Mechanisms of Memory Protection

Eric Thomas Schneider
University of North Carolina at Chapel Hill

Abstract

We explore and discuss mechanisms of memory protection, from well-used mechanisms such as segmentation and virtual memory, to less deployed mechanisms such as capability-based addressing, among others. We discuss how these mechanisms of protection work, and their implementation and use in both industry and academic projects. We assess some of the advantages and disadvantage of these mechanisms. While we chiefly consider mechanisms to address memory *safety*, we also discuss where these mechanisms may fail to address aspects of memory *security*.

1 Introduction

Modern computing systems allow multiple *processing entities*, such as processes, tasks, or virtual machines, to share hardware resources, such as time on the CPU, physical memory, and storage. In this paper, we focus on the different mechanisms of *memory protection* used to achieve *isolation* between different processing entities.

1.1 Scope

We consider mechanisms of memory protection, both software-based and hardware-based, used to achieve *memory safety*, and secondarily, *memory security* in computing systems (defined in the next section).

We include not only modern mechanisms and architectures, but historical ones as well, noting their influence on both modern systems and future ones. For example, while Intel MPK [16] was introduced less than ten years ago, similar architectural features have existed for decades longer in other systems [50, 77]. Thus, it would be remiss to only focus on modern systems.

1.2 Safety versus security

L. Pietre-Cambacedes et al. [72] (which uses the SEMA framework [70]) separates *security* from *safety* by intent; security addresses "risks originating from or exacerbated by malicious intent", while safety addresses "accidental ones, i.e. without malicious intent, but with potential impacts on the system environment".

We define *memory safety*, describing a processing entity (defined in the next section), as a state of being protected from bugs and vulnerabilities dealing with direct memory accesses. Mechanisms providing memory safety to a processing entity should *isolate* that processing entity's memory from direct access of other processing entities. For example, two isolated processes in a standard operating system cannot directly access the memory of each other.

We note that mechanisms addressing memory safety may not sufficient to guarantee *memory security*. To achieve true memory security, a processing entity's memory should be completely inaccessible. If a system is vulnerable to side channels (such as Spectre [45]) or disturbance errors (such as rowhammer [42, 43]) that provide forms of indirect memory access, then protection from direct memory access does not achieve memory security. Still, even if a collection of mechanisms does not guarantee full memory security, the mechanisms may be enough to guarantee memory safety without the presence of malicious actors.

1.3 Processing entities

Traditional operating systems, such as Multics [94] and Unix [74], have the notation of a *process* to describe an instance of a program in execution. A process may be running, or if paused, may be ready to run or blocked (if awaiting some resource). Regardless of its status, a process uses multiple resources, including memory, to function.

However, there are other supervisor software with alternative notations, or terms. Hypervisors for example schedule *virtual machines*, which in turn schedule processes. Real-time operating systems schedule *tasks*, which are analogous to processes or threads. Furthermore, hardware modules outside of the CPU can act similar to a process, using a portion of shared memory or other resources. Thus, we generalize the idea of a

process as a *processing entity*.

We define a processing entity by two properties. First, a processing entity must be an encapsulated (but not necessarily independent) unit of execution on a computing system. Secondly, a processing entity must use some subset of the resources on a computing system; particularly, it must use memory for our use case. That memory it uses "belongs" to the processing entity, and may need to be protected by some memory protection mechanisms.

Notice this definition of processing entity is highly inclusive. This definition can be applied to processes, tasks, virtual machines, supervisor and hypervisor software, as well as hardware modules connected to main memory, and even threads. The CPU or individual functions could meet this definition; however, in this paper, by processing entity we usually mean a process running in a standard operating systems.

Our definition allows for processing entities that can be nested, such as a process in an operating system.

1.4 Protection models

A system uses one or more *mechanisms of memory protection* to achieve a *model of memory protection*. A model of memory protection should distinguish between different types of processing entities and how (or if) they should be isolated.

For example, in the traditional operating system model, the memory used by each process needs to be protected from the other processes on the system, and the operating system's memory needs to be protected from the processes it manages. This model does not include protecting processes from the operating system, but there are alternative models, based on the idea of trusted execution environments (TEEs) that do. For example, Intel SGX [17, 58] includes a processing entity called an *enclave*, whose memory is protected from the operating system.

Generally, we say memory is "protected" from a processing entity when it can neither be read nor written to. However, it is possible that for some models and some processing entities, only integrity needs to be guaranteed, not confidentiality. Many systems may also include other nuances, such as marking pages executable or non-executable for further protection.

2 Physical memory

Physical memory is the location where primary storage or main memory is physically stored, and is the only memory directly accessible to the CPU. After possible layers of translation and protection, the instructions and other data of a processing entity are ultimately served from physical memory.

2.1 Formalization

Conceptually, we may define physical memory as the mapping

$$M : A_{\text{val}} \mapsto V$$

where is $a \in A_{\text{val}}$ is a *valid* address, and $v \in V$ is a fixed-sized value. Each address a is an unsigned integer of a fixed size.

Modern architectures such as x86-64 define 64-bit addresses, although implementations usually support a smaller physical address [16]. For the purposes of this paper, we may define a to be a 64-bit integer. While $A_{\text{val}} \subseteq [0, 2^{64}]$ in such modern systems, there are so far no systems where $A_{val} = [0, 2^{64}]$. We also note that A_{val} does not need to be contiguous, that a CPU could allocate the addresses for physical memory in a way that could have gaps.

We may define $v \in V$ to be an 8-bit "byte", as is ubiquitous in modern systems, although there are historical systems that use different sizes.

We may then define a physical read:

$$\textbf{read} : A_{\text{pos}} \mapsto V \cup \{e\}$$

$$\textbf{read}(a_{\text{pos}}) = \begin{cases} M(a_{\text{pos}}) & a_{\text{pos}} \in A_{\text{val}} \\ e & a_{\text{pos}} \notin A_{\text{val}} \end{cases}$$

where A_{pos} is the set of possible memory address ($A_{\text{pos}} = [0, 2^{64}]$ in the 64-bit case), $v \in V$ is a value from memory (as defined previously), and e is a constant "error" value (with $e \notin V$).

Now, we may define a physical write:

$$\textbf{write} : (A_{\text{pos}}, V) \mapsto \{\varnothing, e\}$$

$$\textbf{write}(a_{\text{pos}}, v) = \begin{cases} M(a_{\text{pos}}) \leftarrow v, \varnothing & a_{\text{pos}} \in A_{\text{val}} \\ e & a_{\text{pos}} \notin A_{\text{val}} \end{cases}$$

where $\varnothing$ is a constant "success" or "void" value ($\varnothing \notin V$). In our definition, if the write is to a valid address, then the memory at that location will be updated to the given value.

While we return a special value (e) as the response when $a_{\text{pos}} \notin A_{\text{val}}$, modern processors will typically issue an exception when an invalid address is used, which may update the program counter to a set location, such as a handler.

The division of possible addresses (A_{pos}) into *valid* addresses (A_{val}) and *invalid* addresses ($A_{\text{pos}} - A_{\text{val}}$) is an important one. While this division is currently between locations in physical memory that exist or do not, various mechanisms of memory protection will expand upon this model.

2.2 In practice

A combination of memory technologies in modern systems forms a *memory hierarchy* [68], with faster, more expensive memory technologies at the top, such as SRAM, and slower,

less expensive memory technologies at the bottom, such as disk. DRAM (dynamic random access memory) is somewhere in the middle, and is the modern technology for implementing main memory. Values in DRAM cells are stored as a charge in a capacitor, which must be periodically refreshed. Modern DRAM is Synchronous DRAM (SDRAM), which is clocked together with the CPU. The fastest versions of DRAM/SDRAM is Double Data Rate (DDR) SDAM, which commits data transfers on both the rising and falling edge of the clock. The current version of DDR is DDR5.

DRAM is commercially released on small boards as dual inline memory modules (DIMMs). Historically, magnetic drums were used for main memory, and eventually it was replaced by magnetic-core memory.

Physical memory is often referred to as "main memory", "core", or "main storage" (in z/Architecture [33]). However, we only use the term physical memory for clarity.

