Komodo: Using verification to disentangle secure-enclave hardware from software

Andrew Ferraiuolo, Cornell University
Andrew Baumann, Microsoft Research
Chris Hawblitzel, Microsoft Research
Bryan Parno, Carnegie Mellon University

ABSTRACT

Intel SGX promises powerful security: an arbitrary number of user-mode *enclaves* protected against physical attacks and privileged software adversaries. However, to achieve this, Intel extended the x86 architecture with an isolation mechanism approaching the complexity of an OS microkernel, implemented by an inscrutable mix of silicon and microcode. While hardware-based security can offer performance and features that are difficult or impossible to achieve in pure software, hardware-only solutions are difficult to update, either to patch security flaws or introduce new features.

Komodo illustrates an alternative approach to attested, on-demand, user-mode, concurrent isolated execution. We decouple the core hardware mechanisms such as memory encryption, address-space isolation and attestation from the management thereof, which Komodo delegates to a privileged software *monitor* that in turn implements enclaves. The monitor's correctness is ensured by a machine-checkable proof of both functional correctness and high-level security properties of enclave integrity and confidentiality. We show that the approach is practical and performant with a concrete implementation of a prototype in verified assembly code on ARM TrustZone. Our ultimate goal is to achieve security equivalent to or better than SGX while enabling deployment of new enclave features independently of CPU upgrades.

The Komodo specification, prototype implementation, and proofs are available at https://github.com/Microsoft/Komodo.

ACM Reference Format:
Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 19 pages.
https://doi.org/10.1145/3132747.3132782

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132782

1 INTRODUCTION

Software guard extensions (SGX) [43] is a set of new instructions in recent Intel CPUs for strong isolation of software *enclaves*. Compared to prior mainstream trusted computing hardware [2, 83], SGX includes strong physical security (memory encryption) and supports multiple enclaves: any number of enclaves may run concurrently without trusting a kernel or hypervisor. Nevertheless, SGX supports a familiar user-mode execution environment for enclave code, and remains compatible with existing OSes and hypervisors. In the short time since the SGX specification was published, a wide range of applications have been devised [e.g., 5, 8, 9, 14, 21, 42, 65, 68, 75], and competing vendors are developing similar features [48].

However, like most new hardware, SGX has been slow to deploy and evolve. SGX version 2, which enables dynamic memory management features essential to many enclave applications [5, 8, 42], was specified in October 2014 [43], but 3 years later there is still no announcement of CPUs that will implement it. The slow pace of hardware development is not new, but SGX is almost unique among CPU features in that there is no alternative—other architecture extensions boost performance, but it is usually possible to achieve equivalent functionality using legacy instructions. Moreover, as we detail in §2, the security of SGX rests on an opaque implementation in microcode and silicon [18], and already has known flaws, including security vulnerabilities [61] and "controlled-channel" attacks that exploit the OS's ability to induce and observe enclave page faults to leak enclave data [78, 88].

Given the slowing pace of silicon scaling [22, 79], it is dangerous to tie critical security features like enclaves to a hardware implementation. Each incremental change, for example to correct security flaws like controlled channels or even to add features like dynamic allocation, must wait for the deployment of new CPUs, and hence will take many years. Software is inherently more malleable than hardware, and an effective split between the two would allow for new features to be developed and flaws to be fixed independently of new hardware. Hardware vendors could simplify the complexity of their CPUs [7], reducing the validation effort and risk of bugs, and focus on improving the capacity and performance of hardware features such as memory encryption.

In this paper, we aim to disentangle the management of enclaves from underlying hardware mechanisms like protection, attestation, and memory encryption. Our core observation is that the security properties of SGX do not depend on its implementation entirely as a CPU feature. Similar isolation mechanisms have existed since the first multi-user systems; what distinguishes SGX is memory encryption, independence from a large untrusted OS, and the folklore intuition that hardware is more secure than software.

Komodo draws on ideas from SGX, but it replaces folklore with formal verification. Like SGX, it relies on hardware support for memory encryption and address-space isolation. However, instead of enclave-manipulation *instructions*, Komodo is implemented as a software reference monitor in verified assembly code. In fact, the design of Komodo mirrors an internal separation in SGX between core hardware and the instruction set. Since the monitor's only role is to protect enclaves, it is substantially simpler (and thus easier to evolve) than a full verified kernel [35, 49, 89]. It can also be readily updated. For example, after developing an initial version of Komodo modelled after SGXv1, we added dynamic memory management similar to SGXv2. We implemented and verified this update in approximately 6 person-months.

We describe Komodo in detail in §4. In addition to formalising its specification in §5, we prove in §6 that it protects the confidentiality and integrity of enclaves from both other enclaves and the untrusted OS. To our knowledge, no other secure-enclave implementations provide such formal guarantees. This proof holds for any correct implementation of the specification, including the ARM TrustZone-based prototype we describe in §7 and evaluate in §8.

Komodo does not support multi-processor execution—while the OS may run on multiple cores, the monitor and enclaves are restricted to a single core. Verification of low-level concurrent code remains challenging, recent progress notwithstanding [35], to which ARM's weak memory consistency [54] adds complexity. We leave this as future work.

The contributions of our work include:

- the identification of hardware requirements (§3) and a design for implementing enclaves in software (§4);
- a formal model of a substantial subset of ARMv7, including user and privileged modes, TrustZone, page tables, and exceptions (§5.1);
- a high-level formal functional specification of Komodo (§5.2), and a proof that it guarantees the confidentiality and integrity of enclave programs, formalised as noninterference (§6);
- a verified prototype (§7) and evaluation (§8) showing performance competitive with SGX;
- evidence for the hypothesis that verified software can evolve faster than hardware (§7.3).

2 BACKGROUND AND MOTIVATION

Prior research efforts have focused on using hardware mechanisms to protect critical software, even if privileged software (such as the OS or hypervisor) is malicious or compromised [15, 16, 19, 26, 50, 57, 67, 82]. SGX is the first commercial attempt at such protection, and its design is partly driven by pragmatic implementation constraints, such as compatibility with existing OS resource-management mechanisms, and avoiding changes to the processor's fast paths [18]. In this section, we provide a high-level overview of the SGX design to the extent that it informs our own.

The SGX implementation consists of three components: *(i)* encryption and integrity protection for a static region of physical memory implemented by an encryption engine within the memory controller, *(ii)* a set of instructions that allow the creation, manipulation and execution of enclaves, and *(iii)* changes to the processor's TLB miss and exception handling procedures that enforce enclave protections on access to the encrypted memory region.

The basic approach taken by SGX is to act as a *reference monitor* for actions taken by the untrusted OS and/or hypervisor (we refer to both as the "OS"). Although it has no direct access to encrypted pages, the OS allocates and maps them to enclaves, and although it cannot directly manipulate an enclave's register state, the OS chooses when, and on which CPUs, to execute enclave threads. OS management of enclave pages is performed indirectly via SGX instructions that manipulate the *enclave page cache map* (EPCM), a data structure maintained in encrypted memory and inaccessible to software. The EPCM stores metadata for every encrypted page, including its allocation state, type, owning enclave, permissions, and virtual address. Effectively a reverse map of encrypted pages, the EPCM is also consulted on a TLB miss to enforce enclave protections on memory—every page table mapping must be consistent with the EPCM.

Since they update a complex data structure, SGX instructions are complex. For example, besides basic argument validity, the `EADD` instruction must check that a page being added to an enclave is free and the enclave is in the correct state, before updating both the new page's EPCM entry and the enclave control structure. In doing this, it must guard against concurrent allocations of the page or modifications of the enclave. Other SGX instructions have even greater complexity—the process for validating a TLB shootdown before recycling EPC pages involves a series of epoch counters maintained in enclave control structures. These instructions are also not generally performance critical; indeed, based on Intel's patents [46, 60], Costan and Devadas [18, §2.14] claim they are implemented entirely in microcode.

Regardless of how the instructions are implemented, SGX's security rests on their correctness. Intel has published details for the formal verification of a high-level SGX model using an SMT solver [31, 44], and has verified the linearisability of a (different) model of concurrent SGX operations [52]. However, there does not appear to be any formal connection between these models and the SGX implementation, in which at least one security vulnerability has already been patched [61]. We can expect more bugs to be found, as they were in past CPU security technologies [86, 87].

Even assuming a correct implementation, SGX remains vulnerable to a variety of attacks. Classic side-channel attacks exploit shared hardware resources such as caches, branch predictors and TLBs, and are not addressed by SGX [13, 76]. Enclaves must instead use (often expensive) mitigations, such as avoiding secret-dependent memory accesses. Schwarz et al. [76] observed that existing hardware support for cache partitioning (Intel cache allocation technology [63]) could defeat such attacks, if only it were activated on enclave entry, but this is not a feature that SGX presently provides. One of our goals in decoupling higher-level enclave implementations from hardware is to permit such fixes to be deployed independently of hardware or architecture changes.

In addition to classic side channels, enclaves are vulnerable to new “controlled-channel” attacks in which the OS exploits its ability to induce and observe enclave page faults to deduce secrets [78, 88]. Mitigations exist, but (at a minimum) they require recompilation of enclave code, prevent use of dynamic paging, and carry a high performance cost [77, 78].

Overall, we argue that the limitations of SGX are systemic: because its entire specification is part of the x86 architecture, SGX is simply too slow to add features or respond to threats, and further, the limitations of hardware implementation also hobble its functionality [7]. In the following section, we describe how to disentangle enclave-supporting hardware from software, allowing them to evolve independently.

3 THREAT MODEL AND HARDWARE

3.1 Threat model

Like SGX, we seek to protect the confidentiality and integrity of user-mode code executing inside an enclave from an attacker who has full control over a platform’s privileged software (OS and hypervisor). To preserve generality across platforms, we consider two variants of this threat model, based on whether physical attacks on memory are in scope.

We assume a software attacker who controls privileged software. We also trust our verification tools (Dafny and Z3, described later in §5), assembler, linker, and bootloader. On the hardware side, we assume a correct CPU. The attacker may inject external interrupts, and attempt to interfere with I/O devices. If physical memory attacks are in scope, the attacker may access any RAM external to the CPU package. This includes bus snooping and cold-boot [36] attacks.

