents, we rely on existing SGX mechanisms: either the privileged EWB and ELDU paging instructions, or the dynamic memory management instructions of SGXv2 (both described in §2). The latter are more flexible, permitting enclave software to implement custom encryption, avoid writeback of clean pages, or use an alternative backing store, however, they incur an extra enclave crossing. We evaluate both approaches in §7.

5.2.2 Efficient software ORAM

Recall from §2.3 that oblivious RAM [20, 62, 73] provides an effective and secure way to obfuscate enclave paging activity, albeit with substantial performance cost [1, 48, 54]. However, using ORAM for an enclave's demand-paging backing store is not sufficient to hide the access pattern: the adversary can observe changes in page mappings made by the fault handler, thereby learning the fetched (and hence, accessed) pages.

Instead, CoSMIX [48] proposed instrumenting all memory accesses to use ORAM, re-shuffling the *page contents* regardless of the underlying mappings. Furthermore, CoSMIX allows selective annotation of static variables or/and memory allocations and automatically instruments all the corresponding accesses to use ORAM. Unfortunately, even selective instrumentation still incurs high overheads.

Since Autarky preserves confidentiality of the address trace for enclave-managed pages, it allows caching of recently accessed ORAM pages in a large pre-allocated buffer acting as an *enclave-managed software cache* without leaking the access pattern. Essentially, memory accesses are instrumented to perform a cache lookup and invoke the costly ORAM protocol only in the case of a cache miss. Effectively, Autarky reduces overheads to the extent that ORAM-based paging becomes practical (§7.2). Note, fetching (evicting) page contents to (from) the cache is an *oblivious copy* operation. The cache is backed by enclave-managed pages that can be pinned and therefore eliminate page fault leaks.

5.2.3 Page clusters

ORAM obfuscates accesses to all enclave pages. However, not all applications require such a strong guarantee. For example, there is no reason to hide sequential accesses, such as parsing of an encrypted stream. Even for more complex

Table 1. Page clusters API

Function	Purpose
ay_init_clusters(n, s)	Initialize *n* clusters of size *s*
ay_release_clusters()	Release all resources
ay_add_page(cluster, page)	Registers page with cluster
ay_remove_page(cluster, page)	De-registers page from cluster
ay_get_cluster_ids(page)	Returns all clusters containing page

data structures such as a hash table, it may be necessary to hide *which* hash bucket was accessed, but not *whether* the table was accessed at all. Allowing developers to express such requirements in terms of application-level resources is the goal of a new abstraction we call *page clusters*, depicted in Figure 3.

The cluster API is presented in Table 1. It may be used by a system runtime or by the programs as we explain below.

A page cluster is a consistent set of enclave-managed pages that are evicted and fetched together. Whenever a fault occurs, the system deterministically fetches and maps all the other pages in a cluster together with the faulting page. Thus, the attacker cannot differentiate which of the fetched pages caused the fault, even if the same fault occurs many times. We note that clusters differ from large pages, since they need not be physically or virtually contiguous, and can be assembled or broken down dynamically.

Formally, the system maintains the following invariant: *for each non-resident page, there is at least one cluster to which it belongs with all of its pages non-resident.* This invariant is trivial to enforce if all the clusters are disjoint. However, pages can be shared between multiple clusters, which is useful in particular for code pages, as we explain below. Importantly, when swapping in such clusters, it is crucial to fetch the transitive set of all clusters sharing pages with the faulting cluster and among themselves. Otherwise, there could exist a situation where all but one page of a cluster are resident in memory, as they may have been fetched previously due to sharing. A subsequent fault on this single non-resident page would uniquely reveal its access.

However, evicting a single cluster that shares pages with others is safe. The intuition is that either all the pages of the evicted cluster remain evicted (invariant holds) or they become resident as part of a fault on any of the pages of any of the other clusters with which they are shared.

The security guarantees of clusters depend on their construction and the program's threat model (§5.3). Intuitively, for programs with a uniform access pattern the larger the cluster, the lower the probability an attacker may infer which page was fetched. However, when the access distribution is skewed, even huge clusters might leak information. In such cases, the less performant ORAM alternative remains.

Clusters for code pages. Common attacks on code pages infer secrets by observing control flow as revealed by instruction fetches. For example, Xu et al.'s [76] attack on the FreeType library infers text being rendered from the unique pattern of code pages executed to render each character. To prevent this kind of attack, one can place all the code pages of a library in a single cluster, ensuring that control flow through the library's internal code does not leak. Note that if two libraries use a third, their respective clusters will share pages and will also be fetched together.

Clusters for libraries and the main program can be created automatically by a libOS, utilizing a trusted loader. A loader may also create clusters at the finer granularity of individual functions for better paging performance, if control flow between functions is not considered sensitive. In our experience, libraries used by enclave applications are substantially smaller than total memory; thus, their code pages can be clustered and kept resident, or paged with low overhead.

Manual clustering for data pages. Clusters can be a powerful tool for developers to defend against paging side channels with low overhead. A user may manually construct clusters with knowledge of application semantics. For example, consider an application with multiple hash tables, each of which fits in memory, but that exceed it when combined. A user may define a cluster for all of a hash table's pages. Then, upon access, all the pages in the cluster would be fetched and an attacker would learn only that the hash table was accessed. We demonstrate further examples in §7.3.