3 Segmentation

With segmentation, each processing entity is associated with a set of *segments*. Each segment has a base address and a maximum offset. When memory is accessed by a processing entity, a segment is provided with an unsigned offset, and the physical address is computing by adding the offset to the base address.

3.1 Formalization

We may define a segment s as a tuple of three properties as follows:

$$s = \langle a_{\text{base}}, a_{\text{max\_off}} \rangle$$

where a_{base} (where $a_{\text{base}} \in A_{\text{pos}}$) is the base address, $a_{\text{max\_off}}$ (where $a_{\text{max\_off}}$ is an unsigned integer of the address size, such as 64 bits) is an offset of the base address, and p is the permissions.

Each processing entity has a collection of n segments, from s_1 to s_n. Now, recall our two core physical memory operations:

$$\mathbf{read_p} : A_{\text{pos}} \mapsto V \cup \{e\}$$
$$\mathbf{write_p} : (A_{\text{pos}}, V) \mapsto \{\varnothing, e\}$$

We mark these with $_p$ to distinguish between physical and other kinds of reads. With this in mind, we may then define a segmented read as:

$$\mathbf{read_s} : (A_{\text{off}}, S) \mapsto V \cup \{e\}$$
$$\mathbf{read_s}(a_{\text{off}}, s) = \begin{cases} \mathbf{read_p}(a_{\text{off}} + a_{\text{base}}) & a_{\text{off}} < a_{\text{max\_off}} \\ e & a_{\text{off}} \geq a_{\text{max\_off}} \end{cases}$$

Similarly, we may define a segmented write as

$$\mathbf{write_s} : (A_{\text{off}}, S, V) \mapsto \{\varnothing, e\}$$
$$\mathbf{write_s}(a_{\text{off}}, s, v) = \begin{cases} \mathbf{write_p}(a_{\text{off}} + a_{\text{base}}, v) & a_{\text{off}} < a_{\text{max\_off}} \\ e & a_{\text{off}} \geq a_{\text{max\_off}} \end{cases}$$

3.2 Permissions generalization

Segmentation in most implementations includes permissions. We can expand our model by adding a permission field, p, to a segment s.

$$s = \langle a_{\text{base}}, a_{\text{max\_off}}, p \rangle$$

We define p as being either read-only (r) or read/write (w), so $p \in \{r, w\}$. In many segmentation implementations, p is extended with an execute (x) option, and all options can be combined together into different configurations (such as r and x, but no w), but we do not include this in our formalization.

Noting a segmented read is the same regardless of permissions, we may then redefine segmented write:

$$\mathbf{write_s} : (A_{\text{off}}, S, V) \mapsto \{\varnothing, e\}$$
$$\mathbf{write_s}(a_{\text{off}}, s, v) = \begin{cases} \mathbf{write_p}(a_{\text{off}} + a_{\text{base}}, v) & a_{\text{off}} < a_{\text{max\_off}} \\ & \wedge p = w \\ e & \text{otherwise} \end{cases}$$

3.3 In practice

Segmentation has usually been combined with virtual memory, with segments layered atop a virtual address space.

Multics [76, 94] was an early operating system that took advantage of hardware segments introduced by the GE 645 computer. In Multics, an address consists of a segment and the offset from the segment, which is then translated into a memory location. The concept of segmentation is also used for storage, as segments and files are considered equivalent in Multics.

The IBM System/370 [77] was an early system that included a form of segmentation. Every storage (memory) reference required an access register (a segment), with the base address stored in a register.

x86 includes six registers for segmentation, with one code segment (CS), one data segment (DS), one stack segment (SS), an "extra" segment (ES), and two "general" segments (FS, GS) [16]. Most of these segments cannot be used outside of 32-bit mode, and as such, segmentation is rarely used in practice in modern systems. The FS or GS registers are however commonly used to access thread local storage (TLS), which is set up by the operating system.

However, it has occasionally been, such as for Native Client [104] which uses 32-bit mode and segmentation to sandbox

untrusted/third-party x86 binaries. Native Client protects its runtime from arbitrary x86 code firstly by using an "outer sandbox", which disallows certain instructions, such as system calls, as well as paradigms like self-modifying code and overlapping instructions. If an inappropriate instruction sequence is identified, Native Client will not execute the binary. Then, Native Client uses an "inner sandbox" which uses x86 segments to protect the service runtime (which handles necessary system calls, such as memory allocations and thread management), as well as to intercept system calls if the outer sandbox fails to contain them. With segmentation in place, Native Client modules can only access the memory belonging to the module, and cannot access the service runtime or other portions of memory.

V. Karakostas and J. Gandhi reevaluated segmentation by proposing Redundant Memory Mappings (RMM) [39]. RMM adds a range table, which maps a contagious range of virtual memory to contagious physical memory, with an associated range TLB. They still maintain virtual memory paging structures and a (page-based) TLB, leading to redundancy in representation, but allowing for flexibility. They find that, for many applications, the range TLB has a much higher hit rate than the page-based TLB due a greater TLB reach, avoiding decreased latency from page walks.

4 Virtual memory

While modern virtual memory can encompass multiple techniques and properties, we focus on the ability of hardware to provide a *virtual address space*. This virtualized address space may then exclude the memory contents of processing entities to be protected. Thus, switching between different processing entities may require switching between virtual address spaces.

Typically, physical memory is divided into fixed-size *pages*, and supervisor software provides a map of virtual pages to physical pages which is enforced by the hardware during *address translation*.

In most modern systems, particularly Unix-based systems, virtual address spaces typically include both a process and the operating system. The operating system is not protected through the scope of the virtual address space, but through other mechanisms typically associated with virtual memory, such as the marking of individual pages as supervisor-only.

Theoretically, a computer system could enforce isolation using purely virtual address space exclusion. Notably, the KAISER [1] [23] patch to x86-64 Linux implements *kernel address isolation*, moving most of the kernel out of each process's virtual address spaces to prevent side-channel attacks such as Meltdown [55]. However, KAISER still requires some minimum of pages to be mapped in the process address space due to limitations in x86-64 and Linux. For example, the interrupt descriptor table (IDT) must be mapped for x86 to correctly handle context switches.

4.1 Formalization

We define virtual memory as the mapping

$$M_v : A_v \mapsto A_{\text{pos}}$$

where $a_v \in A_V$ is a valid virtual address, and a_{pos} is a possible physical address. We may then define a virtualized read as:

$$\mathbf{read_v} : A_{\text{pos}} \mapsto V \cup \{e\}$$

$$\mathbf{read_v}(a_{\text{pos}}) = \begin{cases} \mathbf{read_p}(M_v(a_{\text{pos}})) & a_{\text{pos}} \in A_v \\ e & a_{\text{pos}} \notin A_v \end{cases}$$

Similarly, we may defined a virtualized write as:

$$\mathbf{write_v} : (A_{\text{pos}}, V) \mapsto \{\varnothing, e\}$$

$$\mathbf{write_v}(a_{\text{pos}}, v) = \begin{cases} \mathbf{write_p}(M_v(a_{\text{pos}}), v) & a_{\text{pos}} \in A_v \\ e & a_{\text{pos}} \notin A_v \end{cases}$$

4.2 Generalization

Modern CPUs often support *second level address translation*, which nests virtual address spaces. We can generalize our model by defining up to n virtual memory mappings:

$$M_{v_1} : A_{v_1} \mapsto A_{\text{pos}}$$
$$\ldots$$
$$M_{v_n} : A_{v_n} \mapsto A_{\text{pos}}$$

We may then redefine a virtualized read and write:

$$\mathbf{read_v}(a_{\text{pos}}) = \begin{cases} \mathbf{read_p}(M_{v_1} \circ \ldots \circ M_{v_n}(a_{\text{pos}})) & a_{\text{pos}} \in A_v \\ e & a_{\text{pos}} \notin A_v \end{cases}$$

$$\mathbf{write_v}(a_{\text{pos}}, v) = \begin{cases} \mathbf{write_p}(M_{v_1} \circ \ldots \circ M_{v_n}(a_{\text{pos}}), v) & a_{\text{pos}} \in A_v \\ e & a_{\text{pos}} \notin A_v \end{cases}$$

4.3 In practice

In x86/x86-64, virtual memory is mapped using a hierarchical paging structural [16], called a *page table*. Memory is divided into pages, usually 4096 bytes in size, and each address is divided into a *page number* [2] and an *offset*. A portion of the page number, initially the upper bits, is used to index each paging structure. The initial paging structure is stored in a

[1] KAISER stands for "Kernel Address Isolation to have Side-channels Efficiently Removed. Nowadays, this feature is referred to as KPTI ("Kernel Page Table Isolation"), since KAISER does not efficiently remove all transient execution attacks, such as Spectre [45].