As with SGX, general side-channel attacks are out of scope. Hardware isolation technologies such as cache partitioning [19, 63] are required to defeat these in a practical manner for general-purpose code, and we anticipate that a future version of Komodo could enable them. Komodo is immune to controlled-channel attacks [88]; as our confidentiality proof (§6) ensures, the OS learns only the type of exception taken, and cannot induce an exception.

3.2 Hardware requirements

Our basic approach with Komodo is to implement a highly privileged program in verified assembly code; its role mirrors that of the enclave-management instructions in SGX: maintaining an EPCM-like “database” of secure pages by acting as a reference monitor for enclave manipulation and execution. In order to implement enclaves in software, we rely on four hardware primitives: isolated memory for monitor code/data and enclave pages, protected execution environments for both the privileged monitor and unprivileged enclaves, a root of trust for attestation, and a source of randomness.

Isolated memory. Komodo requires a region of physical memory whose confidentiality and integrity is protected by hardware. Our design is agnostic to the exact memory isolation mechanisms, which we expect will vary in different applications depending on the hardware threat boundary.

If physical attacks on memory are in scope, the hardware must include memory encryption and/or on-chip RAM. For example, SGX performs encryption and integrity protection of RAM. This offers strong protection against physical attacks, at the cost of limited size and a performance penalty for integrity protection [44]. Unfortunately Intel’s memory encryption engine is accessible only by SGX, so Komodo cannot make use of it. IBM SecureBlue [11] also includes memory encryption hardware, but with limited public information it is difficult to be sure whether IBM’s design is suitable. AMD recently published a proposal for hardware memory encryption configurable by privileged software [48]. Since this proposal lacks integrity protection, it would scale to large memories, at the cost of weaker security.

As an alternative to encryption, some “systems on a chip” (SoCs) include scratchpad RAM, which is protected against most physical attacks by virtue of its on-chip location [41, 64]. Although size-limited, this may be effective for secure embedded applications, since it avoids the complexity and energy/performance overhead of encryption [17].

Finally, if physical attacks on memory are out of scope (as is common in many pre-SGX applications), all that is needed in hardware is an IOMMU-like filter to partition RAM and prevent access by unprivileged software or devices.

Privileged environment for monitor. Komodo's monitor must be protected from malicious privileged code on the platform, including the OS and hypervisor. This requirement includes a secure control transfer mechanism between monitor code and normal execution, protection against unprogrammed control transfers within monitor code or access to its intermediate states (e.g. registers). Note that we do not assume another (costly) layer of memory translation—the monitor requires access to isolated memory, but this can take the form of a direct physical mapping or restricted segment which is otherwise inaccessible.

A variety of architectures already include such an environment. In SGX, it is effectively provided by the microcode engine: the processor guarantees that the execution of microcode sequences is uninterruptible and protected from interference. Other examples of a super-privileged restricted environment include DEC Alpha PALcode [24] and RISC-V machine mode [85]. Our prototype leverages the secure monitor mode of ARM TrustZone, which we describe below.

Enclave execution environment. Komodo must be able to control the execution of enclaves, protecting itself and other code on the platform against a malicious enclave. A typical user mode suffices, if it can also be protected from the OS.

One unique feature of SGX enclaves that we do not attempt to emulate is their virtual memory layout. SGX enclaves execute within the context of an untrusted user process—memory outside the enclave region transparently reflects the address space of its parent process. Instead, Komodo executes each enclave in its own virtual address space, and memory shared with an untrusted process must be established by explicit mappings. In our experience, there are very few applications that require uncontrolled ad-hoc access from an enclave to an untrusted parent process. Not only would such a feature require extra hardware support, it also arguably reduces overall enclave security, since most of the enclave's virtual memory space is untrusted, as opposed to our model where the entire virtual space is trusted except for well-defined shared mappings.

Remote attestation. Komodo requires a hardware-backed root of trust for remote attestation. Much like a TPM-based trust chain, our expectation is that either hardware or an early bootloader would attest to a secure hash of the monitor (the monitor in turn implements enclave attestation). Such attestation mechanisms are widely supported [70, 83].

Random number source. Finally, we require a hardware-backed cryptographically secure source of randomness. This could be an instruction (like the x86 `RDRAND`/`RDSEED` [45]) or a hardware device accessible to the monitor.

We note that Sanctum [19] (a modified RISC-V CPU) meets all the above requirements, with the caveat that physical attacks are out of scope. Indeed, Sanctum shares much with Komodo, with several key differences: it relies on significant hardware modifications (per-enclave page tables); it calls for a far more elaborate attestation mechanism than what Komodo employs, and its security guarantees rely on the correctness of an unverified monitor implemented in 5k lines of C++. In this paper, we formally verify the correctness of such a security monitor implemented on ARM TrustZone. We discuss Sanctum further in §10.

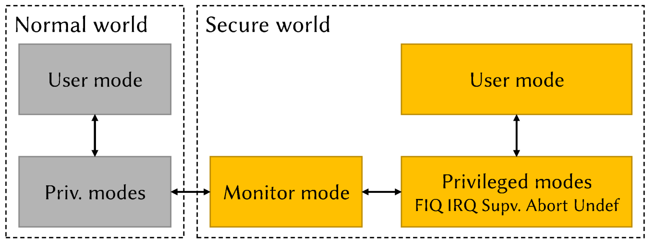

Figure 1: ARM TrustZone modes and worlds

3.3 ARM TrustZone

We prototype Komodo on TrustZone [2], a security technology in many ARM CPUs. TrustZone extends the core architecture and peripherals such as memory controllers.

As shown in Figure 1, a TrustZone processor runs in one of two *worlds*: normal world, where a regular OS and applications run, and secure world. Each world contains both user mode and five different privileged modes; the latter are used by different exceptions (e.g., page faults enter a different mode from system calls) but are all equally privileged.[1] Secure world has a sixth privileged *monitor mode* which is used to switch worlds: a *secure monitor call* (SMC) instruction in normal world can cause an exception taken in monitor mode.

Some system control registers are banked, with one copy for each world. These include the MMU configuration and page-table base registers, so a world switch may enter a different address space. TLB and cache entries are also tagged according to world.I/O devices may also participate in TrustZone. A secure bit in page-table entries is used to tag memory references issued by secure-world code, and devices such as a TrustZone-aware memory-controller or IOMMU may use this to prevent normal-world access to secure-world memory or devices [1, 4]. The accessibility of memory between worlds depends on the specific platform configuration; for example, on some SoCs the secure world has exclusive access to an isolated region of memory [17, 70].

We chose to prototype Komodo on TrustZone, because its secure world satisfies our requirements for executing both

[1] Some ARM CPUs include virtualisation extensions, which add a hypervisor mode to normal world with an additional level of translation. Komodo offers the same functionality and security, regardless of whether these are present.

Table 1: OS and enclave APIs to Komodo monitor

Secure monitor calls (SMCs, from OS):	
`GetPhysPages()`→`int npages`	Return number of secure pages
`InitAddrspace(PageNr asPg, PageNr l1ptPg)`	Create address space (enclave) given two empty pages
`InitThread(PageNr asPg, PageNr threadPg, void *entry)`	Create thread
`InitL2PTable(PageNr asPg, PageNr l2ptPg, int l1index)`	Allocate 2nd-level page table
`AllocSpare(PageNr asPg, PageNr sparePg)`	Allocate spare page to given address space
`MapSecure(PageNr asPg, PageNr dataPg, Mapping va, InsecurePg content)`	Allocate a data page, mapped at address and perms in `va`
`MapInsecure(PageNr asPg, Mapping va, InsecurePg target)`	Map an insecure (shared) page at address and perms in `va`
`Finalise(PageNr asPg)`	Mark enclave final, compute measurement and allow execution
`Enter(PageNr thread, int arg1, int arg2, int arg3)`→`int retval`	Enter enclave on an idle thread, passing parameters
`Resume(PageNr thread)`→`int retval`	Resume execution of a previously suspended thread
`Stop(PageNr asPg)`	Mark enclave stopped, permitting deallocation
`Remove(PageNr pg)`	Deallocate any page in a stopped enclave or a spare page in any enclave
Supervisor calls (SVCs, from enclave):	
`GetRandom()`→`u32 val`	Hardware source of secure random numbers
`Attest(u32 data[8])`→`u32 mac[8]`	Construct attestation of enclave's identity
`Verify(u32 data[8], u32 measure[8], u32 mac[8])`→`bool ok`	Check validity of an attestation
`InitL2PTable(PageNr sparePg, int l1index)`	Create 2nd-level page table from a spare page
`MapData(PageNr sparePg, Mapping vaddr)`	Map spare page as zero-filled data page at address and perms in `vaddr`
`UnmapData(PageNr dataPg, Mapping vaddr)`	Unmap data page, turning it back into a spare page
`Exit(int retval)`	Return control to the OS

the monitor and enclave code: enclaves run in secure user world, using a page table established by Komodo running in secure privileged modes (mostly monitor mode). In addition, the ARM ecosystem presently lacks enclave-like features; existing TrustZone applications either assume that all secure-world code is trusted [6, 30, 47] or rely on language-based isolation for "trustlets" [74].

4 KOMODO DESIGN AND API

The Komodo monitor builds on the hardware described in the previous section to implement enclaves. Like SGX, it manages a region of isolated physical memory, making *secure pages* available for constructing enclaves, and enabling enclave execution while protecting enclave-internal state. The API calls in Table 1 mirror SGX operations, but rather than distinct instructions, they are invoked as monitor calls.

Page types and enclave construction. The monitor must ensure consistent use of secure pages, preventing, for example, double-mapping between distrusting enclaves. Komodo tracks the state of secure pages using a data structure we term the *PageDB*. This is roughly equivalent to the EPCM of SGX; for every secure page, it stores the page's allocation state, and, if allocated, its type and a reference to the owning enclave. The monitor does no allocations of its own—the OS must choose pages it knows to be free, or API calls fail.