Automatic clustering for data pages. Data pages are harder than code to cluster automatically since their access semantics depends on the program. We propose an automatic policy that eagerly fills clusters with allocated pages by extending the libOS page allocator. A user specifies the desired size of data clusters. Each allocated page is added to a cluster, up to the maximum size, at which time a new cluster is created. When enough pages are freed, the libOS allocator merges clusters to keep them near-full.

Consider for example, a hash table with internal node chaining for resolving collisions. An attacker may infer which entries are accessed based on their unique page access signature, as in the Hunspell attack [76]. Obviously, smaller entries are more secure since they occupy the same page, and larger entries that share fewer pages leak more. Clustering can mitigate such attacks; specifically, the more pages in a cluster, the lower the probability that an attacker infers the accessed entry. However, larger clusters impact performance when paging—we evaluate this tradeoff in §7.

5.2.4 Bounded leakage for unmodified applications

One of our motivations for Autarky is to support unmodified application binaries using a library OS [6, 65]. However, both ORAM and page clusters require application changes and/or recompilation. We describe here a scheme that works with *completely unmodified application binaries* whose memory allocations *exceed available EPC* (thus requiring paging).

To achieve this goal without application knowledge, we must accept some leakage when we handle page faults and map pages. We use a combination of three techniques: (i) *automatic clusters for code pages* (as described in §5.2.3) prevent leaks of control flow by fetching all code pages of a library as one; (ii) *enclave-managed data pages* use traditional demand paging inside the enclave; and (iii) an optional, application-specific *bound on the maximum permitted page fault rate.* The combination of these techniques allows us to provide the following guarantees to applications: given a reasonable minimum EPC size (which may be configured by the user and checked at enclave startup), the only page accesses that leak are those to data pages that would have triggered demand-paging (i.e., cold pages), and to reduce the risk of an active attacker, the enclave will terminate if the rate of legitimate page faults exceeds a user-defined threshold.

Ideally, the maximum page-fault rate would be expressed in terms of enclave runtime, such as CPU cycles. Unfortunately, while the enclave can easily count page faults, it lacks a reliable time source: the cycle counter is untrusted, and the real time clock provided by SGX platform services is too slow to query in a fault handler. Instead, we are restricted to counting application-specific measures of forward progress observed by the libOS, such as I/O, memory allocations, and system calls. For example, a server application may limit page faults per socket receive call, while a machine learning task may express its limit in faults per memory allocation.

We note that despite providing only a weak bound on fault rates, this scheme is substantially better than the similar software mitigations [46, 58] with fault rate limiting. Not only is the performance overhead lower (thanks to Autarky hardware), but the only enclave accesses that leak are via the demand-paging side channel (legitimate paging). This channel leaks less than the silent attack on page tables [67, 72], which is not fully eliminated by any known mitigations.

5.3 Security analysis

Within the assumed threat model (§3), Autarky guarantees that the attacker can infer only accesses to enclave-managed pages at the time they are fetched or evicted by the fault handler, in which case the nature of any leak depends on the self-paging policy. For ORAM, there is no leak; for page clusters, the faulting page is indistinguishable from others in the same cluster; for the bounded leakage policy, accesses to data pages may leak below an application-specific rate limit. We discuss some potential attacks within these constraints.

Termination/lack-of-faults attacks. The attacker may attempt to unmap one or more enclave-managed pages (or clear their accessed/dirty bits, which has the same effect). If later accessed by the enclave, this will cause a spurious page fault which will be detected by the enclave's fault handler, resulting in enclave termination. However, the attacker learns something as a result, leading to what we term the *termination* attack, and the *lack-of-faults* attack.

We assume that when the runtime detects an OS-induced fault on an enclave-managed page it will terminate. Hence, the OS learns that a page it unmapped was accessed (because it knows that an exception occurred), but not which specific page of those it unmapped triggered the fault.

Conversely, if the enclave does not terminate when pages are unmapped, the attacker infers that they were *never* accessed. Assuming application knowledge, this lack of faults may be used to infer that a complement set of pages was indeed accessed. The bandwidth of this attack is comparable to the termination attack, so we do not see it as significant.

The attacks result in (or run a risk of) enclave termination, requiring that the enclave be restarted. As described in §3, we assume that such restarts can be detected by a trusted party through the use of attestation, and so disregard the potential accumulation of page traces across repeated runs.

Leakage via legitimate page faults. To mitigate leakage of page accesses when enclaves exceed their available EPC, users may construct clusters. Clusters fetch all their pages in to memory such that an attacker knows that one of the pages in the cluster was accessed, but not which one. This means that page access leakage correlates with the cluster size. Furthermore, clusters may eliminate page faults if they prefetch pages that may be accessed in the future. If clusters provide insufficient guarantees, a user may defer to ORAM.

Invoking the enclave's exception handler. An attacker cannot arbitrarily invoke the exception handler. It is called only by the trusted enclave code with fault information from the SSA. Nested faults can be avoided by pinning all the handler's code and data pages in enclave-managed memory. Thus, any re-entrancy can be regarded as an attack, and we provision sufficient SSA stack to permit detection thereof.

5.4 Discussion: VM support

In virtualized environments, both the guest OS and hypervisor may observe enclave page faults and mount controlled-channel attacks. Autarky mitigates the VM-level attack, which has implications for hypervisor implementations.