[2] Intel uses the term *page number* to refer to the upper bits of a virtual address and the term *page frame* to refer to the upper bits of a physical address. However, we use the term page number, which has often been to used to describe upper bits of both.

special register (CR3), and each entry in a paging structure contains either the next paging structure or the physical page number, which can combined with the offset to obtain a physical address. If at some point the entry is not marked as present, a *page fault* is generated, transferring control to supervisor software.

Because translating a virtual memory address can require many memory accesses to iterate through the page table, mappings of virtual page numbers to physical page frames are cached in a CPU structure called the *translation lookaside buffer*, or *TLB*. Switching between virtual address address spaces may require the TLB to be flushed; however, x86-86 allows for a 12-bit *process-context identifier* (PCID) to be set, which can be used to distinguish between different processing entities using different virtual address spaces. A TLB can then be indexed by both the present PCID and the virtual page number, allowing address translations with different PCIDs to remain in the TLB.

In 32-bit MIPS [37], the TLB is completely managed by software. Virtual page to physical mappings must be supplied by supervisor instructions; it is up to the supervisor to create its own paging structure, or use an alternative structure, to track mappings that do not fit in the TLB. Similar to x86, TLB entries can be marked with an *address-space identifier* (ASID), and there is an 8-bit register[3] for the current ASID.

In RISC-V [95], address translation is handled similarly to x86. However, RISC-V provides configurably sized virtual memory systems; with a greater virtual address space, more can be represented, but address translation requires more resources (more memory used for page tables, latency of traversals, microarchitectural size of translation hardware). 64-bit RISC-V has 3 virtual memory systems to choose from: Sv39 (39-bits, allowing a 512 GiB virtual address space), Sv48 (48-bits, 256 TiB), and Sv57 (57-bits, 128 PiB). A RISC-V implementation must implement at least Sv39, and must implement Sv48 if Sv57 is implemented to be backwards compatible with software assuming a smaller version.

Virtual memory is referred to as "virtual storage" in z/Architecture [33]. A virtual address is converted to a *real address* using *dynamic address translation* (DAT), and then that real address is converted to an absolute (physical) address using *prefixing*. Dynamic address translation is similar in mechanics to x86 virtual address translation. Prefixing reassigns the real memory range of 0 to 8191 to a different block in physical memory. This provides a private area of storage for each CPU core, which is mainly used for interrupts.

Virtual memory has a non-negligible performance cost. H. Choi et al. [15] benchmarked Linux and uClinux (Linux with MMU use disabled) on ARM and found uClinux to be significantly faster at context switching. However, the analysis is severely out of date (2005). Singularity [1, 30], discussed in greater detail later, found software-based process isolation to be faster on microbenchmarks, especially on those relating to interprocess communication (IPC). Similarly, J. Liedtke [53, 54] found system calls and virtual address space switching to be very expensive for IPC, together taking up 60% of the cycles used for an IPC message to be sent, with an address space switch taking up about 20%. While updating the root of the page table can be a small cost, if the TLB must be flushed, then switching between processes is a massive overhead. These overheads are especially bothersome on microkernels, which heavily rely on IPC.

[3] This is describing the R6000 specifically. The register is actually is 32 bits, but the upper 24 bits are reserved.

4.4 Beyond isolation

In modern systems, virtual memory has greater application than being an isolation primitive. A contiguous virtual address space can obfuscate fragmentation in physical memory, as well as can map objects not present or only partial present in physical memory, such as contents of disk. Modern operating systems use memory management techniques such as demand paging, which can exclude portions of memory unlikely to be used or used often, but bring them into memory upon a page fault. This leads to less pressure on physical memory. *Swapping* (moving pages in and out of physical memory, to and from secondary storage) can even allow for systems to use more memory than they physically have.

One of the earliest systems to introduce a form of virtual memory was the Atlas system [40, 41]. Atlas used its *one-level store* (quasi-virtual memory) with the goal of presenting processes with a greater amount of memory than was available in core memory. Atlas had 16,000 words in core memory, but 96,000 words in drum, which is slower to access (with drum accesses taking 6 milliseconds, and core taking a few microseconds).

The one-level store allowed for the reassignment of physical addresses. For each page in memory, there was a *page address register*. Upon a memory access, the memory location was compared to the page address register, and if they are not equal, a "non-equivalence" interrupt was generated, which would swap a sector from drum to memory. In an early application of a page replacement algorithm, Atlas also included a "learning program", which would swap out pages unused after a certain period of inactivity. One-level stores blur the distinction between ephemeral and persistent application state; Auora [84] is a modern system which reevaluates the one-level store.

Opal [8], a research operating system, uses virtual memory for these advantages, but it uses a *single virtual address space* are page-groups for protection instead of isolated virtual address spaces. Opal is built atop HP's Precision Architecture (or PA-RISC) [50], which encourages a global virtual address space. With a single address space, sharing is simplified, since virtual addresses are global, and the penalty of address space switching is removed.

In practice, PA-RISC handles virtual memory similarly to

x86. However, because the virtual address space is global, and furthermore, PA-RISC disallows address aliasing, PA-RISC can optimize its microarchitecture by indexing all levels of cache virtually. This allows the caches to be accessed in parallel with the TLB.

5 Group protection

We generalize a series of protection mechanisms, such as *memory protection keys* and *page groups* (or *page-group protection*) into *group protection*. Simply, each of some unit of memory (generally a page, but it can be more or less), is assigned to a *group*. Each processing entity has access to one or more groups; generally, the supervisor has access to all.

5.1 Formalization

We define an address's group through the map

$$G_a : A_{\text{pos}} \mapsto G$$

where $a_{\text{pos}} \in A_{\text{pos}}$ is a possible address, and $g \in G$ is the group associated with that address. We may define the group g as a fixed-size unsigned integer. Then, we may define the groups that a processing context has through the map

$$G_p : P \mapsto \mathcal{P}(G)$$

where $p \in P$ is a processing entity, and $g_s \in \mathcal{P}(G)$ is some subset of G (that is, $g_s \subseteq G$).

Now, we may define a group-respecting read as

$$\textbf{read}_\textbf{g} : (A_{\text{pos}}, P) \mapsto V \cup \{e\}$$

$$\textbf{read}_\textbf{g}(a_{\text{pos}}, p) = \begin{cases} \textbf{read}_\textbf{p}(a_{\text{pos}}) & G_a(a_{\text{pos}}) \in G_p(p) \\ e & G_a(a_{\text{pos}}) \notin G_p(p) \end{cases}$$

Similarly, we may define a group-respecting write as

$$\textbf{write}_\textbf{g} : (A_{\text{pos}}, P, V) \mapsto \{\varnothing, e\}$$

$$\textbf{write}_\textbf{g}(a_{\text{pos}}, p, v) = \begin{cases} \textbf{write}_\textbf{p}(a_{\text{pos}}, v) & G_a(a_{\text{pos}}) \in G_p(p) \\ e & G_a(a_{\text{pos}}) \notin G_p(p) \end{cases}$$

5.2 In practice

In PA-RISC [14, 50], each page entry has an *access identifier* (AID) which contains a page-group number plus a write-disable bit. Each process has four access identifier which can claim up to 4 groups. With the write-disable bit, an AID can give read access to page but no write access. Two AIDs can share a page-group, but one can be write-disabled, with one write-enabled. This can allow for say, a single writing process with multiple readers.