Each allocated page has one of six types: address space, thread, first-level page table, second-level page table, data page, and spare page. An enclave consists of an address space with at least one thread. To begin constructing an enclave, the OS calls `InitAddrspace` to create a new (empty) address space. However, before it can populate the address space, the OS must allocate a second-level page table using `InitL2PTable`. Komodo's API encodes a two-level hierarchical page table with a granularity chosen to reflect ARM's hardware page-table format. The OS may allocate as many second-level tables as it wishes, but for a mapping call to succeed at a given virtual address the relevant page table must exist.

The OS may then populate the address space by mapping one or more *secure* and *insecure data pages*. Secure data pages are located within the isolated memory, and they are private to an enclave. Their initial contents, virtual address and page permissions are included in the attestation measurement described below. Insecure pages are not protected by hardware isolation, and are therefore accessible to the untrusted OS. These may be mapped to the enclave to facilitate untrusted communication channels with the OS or between enclaves.

For the enclave to be executable, the OS must also create a thread, specifying its entry-point address. The enclave is then explicitly *finalised*, preventing the uncontrolled mapping of further pages/threads, before execution.

Enclave execution. A newly created thread belonging to a finalised enclave may be executed by invoking `Enter`, which causes the monitor to switch into secure-world user mode and begin execution at the thread's entry-point address with the given parameters. The enclave thread then executes until an exception occurs: either an interrupt, or an enclave-triggered exception such as a page fault, undefined instruction, or a system call. On an interrupt, the monitor saves register context in the thread page before reporting the interrupt to the OS. The thread context is marked as *entered*, to prevent a suspended thread from being re-entered.

Enclaves may also invoke the monitor via the *supervisor call* (SVC) instruction. One such call, `Exit`, serves to explicitly pass a result back to the OS. In this case, the enclave's registers are not saved, permitting it to be re-entered.

If the enclave takes an exception, the thread simply exits with an error code (but no other information, to avoid side-channel leaks). Unlike SGX [88], the OS cannot induce enclave page faults. Our design is thus secure and also sufficient for simple enclaves that do not emulate illegal instructions nor handle page faults; we anticipate adding a mechanism for enclaves to handle their own faults in future work.

Attestation. Komodo adopts a minimalist attestation design, inspired by previous work on local attestations [40, 58]. This important design choice makes it feasible for us to formally verify the attestation mechanism, which would be challenging with more complex schemes [19].

As the enclave is being constructed, the monitor constructs a hash of the sequence of page allocation calls and their parameters; specifically: *(i)* the enclave virtual address, permissions and initial contents of each secure page; and *(ii)* the entry point of every thread. Like SGX, the OS is free to construct enclaves arbitrarily, but any change in an enclave's layout will be reflected in the hash. When the enclave is finalised, this hash becomes the enclave's immutable *measurement* for attestation purposes.

Like SGX, Komodo implements local (same machine) attestation as a monitor primitive, and defers remote attestation to a trusted enclave (that we have yet to implement). A Komodo attestation is a message authentication code (MAC) using a secret key generated at boot from a cryptographically secure source of randomness. The MAC is computed over *(i)* the attesting enclave's measurement, and *(ii)* enclave-provided data, which may be used to bind a public key-pair to the enclave and hence bootstrap encrypted communication with code outside the enclave [56]. The monitor provides calls for enclaves to create and verify attestations.

Dynamic allocation. Komodo includes support for dynamic management of enclave memory, comparable to SGXv2 [43]. At any time, the OS may allocate *spare pages* to an enclave using the `AllocSpare` monitor call. These do not alter the enclave's measurement, since they do not become accessible until the enclave issues either a `MapData` or an `InitL2PTable` SVC to map them as data pages or page tables. The enclave may also unmap data pages (turning them back into spare pages), and the OS may reclaim spare pages. As a result, the OS may infer that spare pages have been allocated (since attempts to remove them will fail), but it cannot tell whether the enclave has used them as data or page-table pages. This is in contrast to SGXv2, where the OS remains in control of the type, address and permissions of all dynamic allocations. We are not aware of attacks on this side-channel, but nevertheless saw no reason to mirror it.

Deallocation. Before an enclave's pages can be freed, the OS must call `Stop`. This prevents further execution, and permits the use of `Remove` to deallocate secure pages. The address space is reference counted, and must be removed last.

5 SPECIFICATION

We specify and verify Komodo using Dafny [51], a general-purpose verification language. This section describes our trusted Dafny specifications of ARM assembly language (§5.1) and of Komodo's overall correctness (§5.2). We increase our confidence in the Komodo specification by proving several high-level lemmas: that it maintains consistency invariants on page state (described in §5.2) and that it guarantees enclave confidentiality and integrity (§6). Dafny checks the validity of these lemmas with the help of the Z3 SMT solver [23].

5.1 ARM machine model

Our hardware specification, written in Dafny, covers a subset of the ARMv7 architecture [3]. We model execution as a series of machine states, where a state includes everything visible about a machine (e.g. registers and memory). Our model includes core registers R0–R12, stack pointer (SP), link register (LR), portions of the current and saved program status registers (CPSR and SPSRs), privilege modes, control flow, interrupts, and exceptions. We model the semantics of 25 instructions, including integer and bitwise arithmetic, and access to memory and control registers.

At present, our model must be fully trusted, but this could be avoided by proving its correctness against another formal model for ARM [29, 71, 72]. To help ensure trustworthiness, we adopt the methodology termed *idiomatic specification* by Ironclad [38]: we specify only the features that a Komodo implementation needs, and write the specifications such that the implementation cannot trigger other unspecified behaviours. For example, a verified implementation cannot execute unspecified instructions.

To simplify reasoning about control flow, we do not explicitly model the program counter (PC) register. Instead, our model encodes a limited form of structured control flow: if statements, while loops, and subprocedure calls. This avoids the verification burden of reasoning about PC updates and the effects of control-flow instructions like conditional branches. However, we do model the side-effects of two control transfers crucial to the correctness of Komodo: the branch from privileged code to user-mode (a `MOVS PC, LR` instruction, which branches to the link register and updates the mode and flags), and the switch back into privileged mode when an exception occurs, which preserves the pre-exception PC value in LR. The Komodo specification can therefore use its value to refer implicitly to the PC at the time of an exception.

The 32-bit ARM architecture includes a *register banking* feature that we also model: the SP, LR and SPSR registers are banked according to the current mode—user-mode accesses to SP refer to a concrete register SP_usr, whereas monitor-mode code accesses SP_mon, etc. We model all the banked registers, with the exception of those banked only in FIQ mode (which is not needed).

Memory. In designing our memory model, we made several design decisions that proved crucial to building a scalable proof. For example, our machine state models memory as a mapping from word-aligned addresses to 32-bit values; reasoning only about aligned memory accesses simplifies proofs, since accesses to distinct addresses are independent.

We do not directly model virtual memory translation—load and store instructions directly manipulate the contents of the memory map at the address specified by their operands. This allows us to define address validity solely based on the effective address value, not on the overall machine state. This expedites verification since the prover readily sees that validity is not affected by state changes.

The Komodo specification (§5.2) ensures valid address regions for the monitor's stack, global variables, and secure/insecure enclave pages. §7.2 later describes how these are provided by the bootloader using a static page table.

User-mode execution. Besides the privilege separation offered by ARM user-mode, Komodo's design does not constrain the code that can be run in an enclave. To model enclave execution, we might therefore need to model every permissible user-mode instruction, along with its effects on the machine state. This would imply specifying a large number of instructions, along with a more complete model of machine state and virtual memory translation. However, we do not seek to verify the code that runs in enclaves, and such a model would needlessly bloat our trusted computing base (TCB) and increase verification times.

Instead, we model only the aspects of user-mode execution necessary to reason about Komodo's correctness, including a limited view of virtual memory: when user code executes, it havocs (trashes) all user-mode registers and all user-writable pages before taking an exception. Writable pages are found by walking page tables starting from the page-table base register, and translated into the monitor's memory map. Essentially, we specify that the monitor executes in a 1:1 mapping of physical memory at some fixed virtual offset (established by the bootloader, per §7.2). We model the effects of user-mode code by translating writable pages into the memory map using the offset.

As another example of idiomatic specification, ARM supports many page table formats, but we model only one: 4 kB "small" pages in the short descriptor format. If an unrecognised page-table entry is encountered, the model says nothing about the results of user execution—this forces the implementation to prove that its page tables meet the specification in order to reason about states after user-mode execution.

As well as page tables, we also model TLB consistency. Executing a TLB flush instruction marks the TLB as consistent. Loading the page-table base register, or executing a store to an address in either the first-level or any second-level page table, marks the TLB as inconsistent. This gives the implementation freedom to either simply flush the TLB whenever consistency is required, or else to prove that its stores did not modify the page table. For simplicity, we model only flushes of the entire TLB (not tag- or region-based flushes).

Exceptions. Our strategy for dealing with exceptions is primarily to avoid them. For example, preconditions on the load and store instructions prevent the possibility of a page or alignment fault in verified code. However, we must model the CPU interrupt-enable flags (we describe why later, in §7.2). Our core specification for instruction evaluation states that if interrupts are enabled, and if an interrupt (non-deterministically) occurs, then the instruction executes only after first running the interrupt handler, which is modelled as an arbitrary implementation-specific predicate. This forces a correct implementation to prove either that interrupts remain disabled, or else to implement an interrupt handler and prove that the pre-conditions of any instruction executed with interrupts enabled are satisfied by the handler's postcondition.

Limitations. Our model precludes the implementation from using many features irrelevant to Komodo, including I/O devices, most co-processor registers, floating point and vector state, and unaligned memory accesses. Since we do not support multi-core execution, we do not model caches or memory consistency.

5.2 Komodo specification

In addition to the ARM model, we must also trust our high-level behavioural specification of the Komodo monitor, also written in Dafny. We increase our confidence in this specification by proving that it maintains internal consistency invariants (described below) and guarantees high-level security properties (described in §6). At the core of this specification is an abstract representation of the PageDB: a map from page numbers to entries, each of which has one of the six types described earlier in §4.

