# Shielding applications from an untrusted cloud with Haven

Andrew Baumann Marcus Peinado Galen Hunt
*Microsoft Research*

## Abstract

Today's cloud computing infrastructure requires substantial trust. Cloud users rely on both the provider's staff and its globally-distributed software/hardware platform not to expose any of their private data.

We introduce the notion of shielded execution, which protects the confidentiality and integrity of a program and its data from the platform on which it runs (i.e., the cloud operator's OS, VM and firmware). Our prototype, Haven, is the first system to achieve shielded execution of unmodified legacy applications, including SQL Server and Apache, on a commodity OS (Windows) and commodity hardware. Haven leverages the hardware protection of Intel SGX to defend against privileged code and physical attacks such as memory probes, but also addresses the dual challenges of executing unmodified legacy binaries and protecting them from a malicious host. This work motivated recent changes in the SGX specification.

## 1 Introduction

Although users of cloud computing infrastructure may expect their data to remain confidential, today's clouds are built using a classical hierarchical security model that aims only to protect the privileged code (of the cloud provider) from untrusted code (the user's virtual machine), and does nothing to protect user data from access by privileged code. As a result, besides the hardware used to execute their applications, the cloud user must trust: (i) the provider's software, including privileged software such as a hypervisor and firmware but also the provider's full stack of management software; and (ii) the provider's staff, including system administrators but also those with physical access to hardware such as cleaners and security guards. Furthermore, as the Snowden leaks demonstrate [18, 19], the cloud user also implicitly trusts (iii) law enforcement bodies in any jurisdiction where their data may be replicated. By any measure, this is a large and inscrutable trusted computing base, and the related concerns are a significant factor limiting cloud adoption [13, 43].

The current best practice for protecting secrets in the cloud uses hardware security modules (HSMs) [e.g., 1]. These dedicated appliances rely on tamper-proof hardware to protect critical secrets, such as keys, and support a range of cryptographic functions, but come at a significant cost, and do not usually run general-purpose applications. Typical deployments use HSMs to protect key material, but transiently decrypt data on untrusted nodes for computation, rendering the data vulnerable to the threats outlined above. Previous research relied on trusted hypervisors to protect an application from a malicious OS [11, 25, 54, 60, 63], but cannot protect against a hypervisor controlled by a malicious or compromised cloud provider. Finally, although some applications can operate on encrypted data [4, 46, 55], cryptographic schemes for general-purpose computing [20, 21] have severe performance limitations.

Our objective is to run existing server applications in the cloud with a level of trust and security roughly equivalent to a user operating their own hardware in a locked cage at a colocation facility. Like the colocation provider (who is responsible only for power, cooling and network connectivity), the cloud provider is limited to offering raw resources: processor cycles, storage, and networking; it can deny service, but cannot observe or modify any user data except what is transmitted over the network. We refer to this property as *shielded execution*, and define it in §2. Essentially the inverse of sandboxing, it protects the confidentiality and integrity of code and data from an untrusted host. The high-level guarantee to the user is that secrecy is always preserved, and if their program executes, it behaves as if it ran on reference hardware under the user's control. The provider retains control of resource allocation, and may protect itself from a malicious guest.

Our prototype, *Haven*, implements shielded execution of unmodified Windows applications. It leverages Intel software guard extensions (SGX) [28, 29, 41], a set of new instructions and memory access changes summarised in §3.1. SGX allows a process to instantiate a secure region of address space known as an *enclave*; it then protects execution of code within the enclave, even from malicious privileged code or hardware attacks such as mem-

ory probes. While SGX was designed to enable new trustworthy applications to protect specific secrets by placing portions of their code and data inside enclaves [24], Haven aims to shield *entire unmodified legacy applications* written without any knowledge of SGX. This leads to two key challenges. First, executing legacy binary code inside an enclave pushes the limits of the SGX execution model: our target applications are large, raise and handle exceptions, dynamically allocate memory, and may execute arbitrary x86 instructions. Second, the code we seek to protect was written assuming that the OS it ran on would operate correctly, but the host OS may be malicious. To defeat such "Iago attacks" [10], where a malicious OS subverts a protected application by exploiting the application's reliance on correct results of system calls, we use an in-enclave library OS (LibOS). The LibOS used by Haven is derived from Drawbridge [47]; it implements the Windows 8 API using a small set of primitives such as threads, virtual memory, and file I/O. As we describe in §4, Haven implements these primitives using a mutually-distrusting interface with the host OS. This ensures shielded execution of unmodified applications; a malicious host cannot trick an application into divulging its secrets nor executing incorrectly. Combined with a remote attestation mechanism [2], Haven gives the user an end-to-end guarantee of application security without trusting the cloud provider, its software, or any hardware beyond the processor itself.