In a simple application of page-based groups, x86 annotates page table entries with a single bit to distinguish between a *supervisor-mode address* and a *user-mode address* [16], providing two effective groups. In a virtual address space, all user-mode addresses are accessible, but supervisor-mode addresses are only accessible when the CPU is in a privileged mode (ring ≤ 0). In modern operating systems, this allows a supervisor and a process to share the same virtual address space safely (absent vulnerabilities such as Meltdown [55], discussed later), and thus lower the overhead of context switching.

Similar to x86, Atlas annotates page address registers with a lockout bit, to prevent processes from accessing supervisor memory and pages not available to it [41]. Sometimes, a block of information may be locked out if a supervisor operation on that information is not yet complete.

Recent versions of x86-64 introduce Memory Protection Keys (Intel MPK), which uses 4 bits of each page table entry to store a *protection key* [16]. A 32-bit register (PKRU, protection-key rights for user pages) stores 2 bits for each protection key. When MPK is enabled, the PKRU sets read and write permissions for each page, which can limit access to pages in usermode. Even if a page is mapped virtually and lacks a kernel bit, if the page entry uses a key with access disabled in the PKRU, the page cannot be accessed in user mode. While the number of keys is limited to 16, they can be virtualized in software fairly efficiently [24, 67]. The PKRU register can be modified from userspace, limiting its ability to only providing some extra isolation between trusted components, such as for helping sandbox JIT-compiled code.

In z/Architecture [33], pages can similarly be protected with a 4-bit key, which was introduced in the IBM System/360 [77]. A non-supervisor processing entity only has one key at a time, and can only access pages marked with that key. However, one key, the zero key, can be accessed by any processing entity. Operating systems built on System/370 use a key to protect control information inside of a virtual address space.

Group protection has usually been combined with virtual memory, with each page being labeled with a group inside of a TLB entry. However, this need not be the case; E. Koldinger et al. [14] suggest separate address translations for virtualization and for protection, with a *protection lookaside buffer* (PLB) to store protection information. In Opal, a single address space operating system, the virtual address space is global, but the protection is not; that is, a different translation structure for protection is needed for each process, but only one translation structure is needed for virtual memory. A PLB can also be accessed in parallel with a TLB, if the PLB is indexed virtually.

Separate TLB and PLB structures can also allow for differences in granularity; it is possible that for certain workloads, large pages may be preferred to optimize TLB use, but fine-grained protection could be preferred. On the other hand, larger protection pages could be useful; segments of memory such as the stack and code span several virtual pages, but

have the same protection. Similar to how modern TLBs can support multiple page sizes, there is no reason why a PLB couldn't either.

6 Analytic protection

We generalize a few software-based memory protection mechanisms as *analytic protection*. Before a processing entity is spawned and executed, the executable format of a processing entity is either analyzed to ensure it does not access memory outside of itself, or that the executable format is limited to a language or format that has been previously analyzed to follow memory safety (or at least, to not interfere with the other processing entities on a system).

These techniques are all software-based, although the memory safety features of capability architectures (and their derivatives) are not dissimilar.

6.1 Formalization

First, we define a $p_{\text{child}} \in P$ as a child processing entity, and a $p_{\text{parent}} \in P$ as a parent processing entity. The p_{child} is embedded in the address space of p_{parent}, and p_{child} is spawned by p_{parent}; the canonical example of each is a process and an an operating system, but there are many more examples, such as a JS script running in a thread of a web browser, or a Java process running in the Java Virtual Machine (JVM).

We can then describe an approval function as

$$\textbf{approval} : P \mapsto (\{\varnothing, e\}, P)$$

where, given a processing entity loaded into memory $p_{\text{pre}} \in P$, the **approval** operation either approves (returning $\varnothing$) or rejects the process (returning e), and then returns a translated process entity $p_{\text{post}} \in P$.

Translation may be a no-op, depending on implementation.

6.2 In practice

Singularity [1, 21, 30] uses a single address space with *software isolated processes* (SIPs), as opposed to *hardware isolated processes* (HIPs). Singularity SIPs must be compiled into an intermediate language (MSIL), which is statically verified on load time to ensure that that the SIP cannot access memory outside of itself, and that it does not use any privileged instructions. The latter property allows SIPs to run in a privileged CPU level (ring 0 in x86). Singularity can be configured to use HIPS, but running SIPs without virtual memory and in ring 0 significantly removes the overhead caused by switching privilege modes and virtual address spaces.

SIPs cannot share pages or other memory locations directly. However, Singularity supports and facilitates *contract-based channels* that can be made between SIPs for zero-copy interprocess communication. Channels are bi-directional, and are implemented as a lossless, in-order queue of messages. Channel contracts are specified in Sinq# and are statically verified.

Tock [51], an embedded operating system, has hardware-isolated processes, but also adds *capsules*. Capsules are embedded in the kernel and must be written in Rust, to ensure memory and type safety. Similarly, RedLeaf [61] is another Rust-based operating system, similar to Singularity, where programs must be written in Rust.

eBPF (extended Berkeley Packet Filters) [20] is a feature of Windows, Linux, and other Unix-based operating systems based on BPF (Berkeley Packet Filter, or BSD Packet Filter originally) [57]. BPF was originally a tool built into BSD that filters out packets using a given small filter program, then forwarding the results to a usermode process for further processing. Over time, BPF was expanded to allow for more functionality and complexity in the filter, and to respond to events other than network packets, such as system calls, kernel tracepoints, and disk communication.

eBPF programs run in a privileged context (such as ring 0 in x86) and fully in the kernel's section of memory. After being loaded, an eBPF program then undergoes verification to ensure its safety. The verifier will check if the process attempting to use eBPF has those permissions (as of recently, only root can use eBPF per default), that the eBPF program does not use uninitialized memory or access memory out of bonds, that the eBPF program is under a certain size, and that programs have a finite complexity and always run to completion (eBPF programs are not Turing-complete). Then, the JIT compiler converts eBPF bytecode into machine code.

eBPF programs can use eBPF maps to collect and store information. Since eBPF programs are event-driven and only run for a small amount of time, eBPF maps are essentially for storing state. eBPF cannot access kernel methods or structures directly, and thus, there is a standard API providing eBPF helper calls. There are helper calls for generating random numbers, accessing or updating eBPF maps, reading from a trace, or forwarding results. eBPF programs can call other eBPF programs, but only as a tail call, and the composite eBPF program cannot be cyclical.

Portable Native Client [18], or PNaCl, is a successor to Native Client [104]. Instead of using x86 segmentation, it accepts a binary made of LLVM bitcode, an intermediate format, and performs static analysis on it before compiling it to a native format, such as x86-32, x86-64, or ARM. This is not dissimilar to WebAssembly [25], which uses an efficient intermediate format crafted with sandboxing in mind. By design, a WebAssembly module can only modify its own memory, even if the module is sharing an address space.

7 Capabilities

With capability-based addressing, each processing entity is associated with a set of *capabilities*. Each valid capability

describes a memory region and associated permissions; often, capabilities describe a single object, but they can describe larger structures too, such as the stack or all of addressable memory. Similar to segmentation, memory accesses are done on a capability and offset rather than an arbitrary address, which is then translated. New capabilities can only be derived from an existing capability and cannot exceed the scope (in addressing or permissions) of the capability that it is derived from.

The idea of capabilities is often used outside of memory protection in security, such as by file systems and distributed systems.

7.1 Formalization

A capability, in essence, is not dissimilar from a segment. We may reuse our generalization of a segment to define a capability c, but with one extra field:

$$c = \langle a_{\text{base}}, a_{\text{max\_off}}, p, t \rangle$$

where t is the validity tag, a boolean value.