SGX [10] allows a hypervisor to virtualize enclave memory through static partitioning, ballooning [71] or demand paging [77]. Autarky supports static partitioning similarly to a bare-metal OS hosting multiple enclaves. Notably, cloud platforms that statically partition EPC [53] will require no modification. Ballooning can also be supported with minor changes: an enlightened guest OS enables cooperative paging, which allows a hypervisor, guest OS and enclaves to invoke secure self-paging polices (§5.2.1). We defer the details of such a mechanism to future work.

However, transparent demand paging by the hypervisor cannot be supported, since Autarky prevents the VM from

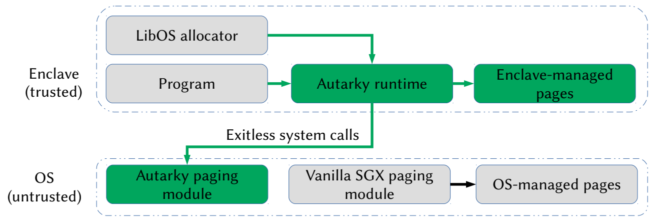

Figure 4. Autarky's prototype implementation.

observing fault addresses. Enlightenments are required to inform the hypervisor of enclave-managed pages, and mediate handling of faults on OS-managed pages. Recent extensions to the SGX ISA [10] already give a hypervisor knowledge of OS details such as the EPC pages belonging to specific enclaves. While cooperative paging will requires further hypervisor changes, it is ultimately more flexible, and requires simpler hardware support, than the existing design.

6 Implementation

We prototype Autarky's software components by modifying the Graphene-SGX libOS [65] (4,470 LoC modified out of 1.85 M total) and Intel SGX driver [26] (913/3,077 LoC). Figure 4 shows an overview. Since we lack hardware that includes our architecture changes (§5.1), our prototype remains vulnerable to the controlled-channel attack and serves only to estimate the performance and usability of our design.

We support two paging mechanisms: the SGXv1 privileged instructions (EWB and ELDU), and the dynamic memory management instructions of SGXv2. Both uses exitless host calls to reduce the cost of enclave transitions [4, 47, 75].

When fetching a page with SGXv2, we add a mapping using the EAUG and EACCEPTCOPY instructions. We use AES-NI and Intel's IPP crypto library [25] to decrypt and validate page contents. Moreover, to improve performance, we overlap EAUG with decryption using a temporary buffer. To evict a page with SGXv2, we first set it to read-only with EMODPR and EACCEPT. Then we can safely encrypt, sign and write it while maintaining thread-safety. We finalize the eviction process with the EMODT, EACCEPT and EREMOVE.

We modify the SGX driver to implement Autarky's system calls (as IOCTLs) to prevent the eviction of enclave-managed pages, and to avoid the use of accessed and dirty bits in the eviction algorithm for OS-managed pages.

Our runtime extends Graphene: the enclave's exception handler is called to resolve each page fault, by verifying that it is not malicious, and (if needed) evicting pages to free EPC prior to fetching pages to satisfy security guarantees (e.g., those sharing a cluster). We implemented new Graphene APIs to allow applications to define page clusters. Finally, we extended Graphene to support exitless calls to reduce the performance overheads of enclave transitions.

Our ORAM prototype is based on the CoSMIX PathORAM memory store [48] (1,960 LoC). At a high level, page contents are stored in a dedicated section of enclave memory. The CoSMIX compiler instruments application memory accesses using the PathORAM construction. Since Autarky protects all enclave-managed pages, we implement a new instrumentation policy that utilizes a large enclave-managed buffer as a cache for page contents. Our instrumentation first checks whether the page exists in the cache, and if so, accesses the memory directly. Otherwise, it obliviously fetches (and evicts if needed) to/from the cache. To reduce thrashing, we store page contents securely (encrypted and signed) in untrusted memory, validating them when they are fetched to the cache. Finally, we avoid costly linear scans used by CoSMIX to hide access patterns to PathORAM's data structures (position map and stash) by marking their pages as enclave-managed.

7 Evaluation

Setup. We use a Dell XPS 13 2-in-1 laptop with 4-core Intel i7-1065G7 (Ice Lake) CPU, 16 GB RAM, and 256 MB enclave reserved memory (≈190 MB EPC) running Ubuntu 19.04 (64-bit), Linux kernel 5.0.0, and CPU microcode version 2E.

The Autarky runtime in Graphene-SGX automatically marks pages for program code, stack, and self-paging metadata as enclave-managed (pinned in EPC). Results do not include initialization. Each run pre-loads the same set of pages to enclave memory. We report the mean of 10 runs; standard deviation is below 5%. The baseline uses a clock page eviction policy in the SGX driver, Autarky uses FIFO eviction since page access bits are not available.

Overhead from SGX architecture changes. Our architecture changes (§5.1) do not influence normal operation except for: *(i)* on TLB fill, to validate that access/dirty bits are set, and *(ii)* on AEX, EENTER and ERESUME to set, clear and check the pending exception flag respectively.

TLB fill reads the entire PTE including access/dirty bits, so the only overhead arises from the check itself, and depends on the number of fills. To measure this, we run the nbench [39] benchmark suite also used to evaluate the T-SGX defense [58]. Its datasets fit in EPC (no paging).