We developed Haven on an instruction-accurate SGX emulator provided by Intel, but evaluate it using our own model of SGX performance. Haven goes beyond the original design intent of SGX, so while the hardware was mostly sufficient, it did have three fundamental limitations for which we proposed fixes (§5.4). These are incorporated in a revised version of the SGX specification [29], published concurrently by Intel.

The contributions of this paper are:

- We define the concept of shielded execution (§2), describe how SGX supports it (§3.1), and later outline generalised hardware requirements (§7.4).
- We present Haven, the first system to implement shielded execution of unmodified binaries for a commodity OS, achieving mutual distrust with the entire host software stack (§4–5). Haven shields applications using mechanisms such as private scheduling, distrustful virtual memory management, and an encrypted and integrity-protected file system.
- We evaluate Haven's performance using unmodified server applications: SQL Server and Apache (§6).
- We identify minimal changes to SGX to enable efficient shielded execution of unmodified applications (§5.4), and note optimisation opportunities (§7.3).

# 2 Security Overview

## 2.1 Shielded execution

Like others [41, 44, 59], we use the term *isolated execution* to refer generally to mechanisms that protect the confidentiality and integrity of specific code and data from other actors. In contrast to previous protection mechanisms such as process isolation, sandboxing, managed code, etc. which serve to confine an untrusted program and protect the rest of a system from its actions, isolated execution refers to the inverse: *protecting specific code from the rest of the system, however large or privileged.*

Various forms of isolated execution are possible. Software implementations rely on a trusted component such as a hypervisor that implements isolation (e.g., using page protection and/or encryption) [11, 14, 25, 39, 54, 60, 63]. Conversely, in pure-hardware implementations, no software other than the isolated code is in the trusted computing base. While several hardware isolation mechanisms exist [9, 31, 34, 44, 59], SGX is the first commodity hardware that permits efficient multiplexing among multiple isolated programs without relying on trusted software.

However, isolation alone is not sufficient to protect applications. In order to be useful, an isolation mechanism must permit interaction with untrusted software or hardware, to communicate results or access system services, and it is at these points that a naïve isolated program is vulnerable [10]. For example, SGX isolates self-contained sequences of x86 instructions (typically, individual modules or functions) that are aware of the SGX protection model and are explicitly written to defend against threats outside the enclave, that do not handle faults or exceptions, and do not interact with the OS.

*Shielded execution* builds on an isolation mechanism to provide higher-level security properties; specifically, for an abstract program, it guarantees:

- *Confidentiality:* The execution of the shielded program appears as a "black box" to the rest of the system. Only its inputs and outputs, but no intermediate states, are observable.
- *Integrity:* The system cannot affect the behaviour of the program, except by choosing not to execute it at all or withholding resources (denial of service attacks). If the program completes, its output is the same as a correct execution on a reference platform.

While the term may be new, the underlying concept is not. In using a new term, we attempt to generalise beyond specific implementations such as cloaking [11, 12], "pieces of application logic" [33, 38, 39], protected modules [45] and high-assurance processes [25] that provide

shielded execution under a specific set of constraints. Moreover, in Haven, our goal is to relax those constraints to achieve shielded execution for unmodified application binaries with complex OS dependencies.

Note that shielded execution is necessary but not sufficient to meet our goal of confidential execution in the cloud. We also require an attestation mechanism to establish confidence in the integrity of a remote shielded program, and a mechanism to provision secrets directly to it, allowing it to operate on encrypted inputs and extend confidentiality beyond the confines of the shield mechanism. We describe how SGX supports this in §3.1.