And then we may define a capability-based read similarly:

$$\mathbf{read_c} : (A_{\text{off}}, C) \mapsto V \cup \{e\}$$

$$\mathbf{read_c}(a_{\text{off}}, c) = \begin{cases} \mathbf{read_p}(a_{\text{off}} + a_{\text{base}}) & a_{off} < a_{\text{max\_off}} \\ & \wedge t \\ e & otherwise \end{cases}$$

And a capability-based write similarly:

$$\mathbf{write_c} : (A_{\text{off}}, C, V) \mapsto \{\varnothing, e\}$$

$$\mathbf{write_c}(a_{\text{off}}, c, v) = \begin{cases} \mathbf{write_p}(a_{\text{off}} + a_{\text{base}}, v) & a_{\text{off}} < a_{\text{max\_off}} \\ & \wedge p = w \wedge t \\ e & \text{otherwise} \end{cases}$$

One key difference between segmentation and capabilities is the ability to easily create new capabilities from existing ones. Thus, we can define an operation for creating capabilities as follows:

$$\mathbf{create} : (C, A_{\text{base}}, A_{off}) \mapsto C \cup \{e\}$$

$$\mathbf{create}(c, a_{\text{base}_{\text{new}}}, a_{\text{max\_off}_{\text{new}}}) =$$

$$\begin{cases} \langle a_{\text{base}_{\text{new}}}, a_{\text{max\_off}_{\text{new}}}, p, t \rangle & a_{\text{base}} \leq a_{\text{base}_{\text{new}}} \\ & \wedge t \\ & \wedge a_{\text{base}_{\text{new}}} < a_{\text{base}} + a_{\text{max\_off}} \\ & \wedge a_{\text{max\_off}_{\text{new}}} \leq a_{\text{base}} + a_{\text{max\_off}} \\ e & \text{otherwise} \end{cases}$$

7.2 In practice

CHERI (Capability Hardware Enhanced RISC Instructions) [97, 98, 102] is an hybrid capability ISA extension, extending MIPS (originally), ARM, and RISC-V, with a "sketch" for x86. CHERI is designed to be incremental with existing architectures and can be mixed with conventional MMU-based protections. CHERI-ARM is available as an experimental commercial product as ARM Morello [96].

CHERI capabilities are 128-bit on 64-bit platforms, and 64-bit on 32-bit platforms[4], plus an extra bit for a validity tag [97, 101]. The 128 bits are split into a 64-bit address, and 64 bits of metadata, including a bounds, object type, and permissions. The address can either be a virtual or physical address; it is interpreted with respect to the current addressing mode. The bounds is relative to the address and compressed. The bounds are represented using a float point representation[5], allowing for high precision for small objects, but lower precision for larger objects. The object type is -1 if the capability is "unsealed" (not associated with an object type); otherwise it is compared with during capability operations. The permissions describe how the capability can be used, with standard read, write, and executable permissions, but also permissions to limit capability propagation and some for software. Lastly, the validity tag determines if a capability can be used or not; if invalid, the capability cannot be used, though its fields can be extracted if needed.

Capabilities are stored either in registers, which are doubled in size to fit them, plus a validity tag, or they are stored in memory. Capabilities stored in memory must be aligned, such that each 16 bytes of memory is tagged; the validity tags for in-memory capabilities are also stored in memory in a protected region called a *hierarchical tag table* [36]. Originally, a flat table was used, but the hierarchical tag table, essentially a two-level table, saves space in memory. Any direct write to a memory location or register will invalidate the tag, ensuring capabilities can only be modified with capability-specific instructions.

Capabilities must follow three properties. The first is provenance validity, that capabilities can be constructed from instructions using other valid capabilities. The second is capability monotonicity, that new capabilities cannot exceed the permissions and bounds of the capability it was derived from. The last is reachable capability monotonicity, that during execution, all capabilities accessible to the current processing entity cannot increase. A formal model of the CHERI ISA (specifically CHERI-MIPS) has been formally proven to follow these properties [62].

CHERI was inspired by M-Machine [12], a pure capability machine, adding 64-bit *guarded pointers*. Each pointer contains 54 bits for the (virtual) address, 6 bits for a segment,

[4] It was 256 bits originally for 64-bit platforms, but compressed over time for performance (see [101]).

[5] I don't mean IEEE 754; see [101] for details.

4 for the permission bits, and an additional 1 bit for the tag. Guarded pointers cannot be forged, which would modify the tag bit. Another historical capability system is the IBM System/38 [29]. In System/38, each word has 40 bits, with 32-bits for data, 7 ECC (error correcting code) bits, and 1 tag. Pointers are composed of 4 words. Similarly, the tag bits are not directly addressable.

8 Trusted Execution Environments (TEEs)

A trusted execution environment (TEE) is a processing environment that guarantees the integrity, authenticity, and confidentiality of the executing code and its state [75]. Most importantly, code executing outside the TEE should not be able to modify nor directly inspect the contents of that TEE, including hypervisor and supervisor software.

The most common TEE abstraction is an *enclave*, which is a protected area of physical memory containing both the code and data of an enclave application. After initialization, enclave memory can only be accessed and modified from within the enclave itself, and even privileged components cannot access the enclave memory. Another common alternative TEE model, the "virtual machine" model, isolates individual virtual machines, including a guest operating system and processes, rather than just a program or portion of a program.

In order to prove authenticity, TEEs typically provide primitives for *attestation*. Upon enclave creation, the initial memory contents of the enclave are cryptographically measured. Then, the authenticity of the enclave can be verified to other applications on the system, or potentially over the network to a remote party. If the initial state is tampered with, then the measurement will not match, and the enclave application will be known to be untrusted.

8.1 Formalization

TEEs are typically implemented as another layer of memory protection and can be modeled differently depending on implementation. Most follow something similar to group protection.

8.2 In practice

Intel Software Guard Extension (SGX) is an extension to x86, and is one of the first major TEE implementations and one of the most widely deployed [58] [17], being employed by cloud providers such as Alibaba [2, 3], IBM [31, 32], and Microsoft [59, 106]. Besides from its use in the cloud, SGX has been used as a foundation of trust for blockchain networks, such as the Secret Network, well as for digital rights management, such as by CyberLink PowerDVD for blu-ray [91].

SGX follows the enclave model, with enclaves created and managed through special instructions. Enclave pages are stored in a physical memory area the Enclave Page Cache (EPC). The EPC is part of the Processor Reserved Memory (PRM), a contiguous range of physical memory, which also includes other data structures storing metadata like the Enclave Page Cache Map (EPCM), the SGX Enclave Control Store (SECS), and the Thread Control Structure (TCS). To protect from attacks targeting DRAM modules, the EPC is cryptographically encrypted using Intel's Memory Encryption Engine (MEE). The MEE encrypts enclave pages whenever they leave the CPU, and decrypts them when entering the CPU.

Because SGX requires special instructions to be able to use, applications not engineered specifically for SGX usually use a library operating system or a similar kind of compatibility layer, such as Gramine [83], Scone [6], Haven [9], and Occlum [81]. Theoretically, a library operating system can be ported to other host OSs and TEEs.

In response to the complexity of SGX, newer TEE implementations follow a "virtual machine" model, like AMD SEV-SNP [4] and Intel TDX [34], where virtual machines are isolated from the hypervisor and from other virtual machines, instead of isolating individual applications or parts of applications. While no modification needs to be done to support applications running on an isolated virtual machine, a greater amount of code (the guest OS) has to be trusted.

Keystone [49] is an open source, software-based RISC-V TEE. Keystone utilizes *physical memory protection* (PMP), a key feature of the RISC-V privileged instruction set, that allows Keystone to run on unmodified RISC-V hardware. The core component of Keystone is its *security monitor* (SM), which is implemented in the M-mode privilege mode, and is more privileged than the untrusted operating system which runs in S-mode (the equivalent of ring 0).

Keystone's security monitor uses PMP to protect sections of physical memory; each enclave uses a PMP entry, and the security monitor uses two. Keystone's use of PMP is both a major strength and a major weakness in that, while using PMP allows Keystone to run on unmodified hardware, there is an architectural limit of up to 64 PMP entities, with many RISC-V systems only having 8 or 16 entries.

Unique to Keystone is its *runtime* (RT) component, which serves as something like a library operating system, managing system calls and memory management. The runtime runs in S-mode. Keystone also includes an SDK for easily creating and launching enclaves.

Keystone has mainly been used in academia to prototype experimental TEE designs; one is Elasticlave [105], a TEE model allowing sharing of memory between enclaves. While there are techniques for enclaves to share data using more rigid TEE models, Elasticlave has significantly less overhead costs. Cerberus [48] is a similar TEE model allowing enclave memory sharing, but is formally verified. Keystone has also been used to explore hardware-accelerated for TEE booting [28], TEE implementation on real-time systems [82], and composite enclaves [78], with enclaves composed of multiple

distributed unit enclaves.