The PageDB representation abstracts away implementation details irrelevant to most of the specification; for example: page tables are represented as entries in an abstract data type, and the enclave measurement established for attestation is represented by an unbounded sequence of words. The contents of memory and registers are of course significant when an enclave executes, so the specification includes a predicate describing the contents of enclave-visible registers, memory, and page tables at the time of execution. The implementation is free to choose its own in-memory representation of the PageDB, as long as it can prove that when an enclave executes, the contents of registers and virtual memory match the abstract PageDB.

The top level of our specification is a predicate describing the SMC handler. It relates the concrete machine and abstract PageDB states just after taking an SMC exception from the OS, to the final states (`s'` and `d'`) just prior to returning:

```
predicate smchandler(s: state, d: PageDb,
                     s': state, d': PageDb)
```

Of all the monitor calls in Table 1, only two involve enclave execution: `Enter` and `Resume`. We specify the body of the rest as pure functions that, given an input PageDB and call parameters, compute an error/success code and resulting PageDB. The top-level `smchandler` predicate holds if the resulting PageDB and error code match the appropriate function (based on the call number and argument registers), and also that certain invariants hold across every SMC: non-volatile registers are preserved, other non-return registers are zeroed (to prevent information leaks), insecure memory is invariant, and we return in the correct mode.

The specifications for `Enter` and `Resume` are also modelled as predicates relating two states and PageDBs. The specifications for these calls forces the implementation to enter user-mode (which it can only satisfy by executing `MOVS PC, LR`) from a highly constrained state. Specifically, the page-table base register must be loaded with the address of the enclave's page table, its representation in memory must match the abstract page table encoded in the PageDB, and the TLB must be consistent. The contents of secure data pages must equal those in the PageDB (either the contents at the time the enclave was created, or as modified by enclave execution). The user-visible registers must be loaded from the PageDB: for entry, the PC is set to the entry-point and other registers are zeroed; for resume, the user-visible registers are restored from context saved in the thread's PageDB entry.

By constraining the concrete machine state only at the time of entry to user-mode, we maintain a significant degree of implementation freedom. For example, an implementation may maintain its data structures in any format it chooses, as long as it can prove that the user-mode execution environment satisfies the specification.

The specifications of SVCs from an enclave are logically nested inside the definition of `Enter` and `Resume`. After user-mode execution, an exception is taken, and the specification then determines the results of the call and final PageDB based on the exceptional state. If the exception taken was for a non-`Exit` SVC that returns to the enclave, then the specification describes how to compute the results of the call, and return to executing the enclave (using a recursively defined predicate). All other exceptions update the PageDB and return results from the SMC handler; for example, the PageDB's data pages are updated to reflect any changes made by the enclave, and if an interrupt was taken, the user-mode context must be saved in the PageDB and the thread marked as entered.

A valid PageDB satisfies invariants guaranteeing internal consistency: e.g., reference counts are correct, internal references (including page table pointers) are to pages of the correct type belonging to the same address space, and all leaf pages mapped in a page table are either insecure pages or data pages allocated to the same address space. To increase our confidence in the specification, we prove that each SMC and SVC preserves the PageDB invariants. These invariants then form the basis of our security proofs in the next section.

6 PROVING SECURITY

We formally prove that the Komodo specification described above protects the confidentiality and integrity of enclave code and data from other software on the machine. Because the implementation is verified to satisfy the specification, these security proofs extend to the concrete Komodo code as well. In particular, we prove that an enclave's contents cannot be modified by any software other than that enclave, and that an enclave's contents do not leak to other enclaves, the OS, or other non-enclave code, unless the enclave itself chooses to leak them either directly (e.g., by writing to insecure memory) or indirectly (e.g., via the pattern of insecure memory addresses to which the enclave chooses to write).

More formally, we establish Komodo's security properties by proving that enclaves are noninterfering [32] with an adversary who controls the OS and colludes with an enclave. Modulo a limited set of declassification operations (§6.2), we establish two separate results for confidentiality and integrity which are respectively: *(i)* enclave state is noninterfering with state observable outside the enclave, and *(ii)* state which can be influenced by software outside the enclave is noninterfering with enclave state. Our model of enclave state is sufficient to show that the confidentiality and integrity of both enclave data and execution are preserved.

From the confidentiality perspective, noninterference requires that all adversarially observable outputs during the execution of the system are determined purely by the adversarially supplied inputs. In other words, public outputs are never influenced by secrets. This is a strong end-to-end security property: it precludes secrets from affecting public outputs even indirectly through control flow. Integrity is dual to confidentiality [10], and requires that trusted outputs are purely determined by trusted inputs.

By modelling a strong adversary who controls both the OS and an enclave, our results generalise to simpler attackers such as an OS or enclave acting alone. In short, we prove formally that enclave secrets do not leak to, and that enclaves cannot be influenced by, any software other than the monitor.

We do *not* aim to prove that enclaves use Komodo correctly. An enclave may leak information directly through its return code, writes to insecure memory, or the use of SVCs for dynamic allocation (a side-channel we formally characterise in §6.2). It may also leak indirectly through hardware side-channels (e.g., via cache effects). The enclave's integrity may be compromised, for example, if it fails to sanitise values passed as parameters or read from insecure memory. Work complementary to ours provides security guarantees for SGX enclaves [33, 80, 81], and could be adapted for Komodo.

6.1 Specification

Komodo executes on a single core, so attackers on that core (including potentially malicious enclaves) cannot observe machine state concurrently with Komodo. However, as described in §5.1, we do permit concurrent execution of the OS on a different core. The OS cannot observe registers or secure memory, but it may access insecure memory concurrently with Komodo execution. Our hardware model prevents information about Komodo's execution from leaking to insecure memory by prohibiting the implementation from writing to insecure memory (it has no reason to do so).

We take advantage of the fact that attackers can only make observations while they are executing to simplify our proofs. It is sufficient for us to reason about the states that transition between different entities in the system (normal world, the Komodo monitor, and enclaves), because it is impossible for adversaries to observe intermediate states (e.g., while a non-malicious enclave is executing). Reasoning about states at these transition points is simpler than reasoning about entire execution traces.

The transition points in our system are at the beginning and end of SMCs and enclave execution. We prove our noninterference theorems for each monitor call, and as we discuss, this is sufficient for guaranteeing security at the start and end of enclave execution. By carefully structuring the pre- and post-conditions so we make no assumptions about the initial state that do not also hold of the final state, we ensure that our result generalises to an infinite sequence of SMCs.

We consider states (s, d) which comprise a concrete machine state, s, and an abstract PageDB, d, such that s is an implementation of d. Our confidentiality result roughly states that publicly observable outputs depend solely on publicly observable inputs. Our integrity result states that trusted outputs depend solely on trusted inputs.

We formalise both results with a relation, $\approx_L$, that characterises the observational power of some observer L—two states are related by $\approx_L$ if the states appear the same to L. The definition of $\approx_L$ depends on the observer under consideration. For the proof of confidentiality, the observer is an adversary, *adv*, who models an OS colluding with an enclave. For the integrity proof, the observer is a trusted enclave, *enc*.

Address-space pages in the PageDB are linked to all pages belonging to an enclave. Therefore, *enc* is an address-space page that identifies an enclave, and the definition of $\approx_{enc}$ characterises the observational power of that enclave. To define $\approx_{enc}$, we rely on an auxiliary relation $=_{enc}$ that relates PageDB entries and characterises pages that look the same to the enclave when they are outside its address space *enc*:

Definition 1 (Weak-equivalence of pages $=_{enc}$). *PageDB entries e_1, e_2 are related by $e_1 =_{enc} e_2$ iff:*

$$\begin{aligned}&(e_1.\text{DataPage?} \wedge e_2.\text{DataPage?})\\ \vee&(e_1.\text{SparePage?} \wedge e_2.\text{SparePage?})\\ \vee&(e_1.\text{Thread?} \wedge e_2.\text{Thread?} \wedge e_1.\text{entered} = e_2.\text{entered})\\ \vee&((e_1.\text{L1PTable?} \vee e_1.\text{L2PTable?} \vee e_1.\text{Addrspace?}) \wedge e_1 = e_2)\end{aligned}$$

where e_1.DataPage? denotes that e_1 has the type data page. In other words, an enclave cannot observe data page contents or thread context unless those pages belong to it.

Definition 2 (Observational equivalence $\approx_{enc}$). *Let $d[i]$ denote page i in PageDB d. Let $\mathcal{F}(d)$ denote the set of pages i such that $d[i]$ is not allocated. Let $\mathcal{A}_{enc}(d)$ denote the set of pages i such that $d[i]$ belongs to the address space enc in d. Then two PageDB states, d_1, d_2, are observationally equivalent from the perspective of an enclave enc, written $d_1 \approx_{enc} d_2$ iff:*

$$\begin{aligned}&\mathcal{F}(d_1) = \mathcal{F}(d_2) \wedge \mathcal{A}_{enc}(d_1) = \mathcal{A}_{enc}(d_2)\\ &\wedge \forall i \notin \mathcal{A}_{enc}(d_1) \,.\, d_1[i] =_{enc} d_2[i]\\ &\wedge \forall j \in \mathcal{A}_{enc}(d_1) \,.\, d_1[j] = d_2[j]\end{aligned}$$

To characterise the observational power of a malicious OS colluding with an enclave, we also define $\approx_{adv}$. Since this adversary has more observational power than an enclave alone, for states to be related by $\approx_{adv}$, they must also be related by $\approx_{enc}$, where *enc* represents a malicious enclave. The additional requirements on $\approx_{adv}$ further restrict the set of pages that look equivalent to the adversary, and hence characterise the observational power of the OS. In particular, the OS adversary can directly observe the registers to which it has access and the entire insecure memory. Hence, two states are related by $\approx_{adv}$ if in addition to the requirements imposed by $\approx_{enc}$, all of the following are the same for both states: the general-purpose registers, the banked registers (excluding monitor mode), and the insecure memory.