Pessimistically assuming a 10-cycle overhead on each fill, the geometric mean slowdown is 0.07% across all 10 benchmark applications. This analysis ignores the potential performance benefit of eliding the accessed/dirty bit writeback. By comparison, T-SGX reports a 1.5× mean slowdown.

Pending exception flag accesses are also cheap. Both EENTER and ERESUME already read the TCS flags word, and although AEX does not use the flags, it updates a field on the same cache line [29]. Therefore, the *only* change in memory access is a dirtied cache line that EENTER previously read. Since AEX and ERESUME already modify that line, the impact is likely negligible.

To summarize, the overheads imposed by our architecture changes are insignificant, and we disregard them in the rest

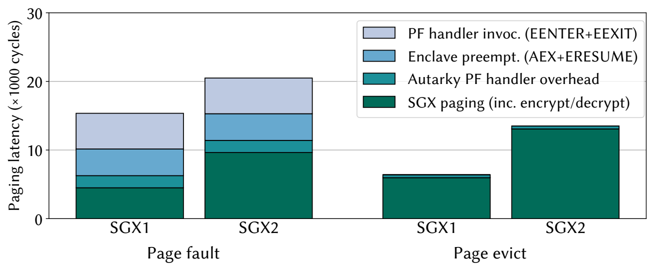

Figure 5. Paging performance using SGXv1/v2 instructions

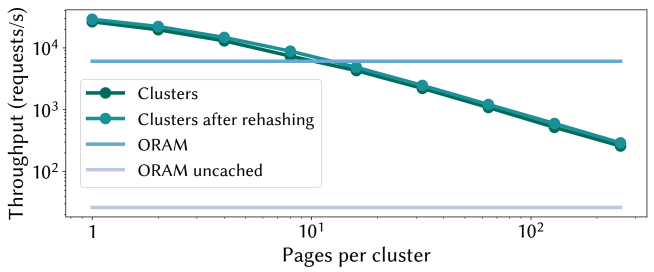

Figure 6. Effect of cluster size on hash table performance

of this evaluation. In particular, we expect Autarky to add *no measurable overhead to page fault-free execution.*

7.1 Microbenchmarks

We measure the latency of page fault (fetch) and eviction, averaged over 100k iterations. We evaluate both SGXv1 (driver-based) and SGXv2 (in-enclave) implementations of these operations (§6). Since the Intel driver evicts batches of 16 pages to reduce overheads, we use the same batch size in both versions, and normalize the latency to a single page.

Figure 5 shows the results. We break them down as page fault preemption (AEX+ERESUME), enclave fault handler (EENTER+EEXIT), Autarky runtime overheads, and aggregate cost of SGX paging instructions including en/decryption.

Enclave preemption and fault handler invocation accounts for 40–50% of the latency. However, the more intrusive architecture optimization (§5.1) to elide the AEX entirely can eliminate these costs. Such a change would make Autarky secure paging faster than today's unprotected paging.

Since SGXv1 paging instructions are more efficient for our purposes, we use them in the rest of the evaluation.

7.2 Secure self-paging policies

ORAM. We use our *cached* ORAM described in §6 with a 128 MB ORAM page cache (the largest possible without exceeding EPC). We mark this cache, ORAM data structures, and code pages as enclave-managed to prevent leaks. We configure PathORAM's tree to cover a 1 GB range, large enough to obliviously fetch page contents to the cache. As a workload we run uthash [22], a popular hash table with chaining collision resolution, configured with up to 10 items per bucket, filled with 431 MB of data and 256-byte items. We also compare ORAM with page clusters (next section), whose security is strictly weaker. Figure 6 shows that cached ORAM and clustering break even at 10 pages per cluster.

Uncached ORAM. Cached ORAM cannot be implemented without Autarky, because an OS adversary would observe EPC accesses. To compare with the cached version, we run the same experiment without the ORAM cache in EPC, using a linear scan to access ORAM structures.

The same 431 MB input did not complete in 24 hours. Instead, we performed the experiment using 100 random entries without changing either the hash table or the PathORAM tree size. This is the best-case scenario, because contention is unlikely. Nevertheless, Figure 6 shows a 232× slowdown.

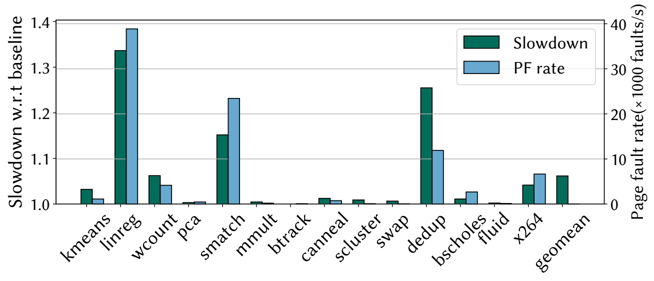

Figure 7. Rate-limited paging for Phoenix and PARSEC

Page clusters. We evaluate automatic protection of large hash tables. Here we analyze the security-vs-performance tradeoff for *application-agnostic* mitigation by modifying the libOS's memory allocator. This tradeoff is different for application-aware protection evaluated later (§7.3). We use uthash with the same input as in the ORAM experiment.