ARM TrustZone [71] is an older but widely deployed TEE/security solution. TrustZone divides memory and other resources into a *secure world* and a *normal world*, with software in the normal world not able to access resources of the secure world. The OS runs in the normal world, isolating software running in the secure world. Newer ARM processors include ARM CCA [5], which adds a *realm world*. The realm world is composed of isolated *realms*, which are similar to SGX enclaves.

Graviton [93] and Cure [7] are two TEE designs exploring protection from hardware outside the CPU; Graviton explores an enhanced GPU with TEE support, while Cure can bind peripherals to specific enclaves. CHERI-TrEE [92] adds TEE support to CHERI [98], a capability architecture. Like CHERI, the design of CHERI-TrEE is formalized in Sail, and implemented on both an open source RISC-V processor and and on ARM Morello.

9 Exploits against memory security

Even if the mechanisms used to ensure memory safety are sound, microarchitectural design of CPUs and of memory modules can lead to exploitable vulnerabilities, as can defects in programs, libraries, and supervisor software, which erode memory security. We discuss such vulnerabilities here.

9.1 Bugs

Programs that fail to check bounds when reading from or writing data into an array can allow for vulnerabilities. Writing data beyond an array on the stack (a buffer overflow) can be exploited to "smash the stack" [63]. Typical exploits override the return pointer on the stack, and insert code which is then used (code injection).

In modern systems, the stack is not executable, so code injection is not possible. Still, arbitrary code execution is basically possible with a technique called *return-oriented programming* [80]. A large enough code base can provide the appropriate *gadgets* (useful, short instruction sequences) for arbitrary execution, including for shellcode (turning a privileged process into a shell to execute arbitrary commands).

Safe programming languages such as Java and Go include runtime bounds checking into the language, but these features have overhead and are not included in languages such as C. Even though these kinds of memory vulnerabilities have been well known and well studied for decades, applications are still frequently vulnerable to them. A buffer over-read vulnerability, that is, failing to check bounds on an array read, led to Heartbleed [19] in 2014, one of the largest vulnerabilities in recent history with 24-55% of top HTTPS websites affected.

```
1  char array[256 * 0x1000];
2
3  void main(){
4      // Flush entirety of array out of cache.
5      clflush_all(array);
6
7      // Perform Meltdown with arbitrary kernel
       memory.
8      char *pt = (char *)0xDEADBEEF;
9      meltdown(pt);
10
11     // Note that the exception generated on line
       19 must be handled such that the next
       statement is executed.
12     // Calculate which page of array is accessible
        in the minimum amount of time.
13     // The page index will be the secret value.
14     char data = minimum_access_time(array);
15 }
16
17 void meltdown(char *pt){
18     // Direct access of data. Causes exception.
19     char data = *pt;
20     // Transiently access data, bringing the data-
       th page into cache
21     array[data * 0x1000] = 1;
22 }
```

Figure 1: Meltdown psuedocode to extract a single byte (at virtual address $0xDEADBEEF$) from memory using Flush+Reload.

9.2 Transient execution attacks

Meltdown [55] and Spectre [45] are the canonical examples of transient execution attacks. Modern CPUs use out-of-order execution, where instructions are executed speculatively. The CPU may execute instructions incorrectly if it mispredicts a branch or memory load; these incorrectly instructions executed *transiently* must be rolled back by the CPU such that the architecture (registers, memory) is not affected. Although the architecture cannot be affected by transiently executed instructions, microarchitectural structures such as the cache can be. The change in microarchitectural state can be transmitted via a covert channel such as Flush+Reload [103] or Prime+Probe [66].

Meltdown allows for the read of arbitrary protected (generally kernel) locations mapped in virtual memory. The attack directly accesses an arbitrary kernel memory location, and transiently transmits the value via cache. By the time an exception is generated from the invalid memory reference, the microarchitectural state is already modified in a measurable way. Psuedocode for Meltdown is provided in Figure 1.

Spectre is similar to Meltdown, but instead of directly accessing protected data, Spectre induces a victim (generally the kernel) to speculatively transmit secrets. This is done either through the speculative execution of a conditional branch (Spectre V1), induced through mistraining of the branch predictor, or through speculative execution of a return branch

(Spectre V2), through mistraining the Branch Target Buffer (BTB). Unlike Meltdown, Spectre functions on more processors (including AMD and ARM processors) and cannot be prevented by removing sensitive kernel information from an attacker's address space (i.e. KAISER [23]). Furthermore, a Spectre attack can occur from a sandboxed environment like JavaScript or eBPF.

Foreshadow-VMM, part of Foreshadow-NG [99], targets the hypervisor/co-residential virtual machines from a malicious virtual machine. An attacker can completely control guest physical addresses to transiently access arbitrary physical memory across the hypervisor boundary. Foreshadow-VMM claims to be "the first transient execution attack that fully escapes the virtual memory sandbox".

Microarchitectural data sampling (MDS) vulnerabilities leaks data across address spaces and other boundaries through speculatively reading of microarchitectural structures. This includes abusing the line-fill buffer (RIDL [89], ZombieLoad [79]), load port (RIDL), and the store buffer (Fallout [11]). While these attacks rely on an attacker to located on the same core as the victim, attacks like CrossTalk [73] can occur across the CPU by sampling the shared staging buffer.

While there are patchwork solutions for individual vulnerabilities, the root cause, speculation leaking through microarchitectural state, is not fully solved. Future CPUs must close microarchitectural covert and side channels to achieve true memory security. One research solution closing cache covert channels is DAWG [44], which dynamically allocates cache ways to processing entities, fully isolating cache hits and allowing for redundancy. Further research needs to be done to balance security with performance in real-world systems.

9.3 Rowhammer and physical attacks

Rowhammer [42,43] is a read-disturb vulnerability in DRAM chips. At increased density, DRAM cells can cause disturbance errors in neighboring cells; more specifically, activation of a row in DRAM can drain the capacity of adjacent rows, and with enough activations (*hammering*), bit flips can be triggered in victim cells. Moreover, the bits that flip due to rowhammer are repeatedly, predictably flipable. The bitflip rate can be improved greatly through using a double-sided rowhammer (attacking a victim row by repeatedly hammering both its adjacent rows) and a hammering pattern tailored to the particular DIMM.

Similar to the double-sided rowhammer technique is the "half-double" technique [46]. While rowhammer attacks usually focus on flipping bits in an adjacent row, it is possible for an hammered row to flip bits in rows at a greater distance. The farther an aggressor row is from a victim row, the less effective rowhammer is, but this can bypass certain defenses that only counter nearby rowhammering. Temperature is also a factor that can improve rowhammer capabilities [65]; rowhammer is more effective as temperature increases for most DRAM chips, although there are exceptions. The correlation between temperature and rowhammer effectiveness is actually strong enough where, given the ability to perform unprivileged remote execution, rowhammer can be used to spy on the temperature on DRAM [64], with an error less than $\pm 2.5°C$.

Row activation (A0)	Secret (S)
Unused	Sampling (A1)
Row activation (A2)	Secret (S)

Figure 2: RAMBleed memory layout. Each column is a page, and A0-A2 belong to the attacker, while S is the victim page.

Using rowhammer for practical attacks requires an intimate knowledge of the mapping of physical memory to channels, DIMM modules, ranks, banks, bank groups, and then rows and columns; two addresses can only been physically adjacent if they use the same channel, DIMM, rank, and bank. Tools such as DRAMA [69] have been developed to reverse engineer this mapping for Intel CPUs; these mapping are determined by the memory controller on the CPU, and different CPUs may implement mappings differently. For example, DRAMA does not work out of the box for AMD Zen [35] and had to be ported to achieve a comparable rowhammer success rate to Intel's CPUs.

RAMBleed [47] is a side channel attack that uses rowhammer as a primitive against confidentiality, rather than just integrity. The attacker must carefully align memory to achieve a layout like in Figure 2, with secret data sharing the same row as an attacker page. By hammering the attacker pages (A0, A2), bitflips can be triggered in the attacker's sampling page (A1).