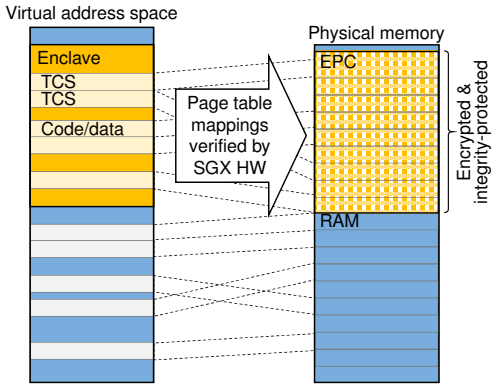



Figure 1: SGX virtual and physical memory layout

## 2.2 Threat model and assumptions

We seek to protect the confidentiality and integrity of a user's unmodified server application from an untrusted cloud provider. We specifically exclude software-based isolation mechanisms, because besides their vulnerability to simple hardware attacks, we wish to give the cloud provider the unfettered ability to patch and update its privileged software (we expand on this later in §8). We therefore assume a powerful adversary that controls most of the provider's hardware and all its software.

At the hardware level, we assume that the processor itself is implemented correctly, and not compromised (so the adversary cannot extract secrets residing within it). The adversary has full control beyond the physical package of the processor, including memory and all I/O devices. They may probe memory, and arbitrarily alter or inject I/O including network traffic.

The adversary also controls the cloud provider's entire software stack, including the host OS, hypervisor, management software, platform firmware, BIOS, code executing in system management mode, and device firmware. As a result, they may interrupt execution of the user's program indefinitely, and may pass arbitrary values across the isolation boundary (e.g., the SGX enclave), including the results of calls to OS services and arbitrarily-injected upcalls. We assume a secure source of random numbers (which recent processors provide). However, the adversary may interfere with other sources of non-determinism such as thread interleaving, subject to the constraints of the hardware specification (e.g., the memory model).

We do not consider any side-channel attacks. Common side-channels, such as timing and cache-collision, have known (but expensive) attack mitigations [e.g., 8] that can be implemented by application software; others, such as power analysis, require hardware modifications, and are ultimately a limitation of our approach.

# 3 Background

## 3.1 Intel SGX

In this section, we summarise the SGX functionality relevant to Haven; readers are directed to the specification [28, 29, 41] for full details. Although SGX protects against any malicious privileged code (OS, hypervisor, firmware, system management mode, etc.), we refer to it collectively as simply the "OS".

**Memory protection** SGX protects the confidentiality and integrity of pages in an *enclave*, a region of a user-mode address space (Figure 1). While cache-resident, enclave data is protected by CPU access controls (the TLB). However, it is encrypted and integrity protected when written to memory, and if the data in memory is modified, a subsequent load will signal a fault.

SGX mediates page mappings at enclave setup and maintains shadow state for each page. Enclaves are created by an `ECREATE` instruction, which initialises a control structure in protected memory. Once an enclave has been created, pages of memory are added to it using `EADD`. These pages are allocated by the OS, but must occupy a specific region of physical memory: the enclave page cache (EPC). For each EPC page, hardware tracks its type, the enclave to which it is mapped, the virtual address within the enclave, and permissions (read, write, execute). On each enclave page access, after walking the page table, SGX ensures that the processor is in enclave mode, the page belongs to the EPC and is correctly typed, the current enclave maps the page at the accessed virtual address, and the access agrees with the page permissions.

Like the RAM backing it, EPC is a limited resource. Therefore, SGX enables the OS to virtualise EPC by paging its contents to other storage [28, §3.5]. Privileged instructions cause the hardware to free an EPC page cho-

sen by the OS, writing its contents to an encrypted buffer in main memory, which the OS may then relocate. To prevent rollback attacks on page-in, the hardware keeps a version number for the page in EPC. It also requires the OS to follow a hardware-verified protocol to ensure that TLB shootdown has completed when evicting a page.

**Attestation** SGX supports CPU-based attestation [2], enabling a remote system to verify cryptographically that specific software has been loaded within an enclave, and establish shared secrets allowing it to bootstrap an end-to-end encrypted channel with the enclave.

During enclave creation, a secure hash known as a *measurement* is established of the enclave's initial state. The enclave may later retrieve a *report* signed by the processor that proves its identity to, and communicates a unique value (such as a public key) with, another local enclave. Using a trusted *quoting enclave*, this mechanism can be leveraged to obtain an attestation known as a *quote* which proves to a remote system that the report comes from an enclave running on a genuine SGX implementation [2]. Ultimately, the processor manufacturer (e.g., Intel) is the root of trust for attestation.

**Enclave entry and exit** Besides protecting the content and integrity of memory mappings, SGX also mediates transitions into and out of the enclave, and protects the enclave's register file from OS exception handlers. This is managed using a thread control structure (TCS).