We formalise our noninterference properties as:

Theorem 6.1 (Noninterference). *Let execution of the SMC handler beginning in state (s, d) and returning in state (s', d') be denoted as* smchandler(s, d, s', d'). *Then,*

$$\forall (s_1, d_1), (s_2, d_2), (s'_1, d'_1), (s'_2, d'_2) \, . \, (s_1, d_1) \approx_L (s_2, d_2)$$
$$\wedge \ \texttt{smchandler}(s_1, d_1, s'_1, d'_1) \wedge \texttt{smchandler}(s_2, d_2, s'_2, d'_2)$$
$$\implies (s'_1, d'_1) \approx_L (s'_2, d'_2)$$

We prove a relaxation of this theorem, and we discuss the way in which our result relaxes this theorem in Section 6.2

For the proof of integrity, $\approx_{enc}$ is used, and *enc* denotes a trusted enclave. For the proof of confidentiality, $\approx_{adv}$ is used, and the *enc* implicit in the definition of $\approx_{adv}$ is a malicious enclave. Although the definition of $\approx_{enc}$ is concise and does not directly constrain the concrete state, both proofs of noninterference ensure strong guarantees when combined with our correctness specification. Proving these noninterference theorems entails proving that the contents of registers and all memory reachable by the enclave at both the start and end of its execution are determined purely by PageDB entries allocated to the enclave prior to the `Enter` or `Resume` call.

6.2 Declassification

Enclaves release a small amount of information to the OS during normal execution: the type of exception or interrupt that ends enclave execution, the return value passed to `Exit` and the fact that an exit call was made. Enclaves that use dynamic memory allocation also leak through a side channel, since the OS can observe which spare pages have been allocated and which data pages have been freed by the enclave during execution. As is conventional for any practical system that enforces noninterference, we rely on *declassification* to permit the communication described above, that would otherwise be precluded by the information flow policy. Our effort to precisely control what information is declassified most closely resembles the delimited release model [73].

Declassification is incorporated into our proofs through four axioms which each have preconditions that precisely control the state transitions during which they can be used. The axiom for releasing the exception or interrupt taken by the enclave can be invoked to reason about states immediately following the execution of enclave code in user-space. For example, the SVC call number is stored in register R0, and the axiom for releasing it can only be invoked to reason about the state after taking an SVC exception from user-mode—this prevents, for example, leaking the enclave's R0 value when an interrupt occurs. Other declassification axioms are predicated on certain SVC calls being invoked. The dynamic memory management calls release information about the pages that are allocated or deallocated by the enclave. The OS can distinguish these by design, because it is permitted to `Remove` deallocated pages.

6.3 Proofs and non-determinism

Our proofs use bisimulation; we reason about two executions beginning from initial states that are related by $\approx_L$ and our proof goal is to show that the final states are also related by $\approx_L$. Our proof is then structured into smaller bisimulation proofs about each monitor call and each SVC. One exception is that we cannot prove that output states are related by $\approx_{enc}$ if the call is `Remove(enc)`, as the relation is undefined for an observer whose only page was just removed.

Because $\approx_{enc}$ is used to characterise both the trusted state during the integrity proof and the observational power of a malicious enclave during the confidentiality proof, many lemmas are re-used between the two proofs.

The `Enter` and `Resume` proofs are the most complex since they involve enclave execution. In order to satisfy the PageDB refinement relation, handling the case where both executions are of the observer enclave requires that the enclave's secure pages and register context are updated in the same way. However, enclave execution is *not* deterministic. We do not know what the enclave code will do; we can merely model what portions of the state it might affect.

Our specification models the non-determinism by updating each part of the enclave state with an uninterpreted function specific to the updated state. Each function takes at least two inputs: *(i)* all of the user-visible state including the general-purpose registers, the PC on entry to the enclave, and all of memory accessible with the current page table and *(ii)* a source of non-determinism modelled as an unknown integer seed. For both noninterference proofs, we require that the seeds in the initial states are the same for successful executions of the observer enclave. This allows us to prove that updates happen deterministically. However, in order to do so, we must prove that the user-visible state on entry to the enclave and on updates to the seed are equivalent. The registers, insecure memory pages, and secure memory pages that Komodo presents to the enclave must be purely determined by that enclave's pages in the PageDB.

The confidentiality proof must show that secret enclave state is not leaked to the adversary through the registers which it can observe during monitor calls or through insecure pages of memory. For calls involving enclave execution, we must show that updates to the registers and insecure memory at the end of the call are purely determined by public state. The enclave is permitted to write to insecure memory. However, correct enclave code should not write anything

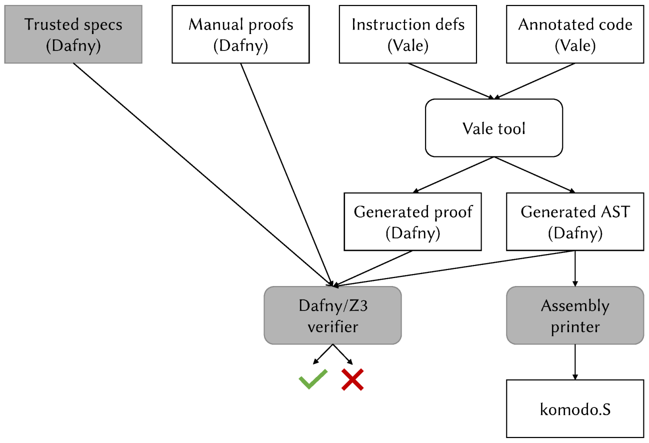

Figure 2: Verification/implementation tools. Shaded boxes indicate trusted components.

```
procedure MRS_STR(operand src:sreg, operand base:reg,
                  operand ofs:word, out operand tmp:reg)
  requires
    SaneState(this) && ValidMem(base + ofs);
    @tmp != OSP && @tmp != @base && @tmp != @ofs;
  modifies mem;
  ensures
    SaneState(this);
    MemContents(this.m, base + ofs) == src;
    MemPreservingExcept(old(this), this, base + ofs,
                        base + ofs + WORDSIZE);
{
  MRS(tmp, src);
  STR(tmp, base, ofs);
}
```

Listing 1: Vale procedure to store a banked register

secret to insecure memory. To model the fact that insecure memory is public, enclave updates to it are handled differently from secure memory: they are still non-deterministic, but do not depend on user state.

7 IMPLEMENTATION

7.1 Vale language

Figure 2 shows the tools used to implement and verify Komodo. Our implementation uses the Vale [12] programming language. Vale programs, shown in the "annotated code" box, consist of assembly language instructions together with annotations, such as preconditions, postconditions, and loop invariants, that describe the behaviour of the instructions.

Listing 1 shows a simple Vale procedure used in Komodo. This two-instruction procedure copies a banked register to a general-purpose register and then stores it to memory at a given base address and offset. Its preconditions (the `requires` clause) include a valid memory address and register allocation—the general-purpose register `tmp` cannot alias the base address nor stack pointer. Its postconditions (the `ensures` clause) guarantee to the caller that only the `tmp` register is modified, and that memory is invariant except the single word at `base+ofs` which equals the banked register.

The Vale tool generates two intermediate Dafny-language objects: an abstract-syntax-tree (AST) representation of the instructions, and a purported proof about the behaviour of the instructions (e.g., that the instructions ensure the postconditions in the annotations). If the annotations or code are wrong, this proof will be invalid.

Since Komodo consists of low-level assembly code, we do not rely heavily on Dafny's features for executable code. In fact, the only executable Dafny code is a simple pretty-printer that turns the instruction ASTs into GNU assembly format. This printer, along with Dafny and the trusted specifications, constitute Komodo's trusted computing base for verification. The Vale tool is not part of the trusted computing base; a bug in Vale could create incorrect ASTs or invalid proofs, and Dafny would reject such ASTs or proofs for failing to correctly fulfil the specifications.

Each Vale procedure is encoded as an AST that when executed takes an input state, and (if the preconditions are satisfied) produces an output state. Following earlier work with Vale [12], we do not model labels and jumps directly, but rather define structured control constructs: conditionals, loops, and subprocedures. The trusted printer then turns these into labels and jumps. All subprocedure calls are inlined; for most of Komodo the complexity of stack-based call/return is unwarranted, but we plan to add this for large procedures (e.g., hash functions) in future work.

7.2 Implementation details

Hardware platform. Our prototype runs on a Raspberry Pi 2, which is widely available and includes a TrustZone-capable CPU and hardware random-number generator, but lacks support for isolating secure-world memory or performing hardware-backed attestation. Instead, we simply assume the existence of a statically configured isolated memory region and hardware-derived attestation secret, and rely on the bootloader to provide them. This means that our prototype unfortunately offers no practical security; however, porting it to an ARM platform that included these features would alter neither its performance nor the proof, since both features affect only boot-time configuration.

Exception-handler procedures. One of the biggest challenges faced by the implementation is a mismatch between the linear control-flow modelled by Vale, which automates

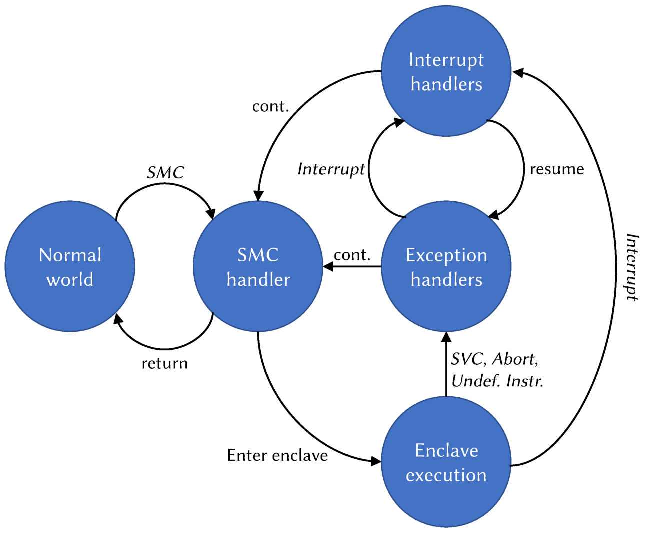

Figure 3: State transitions and top-level procedures

verification of procedures with a single starting and ending state, and the exception-driven style of execution inherent in kernel code. Komodo lacks a single top-level procedure; instead it is implemented by handlers invoked from a table of hardware exception vectors. These include the SMC and SVC handlers invoked for API calls by the OS and enclave respectively, handlers for ARM's two different kinds of interrupt ("FIQ" and "IRQ"), and exceptions for undefined instructions and data aborts (page faults).