With a double-sided rowhammer, a 0-1-0 rowhammer configuration, where the victim bit is charged and the hammering bits are not, is likely to trigger a bitflip to a zeroed state (0-0-0), while a 0-0-0 configuration will not trigger a bitflip in the victim bit. Similarly, a 1-0-1 configuration, with an uncharged victim bit and charged hammering bit, is likely to trigger a bitflip to a fully charged state (1-1-1), while a 1-1-1 will again not trigger a bitflip in the victim bit. RAMBleed uses this observation to its advantage, by examining the contents of the sampling page (A1) after rowhammering its row activation pages (A0, A2) using both patterns. Not all DRAM cells are flippable, so this must be repeated with different rows and columns to find all bits. Ideally, if the target bits are data-dependent, then some can be interfered.

RowPress [56] is a similar read-disturb vulnerability to rowhammer in DRAM chips, but is less understood. RowPress keeps a DRAM row open for an extended period of time, which can cause bit flips in adjacent rows. Few of the cells (fewer than 0.013%) affected by RowPress are also affected by rowhammer, suggesting that the root of the vulnerability is an entirely different physical phenomena. Similar to rowham-

mer, the effects of RowPress are stronger as DRAM modules scale to smaller node sizes, but dissimilarly, RowPress is less effective with greater temperature, and single-sided RowPress is more efficient than double-sided RowPress in some cases.

Cold boot attacks [26] are a physical attack against DRAM. Although the contents of DRAM is lost over time when power is lost, the contents are not immediately lost. Although the residual memory is lost pretty quickly, J.A. Halderman et al. found that it was possible to cool the physical DRAM to $-50°C$ using canned air duster products, significantly preserving residual memory. After a minute, 99.9% of the bits were recovered correctly in the DRAM modules tested.

9.4 Against TEEs

Trusted execution environments (TEEs) theoretically provide greater confidentiality and integrity for enclave processing entities, but their stronger threat model gives a greater toolbox (full kernel control) to hypothetical attackers.

Foreshadow [85] was the first attack against Intel SGX to allow arbitrary reading of enclave memory. Foreshadow unmaps enclave pages and then essentially uses the same flow as Meltdown, except relying on the data to be stored to be stored in the L1D cache. There are a number of tricks to keep or put enclave data into L1, especially with kernel privileges; in particular, the `eldu` instruction, used to swap in an enclave page, moves the entirety of an unencrypted page to L1, giving a privileged attacker a powerful primitive that can read the entire enclave memory space whenever desired. A privileged attacker can also single-step an enclave processing entity (using SGX-Step [87]), and access register values by targeting the SSA, where registers are stored after an enclave exit.

With a highly effective read primitive, Foreshadow also targeted Intel's architectural enclaves, which are built-in enclaves used to implement functionality too complicated to implement in microcode and are trusted. Foreshadow was able to successfully attack the Intel Launch Engine, extracting the launch key. Additionally, Foreshadow was able to extract the report key from the Intel Quoting Enclave, allowing for the signature of fake attestation quotes. While Foreshadow is chiefly an attack against confidentiality, undermining the Quoting Enclaves undermines the integrity guarantees for all SGX-based remote computation.

LVI (load value inject) [86] is a "reverse Meltdown" attack, which injects a victim with attacker-controlled data and then transiently forwards a secret to a gadget, following a similar "confused deputy" [27] style of attack similar to Spectre. In LVI, a microarchitectural buffer is filled with a malicious value, which is speculatively loaded from the victim, and then the injected value is used transiently by gadgets that leave a measurable microarchitectural trace. While the LVI methodology is not specific to Intel SGX, it is difficult to execute LVI attacks in a weaker threat model; similar to Foreshadow, LVI's proof-of-concept attacks use modification of the page table entries and single-step execution (again using SGX-Step) against enclave applications.

CacheOut [88, 90], is an MDS-style attack which similarly excels in targeting Intel SGX. Similar to Foreshadow, it can extract enclave contents even when the enclave is idle. CacheOut can sample reads by evicting the L1D cache and then sampling the data from from the line fill buffer (a microarchitectural cache). Again similar to Foreshadow, CacheOut is able to use instructions for swapping in and out enclave pages to force desired data into L1D and into the line fill buffer to read from any enclave, and again is able to use that to attack the Intel Quote Enclave to be able to sign fake attestation quotes, undermining SGX attestation.

Although Intel SGX is the most studied TEE, there have been vulnerabilities demonstrated in other TEEs as well, including ARM TrustZone [13] and AMD-SEV [10, 52, 60, 100]. In the long term, TEEs must address microarchitectural side channels and other defects for their alternative security model to be seen as sound.

10 Future

We identify seven challenges and avenues for future work:

1. **Comparing performance and other tradeoffs between protection mechanisms**. Research tends to compare mechanisms on a single CPU or simulator. How can comparisons become more generalizable?

2. **Exploring tradeoffs in implementation of existing and experimental mechanisms**. For example, despite decades of research, TLB design [22, 38] is still being questioned. Given a protection mechanism, what is the most efficient implementation? How does that change across workloads?

3. **Better synergizing software and hardware approaches**. Software-based protection mechanisms ("analytic protection") and hardware-based mechanisms are thought of as opposing approaches, but is it possible to reconcile the two?

    - Modern CPUs are designed with memory protection as a key part of the pipeline; for example, Singularity [1] was benchmarked on an AMD x86 processor, which is likely to be microarchitecturally optimized for virtual memory use. Could CPU architecture and/or microarchitecture be redesigned to more efficiently support software approaches?
    - Analytic protection shifts overhead to either start time or installation time. Could software verification be accelerated by hardware?

4. **Reimagining nested and redundant protection mechanisms**. Legacy systems layer protection mechanisms

like sandwich toppings to achieve nested protection; how much of this is needed, and how much is redundant?

- For example, x86 systems support segmentation, virtual memory, group protection (through Intel MPK), and TEEs (through Intel SGX or TDX) to achieve nested protection, a single application could rely on all of these. Which are redundant?
- Capability systems, such as CHERI [102], seem to be efficient for allowing nested protection. Can other mechanisms be reengineered for better nesting?
- On the other hand, redundancy in systems can sometimes be useful, such as for RMM [39], which combines a form of segmentation with page-based virtual memory, which are unnested. When might redundancy be useful, and when might it be a burden?

5. **Battling the incumbency advantage of legacy systems**. Existing systems rely on specific existing mechanisms and cannot easily be changed.
   - What kinds of systems are less hindered by legacy? For example, embedded and IoT systems are often fertile grounds for alternative designs. We identify serverless computing as a possible candidate; while traditional cloud computing offerings such as Amazon EC2 offer virtual machines of a specific operating system on a specific architecture, a Function as a Service (FaaS) platform may only need to support a certain language runtime.
   - How can applications relying on older mechanisms continue to be supported? Is it better to continue to support older mechanisms in hardware, or to emulate them? For example, Apple Rosetta is a proven effective binary translator for supporting x86 applications on ARM; can applications relying on older mechanisms be easily translated?
6. **Reimagining TEE design**. While TEEs like Intel SGX and ARM TrustZone are widely deployed, the TEE design space remains fluid.
   - Relating to redundancy, TEE solutions are usually implemented as another memory protection layer. To reduce such redundancy and overhead, how can TEEs be better integrated with existing protection mechanisms, or existing mechanisms with TEEs?
   - In many applications, such as ML-based ones, the majority of computation is done off on the CPU, such as on a GPU or TPU. How can TEE design be applied to other computational devices? Graviton [93] is an example of a TEE design for GPUs, but how can TEEs be integrated across a heterogeneous computing system?
7. **Closing microarchitectural side and covert channels**. As long as they remain open, memory security cannot be guaranteed.
   - Future microarchitecture designs must close these channels. How can these channels be closed with minimum sacrifices to performance? And while hardware with such vulnerabilities remain in deployment, what mitigations can be made in software or microcode?
   - How can disturbances such as rowhammer and RowPress be avoided? Rowhammer attacks are currently thwarted on commercial DDR5 DIMMs, but hardware manufacturers are not transparent about their solutions. Are DDR5 DIMMs truly secure, or do their defenses rely on security through obscurity?