Autarky partitions memory into fixed-sized clusters by starting a new cluster when the current one is full. We vary the cluster size. For uniformly random accesses, the probability of an attacker guessing the accessed item given a cluster size is $\frac{item\_size}{cluster\_size \times page\_size}$, or 0.62% for 10 pages.

The experiment is as follows. We populate the hash table, measure random reads (loads are excluded), trigger rehashing and bucket expansion, and measure performance again. Since rehashing shortens the bucket chains, the number of clusters fetched per lookup is reduced as the nodes in the chain likely belong to different clusters.

Figure 6 shows the results. As expected, the cluster size is inversely proportional to performance; after rehashing the performance improves by about 1.5×. An unprotected baseline is 1.9× faster than 1-page clusters (not shown), which matches the microbenchmark performance.

Rate-limited paging. This demand-paging policy enforces a limit on the overall fault rate. This is our least secure policy, yet it provides similar guarantees to Varys [46], which enforces a limit on the rate of asynchronous enclave exits (caused, in part, by page faults).

We run 14 out of 15 applications in the same Phoenix [52] and PARSEC [7] benchmark suites used for Varys (*vips* does not run in Graphene). To induce page faults, we reduce the EPC space to about 100 MB for better comparison with Varys, and fine-tune the limit accordingly to prevent false positives. We use larger inputs than Varys only for stringmatch, wordcount and blackscholes to induce demand paging. We cannot compare results directly due to a different libOS and hardware, but note that Oleksenko et al. [46] reported a 15% overhead. Unlike Varys, Autarky does not require recompilation.

Figure 7 shows that this paging policy introduces a 6% slowdown on average. As expected, the page fault rate (right Y-axis) correlates with the slowdown. Eliding AEXs via more intrusive hardware changes would reduce overheads to 2%.

7.3 Protecting real applications

We evaluate Autarky using applications that were shown to be vulnerable to controlled-channel attacks [76]. We use these workloads to evaluate self-paging performance, both by enlightening the applications to use page clusters, and using automatic clustering in Graphene's memory allocator. The published attacks make use of access patterns to code and data pages. We show how Autarky can efficiently mitigate these attacks in several usage scenarios. Finally, we evaluate Memcached [42] with ORAM compared to an insecure baseline, showing that end-to-end overheads are acceptable.

We begin with three applications that were shown to be vulnerable to the attack, demonstrating the simplicity and the efficiency of Autarky. We report performance in Table 2, showing the full execution time (Autarky) as measured, as well as the potential improvement as a result of the proposed optimization to elide AEX and upcalls.

Image processing with libjpeg. Libjpeg [24] is a library to decode and encode JPEG images. The published attack focused on the inverse discrete cosine transform, which uses an optimization to elide needless state updates, making the page access pattern dependent on the image. The attack counts the number of pages accessed (by inducing page faults) and was shown to reconstruct the image being decoded.

Libjpeg streams over the input image to decode or encode it while operating on a temporary buffer. Therefore, the working set size depends on the buffer's size and not the image's, hence the intermediate state does not exceed the EPC. Consequently, this attack can be *automatically* protected against using Autarky, simply by marking all pages as enclave-managed. As the working set fits into EPC, the runtime pins all the pages, and no information leaks.

Allowing OS paging for insensitive pages. Libjpeg is often used in the first/last stages of image processing pipelines, where the image is first fully decoded. Therefore, the effective application memory footprint might exceed the EPC size, rendering automatic protection unsuitable.

However, if the later pipeline stages access the image in a data-independent way, e.g., when applying a filter, then its buffer can be considered *non-sensitive*, allowing OS paging. To differentiate between the protected temporary buffers in libjpeg and unprotected buffers used by the application, we modified libjpeg to call `ay_add_page` after each `malloc`.

We developed a simple test program that uses libjpeg to decode an image, inverse its colors and encode it again. We use a large image (13632×10224) that exceeds EPC in decoded form (398 MB). Table 2 shows the end-to-end performance. The main difference between the unprotected and protected versions is that page faults in the latter are reported to the enclave, only to forward them to the OS. Autarky is 18% slower, due to the extra enclave transitions. Eliding the upcall and AEX overheads improves performance over the unprotected version by 3%. Furthermore, we validate that Autarky does not incur measurable overhead by running the same workload using a small 512×512 image that fits into EPC.

Spell checking server. Hunspell [23] is a spell checker shown to be vulnerable to controlled-channel attacks. It stores its dictionary in a hash table. The original attack [76] logged page accesses when populating the hash table. When a query arrived, the attack matched the page access sequences to reveal input words (assuming correct spelling).

Like libjpeg, Hunspell's resident set can easily fit in EPC: the total memory used by Graphene to execute Hunspell and its libraries is 9.3 MB. Dictionaries are small. For example, the en_US dictionary in the original attack contains 49k entries. Since a typical word is shorter than 10 characters, the resident set fits easily in EPC. Therefore, Autarky can *automatically* mitigate the attack by marking all pages as enclave-managed, without any performance impact.

Using application-defined clusters. The EPC may be too small for a *spelling server* that uses multiple dictionaries. However, in this case the working set is much smaller than the total memory used in the application. To mitigate the attack, the pages of each dictionary can each be a separate cluster. Thus, accesses within a dictionary are protected, but the attacker may learn which dictionary is being accessed, which is less sensitive than learning the words themselves.