User code begins executing an enclave by invoking `EENTER` on an idle TCS; this acts as a call gate, transferring control to a defined entry point within the enclave. Enclave code may access enclave pages according to the protection model outlined above; it may also read and write memory outside the enclave region (as permitted by OS page tables), but any attempt to execute code there faults. The processor continues in enclave mode until software explicitly leaves it by invoking `EEXIT`, or until an interrupt or exception returns control to the OS, which is known as an *asynchronous exit*. After an explicit exit, control resumes outside the enclave at an address chosen by the enclave; in this way `EENTER` and `EEXIT` can be used with stubs that wrap invocations of enclave functions, taking care to validate inputs on entry and scrub secrets on exit from any registers not used as return values.

After an asynchronous exit, control transfers to the OS exception handler; typically this would save the registers for later use (e.g., when next scheduled), but the OS cannot be trusted with the enclave's register state. Instead, SGX saves the full context and information about the cause of the exit in the TCS, replacing it with a synthetic context before reporting the exception to the OS. The enclave may later be resumed by `ERESUME` on the TCS, which restores its last saved context. Alternatively, the OS can re-enter the enclave, giving it the opportunity to inspect and modify its own state before resuming; this is used to report an exception which must be handled by the enclave.

SGX is an imperfect implementation of shielded execution according to our definition in §2.1, because the OS exception handler observes some of the enclave's internal state: the exception vector, and in the case of a page fault, the type of access and base address of the page [28, §4.4]. This allows the OS to retain control over resource management (i.e., CPU time and memory); in general, it can deny service to the enclave, but cannot cause it to execute incorrectly. We discuss hardware designs to decouple resource management from observations of guest behaviour later in §7.4.

**Dynamic memory allocation** As described, SGX does not allow enclave pages to be added after creation, nor EPC permissions changed, which is clearly insufficient for Haven to run unmodified applications. However, revision 2 of the SGX specification [29] includes new instructions allowing the enclave and host OS to *cooperatively* add/remove enclave pages and modify their permissions.

Allocation requires cooperation, because the host manages EPC but cannot be trusted to arbitrarily add enclave pages (e.g., in an unallocated region). To allocate a new page, the host invokes `EAUG` to place an unused EPC page at a specific offset in an enclave; this must then be acknowledged by the enclave executing `EACCEPT`, before it becomes accessible. Similarly, reducing permissions or removing pages also requires cooperation, because like page eviction, hardware must help ensure TLB shootdown has occurred. SGX includes instructions (`EMODT`, `EMODPR`, `EBLOCK`, `ETRACK` and `EACCEPT`) to enable this.

These operations do not change the enclave measurement established by `EINIT`; since the modified pages come under the full control of the enclave, its identity is equivalent for trust purposes.

## 3.2 Drawbridge

Haven builds on Drawbridge [6, 47], a system supporting low-overhead sandboxing of Windows applications. Drawbridge consists of two core mechanisms, both of which Haven leverages: the picoprocess, and library OS.

The *picoprocess* is a secure isolation container constructed from a hardware address space, but with no access to traditional OS services or system calls [15]; instead, a narrow ABI of OS primitives is provided, implemented using a security monitor. The ABI consists of 40 downcalls and three upcalls [6]. Downcalls are requests for OS services including virtual memory, thread-

ing, and I/O streams (e.g., files and network sockets). Upcalls are initiated by the host, have only input parameters, and do not return; they are used for initialisation, thread startup, and exception delivery. In Haven, as in Drawbridge, the picoprocess serves to protect the host (i.e., the cloud provider) from a potentially-malicious guest.

The Drawbridge *LibOS* is a version of Windows 8 refactored to run as a set of libraries within the picoprocess, depending only on the ABI. It consists of lightly-modified binaries for most user-mode and some kernel components of Windows, and a "user-mode kernel" that implements the interfaces on which they depend.

Together, the picoprocess and LibOS enable sandboxing of unmodified Windows applications with comparable security to virtual machines, but substantially lower overheads. While Drawbridge aims only to protect the host from an untrusted guest, Haven shields the execution of the application and LibOS from an untrusted host, thereby enabling mutual distrust between host and guest.