11 Conclusion

We have have explored and discussed the mechanisms of memory protection.

12 Acknowledgements

This paper was produced for COMP 992: Master's (Non-Thesis) in the spring semester of 2024 at the University of North Carolina at Chapel Hill.

References

[1] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, 2006.

[2] Alibaba. Alibaba cloud released industry's first trusted and virtualized instance with support for sgx 2.0 and tpm. https://www.alibabacloud.com/blog/alibaba-cloud-released-industrys-first-trusted-and-v596821, October 2020.

[3] Alibaba. Alibaba cloud, elastic compute services, instance type families, overview. https://www.alibabacloud.com/help/doc-detail/60576.htm?spm=a2c63.p38356.b99.95.32ae1160CQKT0I, August 2023.

[4] AMD. Strengthening vm isolation with integrity protection and more. https://www.

amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[5] ARM. Unlocking the power of data with arm cca. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/unlocking-the-power-of-data-with-arm-cca.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[7] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. {CURE}: A security architecture with CUstomizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1073–1090, 2021.

[8] M Baker-Harvey, J Chase, H Levy, and E Lazowska. Opal: A single address space system for 64-bit architecture. In *IEEE Workshop on Workstation Operating Systems*, 1992.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[10] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, 2021.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[12] Nicholas P Carter, Stephen W Keckler, and William J Dally. Hardware support for fast capability-based addressing. *ACM SIGOPS Operating Systems Review*, 28(5):319–327, 1994.

[13] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[14] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, nov 1994.

[15] Hyok-Sung Choi and Hee-Chul Yun. Context switching and ipc performance comparison between uclinux and linux on the arm9 based processor. In *SAMSUNG Tech. Conf*, 2005.

[16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2024.

[17] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016. https://eprint.iacr.org/2016/086.

[18] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Pnacl: Portable native client executables. *Google White Paper*, 2010.

[19] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[20] eBPF. ebpf documentation. https://ebpf.io/what-is-ebpf/, April 2024.

[21] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, 2006.

[22] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A Bender, Abhishek Bhattacharjee, et al. Mosaic pages: Big TLB reach with small pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 433–448, 2023.

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9*, pages 161–176. Springer, 2017.

[24] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. {EPK}: Scalable and efficient memory protection keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 609–624, 2022.

[25] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[26] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[27] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[28] Trong-Thuc Hoang, Ckristian Duran, Duc-Thinh Nguyen-Hoang, Duc-Hung Le, Akira Tsukamoto, Kuniyasu Suzaki, and Cong-Kha Pham. Quick boot of trusted execution environment with hardware accelerators. *IEEE Access*, 8:74015–74023, 2020.

[29] Merle E Houdek, Frank G Soltis, and Roy L Hoffman. Ibm system/38 support for capability-based addressing. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 341–348, 1981.

[30] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[31] IBM. Ibm cloud data shield now generally available. https://www.ibm.com/blog/announcement/ibm-cloud-data-shield-now-generally-available/, April 2020.

[32] IBM. Provisioning a bare metal server with intel® software guard extension architecture. https://cloud.ibm.com/docs/bare-metal?topic=bare-metal-bm-server-provision-sgx, January 2023.

[33] IBM. *z/Architecture Principles of Operation*. IBM, 2024.

[34] Intel. Intel trust domain extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419, 2023.

[35] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. Zenhammer: Rowhammer attacks on AMD Zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 2024)*, 2024.

[36] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[37] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.

[38] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1178–1195, 2023.

[39] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. *ACM SIGARCH Computer Architecture News*, 43(3S):66–78, 2015.

[40] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[41] Tom Kilburn, R Bruce Payne, and David J Howarth. The atlas supervisor. In *Proceedings of the December 12-14, 1961, eastern joint computer conference: computers-key to total systems control*, pages 279–294, 1961.

[42] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651. IEEE, 2020.

[43] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[44] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[45] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[46] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering from the next row over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, 2022.

[47] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[48] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A Seshia, and Krste Asanovic. Cerberus: A formal approach to secure and efficient enclave memory sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1871–1885, 2022.

[49] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Ruby B. Lee. Precision architecture. *Computer*, 22(01):78–79, 1989.

[51] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[52] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Tlb poisoning attacks on amd secure encrypted virtualization. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 609–619, 2021.

[53] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188, 1993.

[54] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[56] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–18, 2023.

[57] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[58] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP 13)*. ACM, 2013.

[59] Microsoft. Dcsv3 and dcdsv3-series. https://learn.microsoft.com/en-us/azure/virtual-machines/dcv3-series, January 2023.

[60] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, pages 1–6, 2018.

[61] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39, 2020.

[62] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M Norton, Simon W Moore, Peter G Neumann, et al. Rigorous engineering for hardware security: Formal modelling and proof in the cheri design and implementation process. In *2020*

IEEE Symposium on Security and Privacy (SP), pages 1003–1020. IEEE, 2020.

[63] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[64] Lois Orosa, Ulrich Rührmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. Spyhammer: Using rowhammer to remotely spy on temperature. *arXiv preprint arXiv:2210.04084*, 2022.

[65] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S Kim, and Onur Mutlu. A deeper look into rowhammer's sensitivities: Experimental analysis of real dram chips and implications on future attacks and defenses. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1182–1197, 2021.

[66] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.

[67] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[68] David A Patterson and John L Hennessy. *Computer Organization and Design: 5th edition*. Morgan Kaufmann, 2014.

[69] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.

[70] Ludovic Pietre-Cambacédès and Claude Chaudet. The sema referential framework: Avoiding ambiguities in the terms "security" and "safety". *International Journal of Critical Infrastructure Protection*, 3(2):55–66, 2010.

[71] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.

[72] L. Pietre-Cambacédès and M. Bouissou. Cross-fertilization between safety and security engineering. *Reliability Engineering System Safety*, 110:110–126, 2013.

[73] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.

[74] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[75] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[76] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[77] Casper A. Scalzi, Alan G. Ganek, and Richard J. Schmalz. Enterprise systems architecture/370: An architecture for multiple virtual space access and authorization. *IBM systems journal*, 28(1):15–38, 1989.

[78] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. Composite enclaves: Towards disaggregated trusted execution. *arXiv preprint arXiv:2010.10416*, 2020.

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

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

[81] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[82] Alex Thomas, Stephan Kaminsky, Dayeol Lee, Dawn Song, and Krste Asanovic. Ertos: Enclaves in real-time operating systems. *Woodstock*, 2018.

[83] 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 (USENIX ATC 17)*, pages 645–658, 2017.

[84] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 788–803, 2021.

[85] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[86] 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)*, 2020.

[87] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[88] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[89] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[90] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *S&P*, May 2021.

[91] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.

[92] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. Cheri-tree: Flexible enclaves on capability machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1143–1159. IEEE, 2023.

[93] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.

[94] Victor A Vyssotsky, Fernando J Corbató, and Robert M Graham. Structure of the multics supervisor. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 203–212, 1965.

[95] Andrew Waterman, Krste Asanović, and John Hauser. The risc-v instruction set manual - volume ii: Privileged architecture - version 20211203. 2021.

[96] Robert NM Watson, Graeme Barnes, Jessica Clarke, Richard Grisenthwaite, Peter Sewell, Simon W Moore, and Jonathan Woodruff. Arm morello programme: Architectural security goals and known limitations. Technical report, University of Cambridge, Computer Laboratory, 2023.

[97] Robert NM Watson, Simon W Moore, Peter Sewell, and Peter G Neumann. An introduction to cheri. Technical report, University of Cambridge, Computer Laboratory, 2019.

[98] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[99] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[100] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 73–85, 2019.

[101] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019.

[102] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks

Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

[103] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[104] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[105] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4111–4128, 2022.

[106] ZDNet. Cloud security: Microsoft azure's sgx vms hit ga, google's shielded vm is now default. https://www.zdnet.com/article/cloud-security-microsoft-azures-sgx-vms-hit-ga-googles-shielded-vm-is-now-default/, April 2020.