We use Hunspell's included sample program to simulate such a use case, modifying it to assign the pages of initialized dictionaries to distinct clusters. The server then reads an input file and checks spelling for the requested language.

To measure the overall performance, we loaded 15 dictionaries that together exceed EPC and trigger page faults. Like the original attack [76], we spell check "The Wonderful Wizard of Oz" (39,588 words). Our performance measurements pessimistically include the time to load dictionaries and initialize clusters; we load English first to ensure it will

Table 2. End-to-end performance of applications using page clusters

Workload	LOC (modified)		Page faults	Enclave-managed pages	Unprotected	Autarky		
						as measured	no upcall	no upcall/AEX
libjpeg 9c [24]	27,776	(2)	408,500	2,065	38.7 MB/s	32.6 MB/s (−18%)	36.3 MB/s (−6%)	39.8 MB/s (3%)
Hunspell 1.7.0 [23]	16,615	(30)	49,501	44,387	16 kwd/s	12.8 kwd/s (−25%)	13.8 kwd/s (−16%)	14.6 kwd/s (−9%)
Freetype 2.9.1 [18]	122,662	(0)	0	2,963	149 kop/s	149 kop/s (1×)	149 kop/s (1×)	149 kop/s (1×)

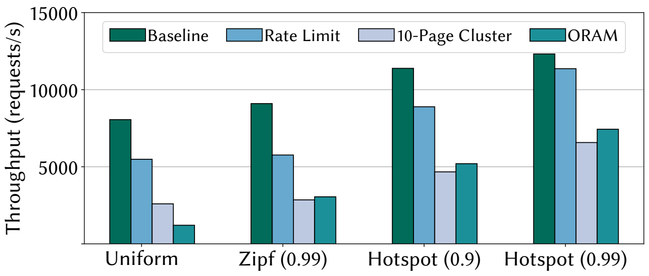

Figure 8. Memcached with Autarky's paging policies.

be evicted by the time of the spell check. Table 2 shows the results. The overheads are dominated by page faults during dictionary load. The spell check itself performs similarly to the baseline, since its first fault brings in a single cluster that contains all pages for that dictionary (US). Note, not all the dictionary pages are used during the spell check, yet they are all brought with the cluster, which is the main cost.

FreeType library. FreeType is a font rendering library. The original attack leaked rendered text by observing control flow via code fetches. Autarky automatically mitigates it with no measurable overheads when rendering different characters by pinning all code pages and libraries in EPC (Table 2).

Memcached. Memcached [42] is a popular key-value store that was also used by much of the prior work on SGX [4, 38, 47, 48]. Memcached can easily oversubscribe EPC resulting in paging and potential leakage of sensitive keys. To overcome this we use our ORAM construction to obliviously access all data. We evaluate Memcached v1.5.17 using YCSB [13] with the predefined *workload C* as in prior work [4, 38]. It performs 100% random GET operations for 1 KB entries. We co-locate YCSB and Memcached on the same machine, pinning each to separate CPUs, allowing us to avoid network overheads. We configure Memcached to hold all data (no misses) with a single serving thread due to thread-safety limitations of our ORAM implementation. We load the server with 400 MB of data to trigger paging and report the maximum throughput.

Figure 8 compares the insecure baseline to Autarky with the different supported paging policies: ORAM, rate-limited paging and page clusters. To support page clusters, we modify Memcached's slab allocation (30 LOC) such that all accesses to the items in the key-value store are managed by clusters holding 10 pages. Similarly, we recompile Memcached to use ORAM for all the items stored, with 1 GB PathORAM tree and 128 MB software cache. Rate-limited paging is supported without any change and we fine-tune the limit to eliminate attacks being reported. We evaluate four configurations: uniform access, Zipfian with $\alpha = 0.99$ (hit rate about 90%) and hotspot. In hotspot, we define 1% of the entries as a *hot set* with an access probability of 90% or 99%. As expected, the rate-limited paging performance impact is the lowest and is mainly due to the added overhead of enclave transitions for each page fault. We observe a lower constant overhead for 10-page clusters compared to ORAM when requests are made with a uniform distribution. In such cases, using clusters may be more favorable. However, for more skewed distributions, the performance difference diminishes and using ORAM can be even faster. The intuition is that clusters are less efficient in utilizing the cache since they bring 10 pages, some of which may not be useful for the other hot requests. Finally, for the hottest distribution, ORAM is only 60% slower than the insecure baseline, which may prove acceptable for some security-sensitive applications.

8 Related Work

Controlled-channel attacks and their implications were extensively studied [45, 59, 61, 66, 67, 72, 76]. One common mitigation approach, on which we build, is to *detect spurious page faults*, and terminate the enclave. However, this approach requires changing or recompiling enclave programs, is prone to false positives, and restricts legitimate demand paging. T-SGX and Déjà Vu rely on transactional memory instructions [12, 58], and Varys modifies the program to add a co-running thread for detecting enclave preemption [46]. Another proposal [63] introduces a minor change to SGX to detect page faults. However, it appears unsuitable for enclaves that would use demand paging or even whose mappings exceed TLB capacity as it requires pre-loading TLB entries for all sensitive pages. To our knowledge, Autarky is the only controlled channel defense that retains demand paging with minimal overheads and minor architecture changes.