As Figure 3 illustrates, these handlers form a state machine which is nested inside the top-level SMC handler and constrained by its specification. We model the state transitions explicitly in Dafny, proving that whenever an exception may occur, its preconditions are satisfied and its handler establishes the conditions required by the next state. We then rely on trusted wrappers in the assembly printer to link these procedures together; for example, the "instruction" used to begin enclave execution prints as `MOVS PC, LR` followed by a label. The exception handlers are all printed with a jump to the label at the end. To ensure this control flow is sound, we prove in a Dafny lemma that the state after user execution starting from the above instruction satisfies the preconditions of any of the exception handlers, and that their postconditions satisfy the postconditions of the user execution instruction.

Interrupts. Whenever possible, the monitor executes with interrupts disabled. This allows us to reason about most instructions in isolation, which is a reasonable tradeoff since all operations are bounded-time (the longest-running monitor call, `MapSecure`, initialises and hashes a single page of memory). Interrupts are enabled when executing an enclave, and disabled automatically when taking either an SMC or interrupt exception. However, it is possible to take an interrupt after entering the handler for a system call, abort, or undefined instruction. Our exception handlers immediately disable interrupts, but there is an unavoidable single-instruction window in which a nested exception may occur.

The interrupt handler's behaviour depends on the prior state of the system. If the interrupt was taken in user-mode, it locates the current thread page, and saves the user-mode register context before branching to the continuation. However, if the interrupt was taken in privileged mode, it simply sets a flag to record that the interrupt occurred, before restoring execution with registers and memory preserved and interrupts disabled. To keep the code simple, the Vale procedures for all instructions in our machine model require that interrupts are disabled, with one exception: the instruction used to disable interrupts which is written such that the interrupt handler may have been executed prior to instruction completion. Again, we prove in Dafny that these pre- and post-conditions match between the different procedures, so the only gaps in our trusted computing base are the jumps emitted by the printer at the end of each interrupt handler.

Enclave execution. The implementation of `Enter` and `Resume` must execute the enclave an unbounded number of times, until either an `Exit` SVC or an exception occurs. The natural way to implement this (as in an early unverified prototype), is to have the SMC handler push its PC on the stack for later return prior to dispatching the enclave. When the SVC handler is invoked, it can handle the SVC and return directly to the enclave, unless the call is `Exit`, in which case it branches back to the SMC handler's return path. This arrangement, however, is impractical to verify in our model of linear control flow. As we mentioned above, we print a single branch after the instruction that initiates user-mode execution, so that any exception handler can unambiguously return to it. This in turn requires that user-space entry occurs at only one point, leading to a loop of the form:

```
while (!done) {
  MOVS_PC_LR(); // enter user-mode,
  // handle exception, branch back
}
```

However, because at the point of the *done* test every user-visible register is live (and even testing a global variable would require a spare register), we were forced to use the least-significant-bit of the monitor's SP register as the *done* flag. Our insight here is that polluting part of the implementation with ugly (but verified) code is preferable to added complexity in our execution model (and thus our TCB).

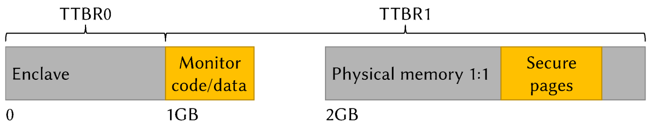

Figure 4: Secure-world virtual memory map

Memory map. Figure 4 shows the secure-world virtual memory layout. We make use of an ARM architectural feature to decouple the monitor’s page table from the one used by enclave address spaces—the enclave’s page table is loaded into the TTBR0 control register, which is configured to map only the first 1GB of virtual address space (the upper address limit for enclaves). The remaining address space is mapped by a separate static page table in TTBR1 created by the bootloader. This latter region is restricted to privileged modes, and includes mappings for the monitor’s code and data (stack and global variables), and a large direct mapping to physical memory. This in turn includes the isolated memory allocated by the bootloader, and is where enclaves live.

The only regions of its virtual address space that the monitor directly loads and stores are its stack, global variables, and secure pages. Additionally, it reads from the OS insecure pages when initialising enclave pages. As described in §5.1, this allows us to reason about the contents of memory while largely ignoring address translation.

Attestation and cryptography. Komodo borrows the core ARM SHA-256 implementation from previous work with Vale [12]. As a result, we benefit from good hashing performance, since the code mirrors the optimised SHA routines from OpenSSL, and a proof of freedom from digital (cache and timing) side channels. We extended the prior implementation to a complete SHA-256 in ARM assembly, including initialisation and finalisation routines which previously relied on a high-level Dafny implementation. We also implemented a SHA-256-based MAC for generating and checking attestations. In our implementation, we leverage a precondition that Komodo only invokes SHA on block-aligned data to significantly simplify reasoning about padding.

7.3 Code size and verification effort

Table 2 shows a breakdown of the number of physical lines of code, excluding comments and whitespace, in Komodo. Specification lines include all trusted Dafny code: our machine model and Komodo functional specification (§5) along with the helper Dafny libraries used to define them (common data types, bitwise functions, etc.), cryptographic algorithms, noninterference properties (§6) and finally the pretty-printer for assembly output. Implementation lines are assembly instructions, procedure calls and control-flow we write in Vale.

Table 2: Line counts

Component	Spec	Impl	Proof	Assembly
	(source lines of code)			(instructs.)
ARM model	1,174	112	985	
Dafny libraries	588		806	
SHA-256, SHA-HMAC	250	415	3,200	170
Komodo common	775	358	3,078	136
SMC handler	591	1,082	4,493	284
SVC handler	204	612	2,509	233
Other exceptions	39	131	940	52
Noninterference	175		2,644	
Assembly printer	650			
Total	4,446	2,710	18,655	875

Proof lines are annotations that help the verifier, such as pre- and post-conditions, loop invariants, assertions, and lemmas.

The table also reports the number of ARM instructions appearing at the Vale source level. Due to our use of structured control flow, this does not include comparisons and branches, which are added by the pretty-printer. After printing, which also adds labels and symbol declarations, and performs inline expansion of procedure calls, the verified prototype is emitted as a 26,800-line assembly file. This could be substantially reduced if Vale supported function call/return.

The complete end-to-end verification of Komodo takes 4 core-hours. However, it is highly parallel, and supports distributed verification. Furthermore, the typical developer works on one procedure or lemma at a time, and most of these take well under a minute to verify.

Excluding the core SHA implementation that we inherited from prior work [12], we spent a total of about 2 person-years specifying and implementing Komodo. We began with the specification of the ARM model, then specified and implemented a simplified version of Komodo using static memory management modelled on SGXv1. Building this first version took around 1.5 person-years, including a steep learning curve for the primary developers who were unfamiliar with the verification tools (at the time under active development).

In the process of developing this first version, we iterated through several phases in which we refactored the core definitions (e.g., the ARM machine model) to make them more amenable to automated verification, and to model progressively more complex features such as exceptions and interrupts. Each such iteration required revising existing proofs to maintain new invariants. As with any engineering project, and in line with the experience of the seL4 developers [49, §7.4], the more pervasive the model or invariants changed the more work it was to re-establish the proofs.

For example, we discovered that reasoning about word alignment was excessively costly for Z3, and was indirectly causing verification timeouts. We therefore changed the core definition of a word-aligned address, which was originally:

```
const WORDSIZE:int := 4;
predicate WordAligned(x:int) { x % WORDSIZE == 0 }
```

In the new definition, `WordAligned` is an *opaque* function (for which the prover doesn't see the definition), and we prove selected lemmas about it, for example that addition of two word-aligned values, or the computation of a word-offset (i.e., $x+n\times$WORDSIZE), always results in a word-aligned value. This required changing all our procedures that perform memory access or manipulate addresses to use the new declarations and lemmas, leading to a week's worth of semi-mechanical refactoring but resulting in much improved proof stability. Given improvements like this, we estimate that repeating the effort to rebuild the first version now with stable tools and specifications would require much less than 1 year.

We then extended the spec and implementation with dynamic memory management; this totalled 6 person-months of extra work, including 3 person-weeks for updates to the noninterference proofs. This work included major changes to the specification, such as modelling TLB consistency, weakening various PageDB invariants to reason about spare pages, and permitting non-trivial changes to the PageDB in an SVC handler. In the process of implementing the new SVCs, we also refactored much of the implementation of the core page-table management code to permit its use in either SMC or SVC handler contexts. This required reasoning about flexible register allocations. For example, the procedure to write a page-table entry previously used a hard-coded register allocation with each operand passed in a specific machine register; this simplifies the verifier's job (and thus permits verifying longer procedures with fewer annotations) because it can trivially see that modifications of one operand do not affect the others. However, to permit calling the procedure in different contexts, this procedure and many others like it were changed to take arbitrary register operands in the style of Listing 1 (but with more operands, and thus many more preconditions for disjointedness).

Although most of the memory-management code changed, other significant portions of the implementation did not, including most of the enclave entry/resume path and top-level SMC handler. Thanks to our use of automated verification tools, the proofs for these were largely unaffected, even by changes in the core specification.

8 EVALUATION

8.1 Microbenchmarks

We tested our prototype on a Raspberry Pi 2 Model B with a 900MHz ARM Cortex-A7 CPU. To do this, we implemented a simple bootloader that loads the monitor in secure world, setting up its memory map and exception vectors. The bootloader was implemented in unverified C and assembly for expedience, but we could use the same approach to also specify and verify it (it runs to completion without taking exceptions, so it is much simpler than the monitor). The bootloader also reserves a configurable amount of RAM as secure memory, before switching to normal world to boot Linux. As mentioned in §7.2, the hardware lacks support for memory isolation, so our prototype is not secure against a malicious OS, but it performs equivalently. Once Linux boots, a kernel driver issues SMCs to create and run enclaves.