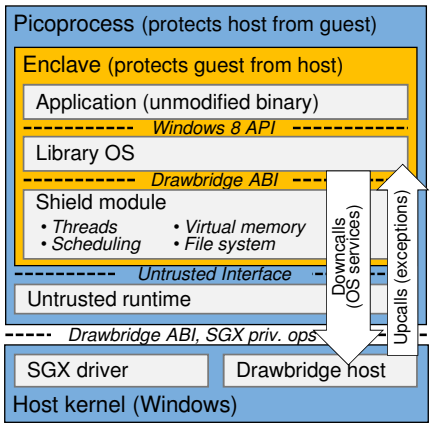



Figure 2: Haven components and interfaces

# 4 Design

We now present the design of Haven, which leverages the instruction-level isolation mechanism of SGX to achieve shielded execution of entire legacy application binaries. In doing this, we address two key challenges: protecting from a malicious host OS, and executing existing binaries in an enclave. We first discuss these in more detail.

## 4.1 Design challenges

**Malicious host OS** A general class of threats known as Iago attacks arises when a malicious OS attempts to subvert an isolated application by exploiting its assumption of correct OS behaviour, for example when using the results of system calls [10]. Besides simply returning semantically-incorrect results from system calls (e.g., returning the address of an already-allocated region for a new memory allocation), the malicious OS may seek to exploit latent bugs in the application. For example, it may allocate valid but abnormally-high virtual addresses, return unusual values for parameters such as memory size and number of processors, alter timing to seek to exploit latent race conditions, inject spurious exceptions, return unexpected error codes from system calls, or simply fail calls that an application naively assumes will succeed.

Our approach to this challenge is twofold. First, we limit its scope using a LibOS within the enclave. The LibOS implements the full OS API using a much narrower set of core OS primitives. Since the LibOS is under user control, and can be arbitrarily tested or inspected offline, we assume that it is not malicious (to the user), even though it may be large, complex, and contain bugs. Second, having reduced the scope of attacks by narrowing the interface they must traverse, we use established techniques to correctly implement the OS primitives in the presence of a malicious host: careful defensive coding, exhaustive validation of untrusted inputs, and encryption and integrity protection of any private data exposed to untrusted code.

**Unmodified binaries** SGX was designed to protect limited subsets of application logic [24], however full application binaries have properties that make them challenging to execute in an enclave. They load code and data at runtime, dynamically allocate and change protection on virtual memory, execute arbitrary user-mode instructions (including some not supported by SGX), raise and handle exceptions (e.g., page faults, divide-by-zero or floating-point exceptions), and use thread-local storage.

Haven addresses each of these challenges. For some, such as thread-local storage, we rely on enhancements to SGX described later in §5.4. For most, we work around the limitations, by emulating unsupported instructions, carefully validating and handling exceptions that occur within an enclave, and modifying LibOS behaviour.

## 4.2 Architecture

Figure 2 shows the architecture of Haven. We create an enclave within the Drawbridge picoprocess containing the entire application and LibOS. To protect the LibOS and application from a malicious host, Haven augments Drawbridge with two layers: a *shield* module below the LibOS in the enclave, and an *untrusted runtime* outside the en-

Upcalls:

```
ExceptionDispatch(ExceptionInfo)
ThreadEntry()
```

Downcalls:

```
AsyncCancel(AsyncHandle)
AsyncPoll(AsyncHandle) -> Results
DebugStringPrint(Message)
EventClear(EventHandle)
EventSet(EventHandle)
ObjectClose(Handle)
ObjectsWaitAny(Count, Handles, Timeout) -> Index
ProcessExit(ExitCode)
StreamAttributesQueryByHandle(StreamHandle) -> Attribs
StreamFlush(StreamHandle)
StreamGetEvent(StreamHandle, EventId) -> EventHandle
StreamOpen(URI, Options) -> StreamHandle
StreamRead(StreamHandle, Off, Len, Buf) -> AsyncHandle
StreamWrite(StreamHandle, Off, Len, Buf) -> AsyncHandle
SystemTimeQuery() -> Time
ThreadCreate(Tcs) -> ThreadHandle
ThreadExit()
ThreadInterrupt(ThreadHandle)
ThreadYieldExecution()
VirtualMemoryCommit(Addr, Size, Prot)
VirtualMemoryFree(Addr, Size)
VirtualMemoryProtect(Addr, Size, Prot)
```

Figure 3: Untrusted interface to enclave.

clave. These effectively interpose on the LibOS/host interface (the Drawbridge ABI) [6], implementing a shielded version in the enclave by calling out to the untrusted host.