Intel suggests that developers should *hide enclave access patterns* [30]. To this end, prior work proposed obfuscating page accesses [8, 19, 35, 59], or using oblivious RAM [1, 48, 51, 54]. Autarky also uses ORAM for secure paging. However, since our architecture changes prevent leaking of access to mapped EPC pages, it becomes possible to cache ORAM data in EPC, reducing the overheads significantly.

Clean-slate enclave designs use *private page tables* to avoid controlled channels. For example, Apparition [16] achieves this through compiler instrumentation of an untrusted kernel, while Sanctum and Keystone rely on hardware extensions [15, 37]. Autarky is compatible with the existing SGX design using a shared page table maintained by the untrusted OS.

Other trusted execution environments exist that cannot support legitimate paging without leaking page access patterns. Notably, the recently-proposed AMD SEV-SNP [3, 33] architecture secures guest VMs from a potentially malicious hypervisor. Similarly to SGX's EPCM structure, it uses a "reverse map table" to check the correctness of nested page table translations, and it therefore appears vulnerable to the same controlled-channel attacks as SGX. We expect that the overall approach proposed by Autarky will apply equally well at the VM level, but defer the details to future work.

Komodo [17] supports enclaves on ARM processors using TrustZone, but lacks support for paging. Autarky's design for enclave-private page faults and self-paging enclaves could be applied to Komodo, ideally without weakening its security guarantees.

While we prototype with Graphene-SGX [65], Autarky could be coupled with various enclave runtimes, including libOSes [4, 6, 50, 60] or other defenses [36, 57]. Vault [64] extends EPC to all available physical memory, reducing demand paging; Autarky would still help to secure it against paging side channels.

Prior work showed that self-paging is useful both within enclaves [47, 48] and outside them [21]. Cooperative memory resource management across privilege domains was also studied extensively [e.g., 47, 71]. Using similar approaches to coordinate memory demands between the OS and multiple distrusting enclaves is an open research topic.

9 Conclusion

Autarky is a hardware/software co-design to mitigate severe *architectural* controlled-channel attacks on SGX enclaves with minimal hardware changes. Autarky modifies SGX to revoke control of paging from the OS and delegate it to a *secure self-paging runtime*. Our results show that Autarky mitigates attacks on real applications and secures demand paging with low overhead. We hope that our work will enable practical controlled channel mitigation on future systems.

Acknowledgments

We thank the anonymous reviewers, Mark Shanahan, Mona Vij, and our shepherd Neeraj Suri for valuable feedback. We gratefully acknowledge the support of the Technion Hiroshi Fujiwara Cybersecurity center.

References

[1] Shaizeen Aga and Satish Narayanasamy. InvisiPage: Oblivious demand paging for secure enclaves. In *46th International Symposium on Computer Architecture*, ISCA '19, pages 372–384, 2019. ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322265.

[2] *ECS Bare Metal Instance.* Alibaba Cloud, 2018. URL https://www.alibabacloud.com/product/ebm. Accessed: 2019-08-08.

[3] *AMD SEV-SNP: Strengthening VM isolation with integrity protection and more.* AMD, January 2020. URL https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[4] 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. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 689–703, 2016.

[5] Andrew Baumann. Hardware is the new software. In *16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 132–137, 2017. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103002.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, October 2014. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454128.

[8] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: Hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, September 2017. URL https://arxiv.org/abs/1709.09917.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017. URL https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[10] Somnath Chakrabarti, Rebekah Leslie-Hurd, Mona Vij, Frank McKeen, Carlos Rozas, Dror Caspi, Ilya Alexandrovich, and Ittai Anati. *Intel Software Guard Extensions (Intel SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment.* 2017. ISBN 9781450352666. doi: 10.1145/3092627.3092634.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. *CoRR*, abs/1802.09085, June 2018. URL https://arxiv.org/abs/1802.09085.

[12] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *12th ACM Asia Conference on Computer and Communications Security*, pages 7–18, 2017. doi: 10.1145/3052973.3053007.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, February 2017. http://eprint.iacr.org/2016/086.

[15] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, pages 857–874, 2016.

[16] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th USENIX Security Symposium*, pages 1441–1458, 2018.

[17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *26th ACM Symposium on Operating Systems Principles*, pages 287–305, 2017. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132782.

[18] *FreeType*. The FreeType Project, 2019. URL http://www.freetype.org/.

[19] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults. In *20th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 357–380. Springer, 2017.

[20] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996. doi: 10.1145/233551.233553.

[21] Steven M. Hand. Self-paging in the Nemesis operating system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, 1999. ISBN 1-880446-39-1. URL https://www.usenix.org/events/osdi99/hand.html.

[22] Troy D. Hanson and Arthur O'Dwyer. *uthash: Hash Table for C Structures*, 2019. URL https://troydhanson.github.io/uthash/.

[23] Hunspell. *Hunspell*, 2019. URL http://hunspell.github.io/.

[24] *libjpeg*. Independent JPEG Group, 2019. URL http://libjpeg.sourceforge.net/.

[25] *Integrated Performance Primitives Cryptography*. Intel, 2019. URL https://github.com/intel/ipp-crypto.

[26] *SGX Linux Driver*. Intel, 2019. URL https://github.com/intel/linux-sgx-driver.