Table 3: Microbenchmark results on Raspberry Pi

Operation	Notes	Cycles
`GetPhysPages`	Null SMC	123
`Enter + Exit`	Full enclave crossing (call & return)	738
`Enter` only	(no return)	496
`Resume` only	(no return)	625
`Attest`	Construct attestation	12,411
`Verify`	Verify attestation	13,373
`AllocSpare`	Dynamic allocation	217
`MapData`	Dynamic allocation	5,826

We performed the microbenchmarks reported in Table 3. The prototype monitor is entirely unoptimised. It conservatively saves and restores every non-volatile register—a needless cost for trivial SMCs like `GetPhysPages`. On enclave entry, it also saves and restores every banked register, although some are known to be preserved, and flushes the TLB, although this could be avoided for repeated invocation of the same enclave (even for distinct enclaves with the use of TLB tags). These are all optimisations that we aim to add, but only after proving their correctness; for example, by proving in Dafny a lemma that the banked registers for FIQ and IRQ modes are unchanged by enclave execution, we can call that lemma in our implementation rather than needlessly saving and restoring the registers.

Despite the lack of optimisations, Komodo's performance compares favourably to SGX. Orenbach et al. [66, §2.2] report EENTER and EEXIT latencies of about 3,800 and 3,300 cycles respectively, or 7,100 cycles for a full enclave crossing. Of course, the x86 runs at a higher clock rate (2GHz vs. 900MHz) and includes memory encryption, but the Komodo result represents an order of magnitude improvement. We can only speculate about the reasons, but there is clearly no inherent penalty for implementing enclaves in software.

8.2 Notary enclave

To test Komodo with a real enclave, and help convince ourselves of the completeness of its API, we ported the trusted

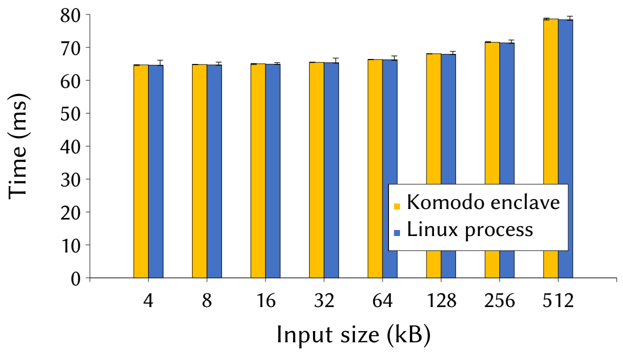

Figure 5: Notary performance

notary application from Ironclad [38, §5.1]. The notary assigns logical timestamps to documents so they can be conclusively ordered. We reimplemented the notary as a standalone 3700-line C program compiled for the Komodo enclave API. When first entered, it constructs an RSA key pair, initialises a monotonic counter, and constructs and returns an attestation of its initial state. On subsequent calls, it hashes the provided document with the current value of the counter and signs it with its RSA key before incrementing the counter and returning the signature. The notary's total memory footprint is 145 kB. Performance measurements (Figure 5) show that, since its execution is dominated by CPU-intensive hashing and signing, the notary performs equivalently in an enclave to a native Linux process.

9 DISCUSSION

9.1 Lessons learned

A small code base is no substitute for verification. Before embarking on the verification of Komodo, we had previously implemented an unverified version in C and assembly, as a way to gain familiarity with the TrustZone design. The unimplemented monitor comprised only about 650 lines of C and 300 lines of assembler (it did not support attestation), and yet it still contained critical security bugs which came to light in the process of specifying and implementing Komodo.

For example, `InitAddrspace` takes two page numbers. The unverified implementation checked that both were free, before proceeding to allocate them and initialise the address space. Only after writing the specification for this call and failing to prove that it maintained PageDB invariants did we discover that we hadn't considered the case when the two arguments are the same page.

As a more subtle problem, when checking the validity of insecure memory pages, we had failed to account for the fact that the monitor's text and data exist in direct-map physical as well as virtual memory (see Figure 4). To check whether an insecure physical address passed to the monitor for `MapSecure` or `MapInsecure` is valid, it is *not* sufficient merely to check that it does not refer to secure pages; instead, it must also avoid any of the monitor's own pages. We discovered this discrepancy in the process of formalising our model of virtual memory—an example of how the process of writing a specification forces clarity.

Trusted components require extra diligence. In verifying any system, one must choose what to trust and what to verify, and against what specification. We discovered bugs in our code when we ran it; unsurprisingly, they were all in trusted code or under-specified portions of our system [28]. For example:

- a bug in the assembly printer caused all instructions intended to operate on *banked* SPSR registers to instead use the current mode's SPSR;
- we were missing barriers (DSB and ISB instructions) when accessing certain control registers;
- inconsistencies in the configuration of caches and page attributes between the bootloader, monitor and Linux driver resulted in incoherent caches for normal-world and secure-world views of shared pages.

Our conclusion is that while verification has great value in eliminating entire classes of errors, it cannot prevent a developer from making any unwarranted assumptions, at least not without a complete and correct formal specification of CPU behaviour. Besides the obvious issues such as memory consistency (which we explicitly chose not to model), modern CPUs include a seemingly endless number of control registers that alter system behaviour and could lead to violations of our ARM model. In this respect, we are encouraged by recent progress on connecting a formal specification of the ARM architecture to its implementation [71, 72].

Opportunities remain to improve verification tools. Past work on verifying systems software [38, 39] extended Dafny with features for information hiding and modular reasoning such as opaque functions. More recent improvements to Dafny have included support for constants, bitvectors, and refinement types (such as 32-bit unsigned integers). We benefited from all these improvements, but still found that Dafny struggled with complex systems such as Komodo.

The most frustrating recurring problem was proof instability. For simple lemmas, Dafny will either report success or a concrete failure, such as an assertion violation. However, as proof complexity increases, solver time may increase exponentially. This happens easily in Komodo wherever we are reasoning about procedures with many instructions (and thus many state transitions) or complex specifications. To avoid an endless search, Dafny implements a time limit before reporting failure. Timeouts are challenging to debug, because the solver generally fails to provide useful feedback. Instead, the developer must simplify the proof and/or add assertions to reduce the complexity. However, even once fixed, the proof may easily timeout again due to minor perturbations. Worse, minor changes can trigger timeouts in seemingly unrelated proofs. Proofs involving bitwise operations or the modulo operator proved particularly unstable. The only reliable way to remove timeouts in a given piece of code was generally to refactor it into smaller subprocedures with their own explicit pre- and post-conditions, but this leads to inelegant and hard-to-maintain code.

9.2 Future work

While we believe Komodo represents a significant step towards practicality, and many applications (e.g., in embedded systems) are already in reach, more work remains.

Dispatcher interface. Komodo is not vulnerable to controlled-channel attacks [78, 88] merely by virtue of the fact that it does not yet support demand-paging of enclave memory. We hope to evolve our current thread-based interface where enclave threads are either started anew, or saved/restored transparently into a LibOS-style *dispatcher* interface [55] with explicit user-mode upcalls to resume a thread or report an exception. This will permit the use of enclave self-paging to manage memory [37, 66], without exposing page faults to the untrusted OS.

Multi-core support. Komodo's biggest remaining limitation is undoubtedly multi-core support. There are several avenues to close this gap, but the simplest is a single shared lock around all monitor activities, which would preserve the sequential (Floyd-Hoare) reasoning used in our current proofs. Experience with microkernels even suggests that this may not unduly harm performance [25].

10 RELATED WORK

Hardware. A wide range of systems have used hardware to isolate sensitive code from an untrusted OS [15, 16, 26, 27, 50, 53, 67, 82]. These vary in their resilience to hardware attacks, size of the software trusted computing base, and granularity of protection. However, to our knowledge none has a formal specification nor a proof of security. SGX [43, 59] is unique mainly because of its implementation in x86.

The most closely related system to Komodo is Sanctum [19]. Like Komodo, Sanctum consists of simple hardware extensions to support a trusted security monitor that in turn manages and protects enclaves. Unlike Sanctum, the Komodo prototype runs on readily available hardware (ARM TrustZone) and includes machine-checked proofs of both functional correctness and noninterference properties that guarantee enclave integrity and confidentiality.

Sanctum and Komodo also differ in their approach to attestation. Sanctum computes measurements in the monitor, but delegates attestation to a privileged signing enclave to avoid side-channel leaks involving the attestation key. We instead implement local attestation directly in the monitor. Our attestation algorithm (HMAC-SHA256) is data-independent in its address trace, and we could prove this using techniques previously developed for Vale [12]. We feel that this is a good verification-complexity tradeoff compared to specifying and implementing the IPC mechanisms that Sanctum uses to support attestation in an enclave.

Software. Other systems have sought to use commodity hardware to provide enclave-like isolated execution environments in software. However, the majority of these did not provide formal guarantees [e.g., 57, 58, 69].

Our verification methodology builds on the tools and techniques developed in Verve [89], Ironclad [38] and IronFleet [39], however the most closely related verified systems are the kernels seL4 [49], CertiKOS [34, 35] and überSpark [84]. Komodo might even be viewed as a microkernel with an unusual API, but this comparison has its limits: Komodo does not handle interrupts nor support device drivers, not even for the system clock or interrupt controller; it does not implement a scheduler nor perform resource management; it lacks an IPC mechanism. It does, however, perform attestation and run alongside an untrusted OS. Like Komodo, seL4 and CertiKOS also benefit from proofs of security properties based on noninterference [20, 62]; seL4's formalisation is more complex since it includes general-purpose IPC. Ultimately the advantage that Komodo gains from simplicity and automated verification tools is adaptability: we can rapidly evolve Komodo while preserving its security guarantees, whereas complex kernels like seL4 and CertiKOS represent substantially more human effort.

11 CONCLUSION

Komodo is the first formally verified implementation of an SGX-like enclave isolation mechanism. Its design decouples enclave hardware primitives from security-critical but formally verified software, enabling independent evolution of the two. We used noninterference to prove high-level guarantees of confidentiality and integrity, we showed that the approach is feasible, that Komodo can evolve more quickly than SGX, and that it can even outperform SGX.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd George Candea for valuable feedback, and Rustan Leino and Jay Lorch for help with Dafny.

This work was supported in part by the National Science Foundation and VMware under Grant No. CNS-1700521.

REFERENCES

[1] *PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147) Technical Overview*. ARM Limited, Nov. 2004. Ref. DTO 0015A.

[2] *Building a Secure System using TrustZone Technology*. ARM Limited, Apr. 2009. Ref. PRD29-GENC-009492C.

[3] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. ARM Limited, May 2014. Ref. DDI 0406C.c.

[4] *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*. ARM Limited, Feb. 2014. Ref. DDI 0504C.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 689–703, 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[6] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *21st ACM Conference on Computer and Communications Security*, pages 90–102, 2014. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660350.

[7] A. 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.

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

[9] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *EuroSys Conference*, pages 222–237, 2017. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064213.

[10] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, 1977.

[11] R. Boivie. SecureBlue++: CPU support for secure execution. Technical Report RC25287, IBM Research, May 2012. URL http://researcher.watson.ibm.com/researcher/view_group.php?id=7253.

[12] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium*, Aug. 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[13] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Aug. 2017. URL https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[14] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *17th International Middleware Conference*, pages 14:1–14:13, 2016. ISBN 978-1-4503-4300-8. doi: 10.1145/2988336.2988350.

[15] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *16th IEEE International Symposium on High-Performance Computer Architecture*, Jan. 2010. doi: 10.1109/HPCA.2010.5416657.

[16] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: a hardware-software approach to full system security. In *International Conference on Supercomputing*, pages 108–119, 2011. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995914.

[17] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189, 2015. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694380.

[18] V. Costan and S. Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, Feb. 2016. http://eprint.iacr.org/2016/086.

[19] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, pages 857–874, Aug. 2016. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[20] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 648–664, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908100.

[21] S. Crosby. Using Intel SGX to protect on-line credentials, Aug. 2016. URL https://blogs.bromium.com/2016/08/09/using-intel-sgx-to-protect-on-line-credentials/.

[22] I. Cutress. Intel's 'Tick-Tock' seemingly dead, becomes 'Process-Architecture-Optimization'. *AnandTech*, Mar. 2016. URL http://www.anandtech.com/show/10183.

[23] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Mar. 2008. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24.

[24] *PALcode for Alpha Microprocessors System Design Guide*. Digital Equipment Corp., May 1996. Order No. EC-QFGLC-TE.

[25] K. Elphinstone, A. Zarrabi, A. Danis, Y. Shen, and G. Heiser. An evaluation of coarse-grained locking for multicore microkernels. *CoRR*, abs/1609.08372, Oct. 2016. URL http://arxiv.org/abs/1609.08372.

[26] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 190–202, 2014. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.25.

[27] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *7th ACM Workshop on Scalable Trusted Computing*, pages 3–8, 2012. ISBN 978-1-4503-1662-0. doi: 10.1145/2382536.2382540.

[28] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys Conference*, Apr. 2017. doi: 10.1145/3064176.3064183.

[29] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st International Conference on Interactive Theorem Proving*, pages 243–258, July 2010. ISBN 978-3-642-14052-5. doi: 10.1007/978-3-642-14052-5_18.

[30] *GlobalPlatform Device Technology TEE System Architecture v1.1*. GlobalPlatform, Jan. 2017. Ref. GPD_SPE_009.

[31] A. Goel, S. Krstić, R. Leslie, and M. R. Tuttle. SMT-based system verification with DVF. In *10th International Workshop on Satisfiability Modulo Theories*, pages 32–43, 2012. URL http://smt2012.loria.fr/paper2.pdf.

[32] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982. doi: 10.1109/SP.1982.10014.

[33] A. Gollamudi and S. Chong. Automatic enforcement of expressive security policies using enclaves. In *2016 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA 2016, pages 494–513, 2016. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984002.

[34] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608, 2015. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676975.

[35] R. Gu, Z. Shao, H. Chen, X. N. Wo, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016.

[36] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium*, pages 45–60, July 2008. URL https://www.usenix.org/legacy/event/sec08/tech/full_papers/halderman/halderman.pdf.

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

[38] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 165–181, Oct. 2014. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel.

[39] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *25th ACM Symposium on Operating Systems Principles*, pages 1–17, 2015. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815428.

[40] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the web. In *10th USENIX Symposium on Networked Systems Design and Implementation*, pages 529–545, 2013. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/howell.

[41] G. Hunt, G. Letey, and E. Nightingale. The seven properties of highly secure devices. Technical Report MSR-TR-2017-16, Microsoft Research, Mar. 2017. URL https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/.

[42] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 533–549, 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[43] *Software Guard Extensions Programming Reference*. Intel Corp., Oct. 2014. Ref. #329298-002 https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

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

[45] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp., Dec. 2016. Ref. #325462-061US.

[46] S. P. Johnson, U. R. Savagaonkar, V. R. Scarlata, F. X. McKeen, and C. V. Rozas. Technique for supporting multiple secure enclaves, Dec. 2010. US Patent 8,972,746.

[47] U. Kanonov and A. Wool. Secure containers in Android: The Samsung KNOX case study. In *6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–12, 2016. ISBN 978-1-4503-4564-4. doi: 10.1145/2994459.2994470.

[48] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, Apr. 2016.

[49] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb. 2014. ISSN 0734-2071. doi: 10.1145/2560537.

[50] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *32nd International Symposium on Computer Architecture*, pages 2–13, 2005. ISBN 0-7695-2270-X. doi: 10.1109/ISCA.2005.14.

[51] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, pages 348–370, Apr. 2010. ISBN 978-3-642-17511-4. doi: 10.1007/978-3-642-17511-4_20.

[52] R. Leslie-Hurd, D. Caspi, and M. Fernandez. Verifying linearizability of Intel software guard extensions. In *27th International Conference on Computer Aided Verification*, pages 144–160, July 2015. ISBN 978-3-319-21668-3. doi: 10.1007/978-3-319-21668-3_9.

[53] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000. doi: 10.1145/356989.357005.

[54] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Draft revision 120, Oct. 2012. URL http://www.cl.cam.ac.uk/~pes20/weakmemory/.

[55] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *13th ACM Symposium on Operating Systems Principles*, pages 110–121, 1991. ISBN 0-89791-447-3. doi: 10.1145/121132.344329.

[56] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[57] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *EuroSys Conference*, pages 315–328, 2008. ISBN 978-1-60558-013-5. doi: 10.1145/1352592.1352625.

[58] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, May 2010. doi: 10.1109/SP.2010.17.

[59] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. ISBN 978-1-4503-2118-1. doi: 10.1145/2487726.2488368.

[60] F. X. McKeen, C. V. Rozas, U. R. Savagaonkar, S. P. Johnson, V. Scarlata, M. A. Goldsmith, E. Brickell, et al. Method and apparatus to provide secure application execution, Dec. 2009. US Patent 9,087,200.

[61] MITRE. CVE-2017-5691, July 2017. URL https://nvd.nist.gov/vuln/detail/CVE-2017-5691.

[62] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, May 2013. doi: 10.1109/SP.2013.35.

[63] K. T. Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 family, Feb. 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[64] NXP. i.MX 7Solo, i.MX 7Dual applications processors, 2017. URL http://www.nxp.com/iMX7.

[65] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium*, pages 619–636, 2016. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko.

[66] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *EuroSys Conference*, Apr. 2017. doi: 10.1145/3064176.3064219.

[67] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy

on untrusted platforms. In *20th ACM Conference on Computer and Communications Security*, pages 13–24, 2013. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516678.

[68] R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure content-based routing using Intel software guard extensions. In *17th International Middleware Conference*, pages 10:1–10:10, 2016. ISBN 978-1-4503-4300-8. doi: 10.1145/2988336.2988346.

[69] H. Raj, D. Robinson, T. B. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted computing for guest VMs with a commodity hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, Dec. 2011.

[70] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A software-only implementation of a TPM chip. In *25th USENIX Security Symposium*, pages 841–856, 2016. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj.

[71] A. Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design*, pages 161–168, Oct. 2016. doi: 10.1109/FMCAD.2016.7886675.

[72] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. End-to-end verification of ARM processors with ISA-formal. In *28th International Conference on Computer Aided Verification*, pages 42–58, July 2016. ISBN 978-3-319-41540-6. doi: 10.1007/978-3-319-41540-6_3.

[73] A. Sabelfeld and A. C. Myers. *A Model for Delimited Information Release*, pages 174–191. Springer, Oct. 2004. ISBN 978-3-540-37621-7. doi: 10.1007/978-3-540-37621-7_9.

[74] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80, 2014. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541949.

[75] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, May 2015. doi: 10.1109/SP.2015.10.

[76] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 3–24. Springer, July 2017. ISBN 978-3-319-60876-1. doi: 10.1007/978-3-319-60876-1_1.

[77] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[78] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *11th ACM Asia Conference on Computer and Communications Security*, pages 317–328, 2016. ISBN 978-1-4503-4233-9. doi: 10.1145/2897845.2897885.

[79] T. Simonite. Intel puts the brakes on Moore's Law. *MIT Technology Review*, Mar. 2016. URL https://www.technologyreview.com/s/601102.

[80] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *22nd ACM Conference on Computer and Communications Security*, pages 1169–1184, 2015. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813608.

[81] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 665–681, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908113.

[82] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 437–450, 2012. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151022.

[83] *TPM Main Specification Level 2.* Trusted Computing Group, Mar. 2011. Version 1.2, Revision 116.

[84] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium*, pages 87–104, 2016. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/vasudevan.

[85] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. Technical Report UCB/EECS-2015-49, UC Berkeley EECS, May 2015.

[86] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT via SINIT code execution hijacking. http://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, Nov. 2011.

[87] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel Trusted Execution Technology. http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf, Dec. 2009.

[88] Y. Xu, W. Cui, and M. 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.

[89] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, 2010. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806610.