[27] *SGX SDK for Linux*. Intel, 2019. URL https://github.com/intel/linux-sgx.

[28] *SGX Tutorial at ISCA 2015*. Intel Corp., June 2015. Ref. #332680-002 https://software.intel.com/sites/default/files/332680-002.pdf.

[29] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp., May 2019. Ref. #325462-070US.

[30] Simon Johnson. Intel SGX and side-channels. Intel Developer Zone, February 2018. URL https://software.intel.com/en-us/articles/intel-sgx-and-side-channels. Accessed: 2019-07-29.

[31] Simon Johnson. *Scaling towards confidential computing*. Intel, 2019. URL https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf. Keynote presentation at SysTEX 2019.

[32] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating Systems Principles*, pages 52–65, 1997. ISBN 0-89791-916-5. doi: 10.1145/268998.266644.

[33] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, April 2016. URL https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[34] Pratheek Karnati. *Data-in-use protection on IBM Cloud using Intel SGX*. IBM, May 2018. URL https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[35] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *Computers & Security*, 82:118–139, 2019.

[36] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: memory safety for shielded execution. In *EuroSys Conference*, pages 205–221, 2017. doi: 10.1145/3064176.3064192.

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: A framework for architecting TEEs. *CoRR*, abs/1907.10119, July 2019. URL https://arxiv.org/abs/1907.10119.

[38] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference*, pages 285–298, Santa Clara, CA, 2017. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[39] Uwe F. Mayer. BYTE magazine native mode benchmarks. URL https://www.math.utah.edu/~mayer/linux/bmark.html. Accessed: 2019-11-02.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[41] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. Intel SGX support for dynamic memory management inside an enclave. In *5th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:9, 2016.

[42] Memcached. *Memcached*, 2019. URL http://memcached.org/.

[43] *Open Enclave SDK*. Microsoft. URL https://openenclave.io/. Accessed: 2019-08-16.

[44] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer International Publishing, 2017.

[45] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled instruction-level attacks on enclaves for maximal key extraction. *CoRR*, abs/2002.08437, February 2020. URL https://arxiv.org/abs/2002.08437.

[46] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference*, pages 227–240, 2018.

[47] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *EuroSys Conference*, pages 238–253, 2017.

[48] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. CoSMIX: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference*, pages 555–570, July 2019. ISBN 978-1-939133-03-8.

[49] Nelly Porter, Jason Garms, and Sergey Simakov. Introducing Asylo: an open-source framework for confidential computing, May 2018. URL https://cloud.google.com/blog/products/gcp/introducing-asylo-an-open-source-framework-for-confidential-computing. Accessed: 2019-07-30.

[50] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *CoRR*, abs/1908.11143, August 2019. URL https://arxiv.org/abs/1908.11143.

[51] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, pages 431–446, August 2015. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane.

[52] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *13th IEEE International Symposium on High-Performance Computer Architecture*, pages 13–24, 2007. ISBN 1-4244-0804-0. doi: 10.1109/HPCA.2007.346181.

[53] Mark Russinovich. Introducing Azure confidential computing, September 2017. URL https://azure.microsoft.com/blog/introducing-azure-confidential-computing/. Accessed: 2019-07-30.

[54] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[55] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer International Publishing, 2017.

[56] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *CoRR*, abs/1905.05726, May 2019. URL https://arxiv.org/abs/1905.05726.

[57] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/sgx-shield-enabling-address-space-layout-randomization-sgx-programs/.

[58] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/.

[59] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *11th ACM Asia Conference on Computer and Communications Security*, pages 317–328, 2016. doi: 10.1145/2897845.2897885.

[60] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, February 2017.

[61] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: Enabling microarchitectural replay attacks. In *46th International Symposium on Computer Architecture*, ISCA '19, pages 318–331, 2019. ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322228.

[62] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *20th ACM Conference on Computer and Communications Security*, pages 299–310, 2013. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516660.

[63] Raoul Strackx and Frank Piessens. The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. *CoRR*, abs/1712.08519, December 2017. URL http://arxiv.org/abs/1712.08519.

[64] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *23th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3177155.

[65] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, pages 645–658, 2017. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *2nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6, 2017. ISBN 978-1-4503-5097-6. doi: 10.1145/3152701.3152706.

[67] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*, pages 1041–1056, 2017.

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the SGX ecosystem with Foreshadow. *IEEE Micro*, 39(3):66–74, May 2019. ISSN 0272-1732. doi: 10.1109/MM.2019.2910104.

[69] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, pages 1399–1417, May 2020. doi: 10.1109/SP40000.2020.00089.

[70] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy*, May 2019.

[71] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844146.

[72] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *24th ACM Conference on Computer and Communications Security*, pages 2421–2434, 2017. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134038.

[73] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *22nd ACM Conference on Computer and Communications Security*, pages 850–861, 2015. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813634.

[74] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security (ESORICS)*, pages 440–457. Springer International Publishing, 2016.

[75] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *44th International Symposium on Computer Architecture*, ISCA '17, pages 81–93, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080208.

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side-channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, May 2015. doi: 10.1109/SP.2015.45.

[77] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *5th International Conference on Virtual Execution Environments*, pages 21–30, 2009. ISBN 9781605583754. doi: 10.1145/1508293.1508297.

[78] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 283–298, 2017. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.