Our design is independent of the specific LibOS; recent Linux LibOSes [6, 27, 58] might also be used.[1]

**Shield module** As with all code inside the enclave, this is in the application's trusted computing base. Its high-level role is to implement the Drawbridge ABI required by the LibOS in terms of a more limited subset of core OS operations. It therefore includes private implementations of typical kernel functionality such as memory management, a file system, and thread synchronisation. It also acts as a trusted bootloader for the LibOS and application.

The shield is responsible for protecting the LibOS and application from Iago attacks outside the enclave. It does this by careful validation of all parameters and results passed across a narrow interface with the untrusted runtime. At its most basic, this validation consists of ensuring that the parameters of upcalls and results of downcalls are consistent with their specification. For example, the number of bytes read from a stream cannot be more than the requested size, and it is not acceptable to return an error code indicating a timeout for an operation that cannot do so (more generally, each downcall has a specific list of acceptable failure codes). Specific calls require further validation, using either hardware support (e.g., when changing virtual memory permissions), or additional software (e.g., for thread synchronisation), and are discussed in the relevant sections below.

Since our threat model permits denial of service, the shield can and does handle any incorrect host behaviour by panicking: it emits a short debug message, requests the host to terminate its process, and rejects subsequent attempts to enter the enclave.

**Untrusted interface** The interface at the enclave boundary must allow the shield to verify the correctness of all operations while also enabling an efficient implementation. Besides minimality, our guiding principle in designing it was a form of policy/mechanism separation [32]: *the guest controls policy for virtual resources (virtual address allocation, threads, etc.), while the host manages policy only for physical resources (e.g., memory and CPU time)*. In general, this prevents operations that give any implementation freedom to the host beyond physical resource allocation, makes verification efficient, and limits the scope of attacks.

The interface is summarised in Figure 3; it is expressed as a Drawbridge ABI subset, with fewer (22 rather than 40) calls and fewer permissible arguments; specifically:

- calls to commit, free and protect specific pages;
- thread management and signalling;
- I/O streams to access untrusted storage and network;
- a source of system time.

**Untrusted runtime** Primarily bootstrap and glue code, this is trusted by neither enclave nor host kernel. Its main tasks are creating the enclave, loading the shield, and forwarding calls between the enclave and host OS.

## 4.3 Shield services

**Virtual memory** The virtual address region occupied by a Haven enclave always starts at zero (enforced by a check at startup), allowing the enclave to reliably detect and handle NULL pointer dereferences. Otherwise, a malicious host OS could map pages there, and redirect NULL accesses to data of their choosing. The enclave's virtual size must be large enough for all possible allocations by

[1] We note however that `fork()` would be both complex and expensive, as it requires a new enclave communicating via untrusted channels.

the application/LibOS, and small enough to leave some address space for the untrusted runtime and host OS. In our prototype, enclaves occupy 64GB of address space.

The shield manages virtual memory within the enclave. It includes a region allocator, tracking sub-regions of the enclave that are reserved for use. For each reserved region, it tracks which pages are committed (i.e., accessible to the application), and for those pages, their permissions (read, write, execute). For each allocation, the shield chooses an address based on its knowledge of allocated regions. When memory is committed or its protection is changed, the shield calls out to the host to make the appropriate changes (e.g., allocating and mapping EPC pages and performing TLB shootdown if necessary), then uses the dynamic memory allocation instructions described in §3.1 to ensure that the expected changes were made. To prevent exploits of latent bugs in the application or LibOS, the shield never allows the host to choose virtual addresses. It also blocks the application from using non-enclave memory, by failing requests to allocate it.

**Storage** While SGX provides confidentiality and integrity protection for data in memory, Haven must also support secure persistent storage. Rather than simply encrypting file contents, which risks leaking guest state through file metadata, the shield implements a private filesystem. Our prototype uses a FAT32 filesystem inside an encrypted virtual hard disk (VHD) image.

The shield encrypts each disk block independently with an authenticated encryption algorithm (AES-GCM [40]), keying the encryption to the block number. Like other systems [16, 17, 26, 36, 62], a Merkle tree [42] protects the integrity of the overall disk. This can be implemented with little overhead, as only the root and the leaf nodes of the tree are persisted to disk [26]. Like InkTag [25], we store the crypto metadata (message authentication codes of data blocks, nonces, and the Merkle tree root) in separate blocks from filesystem data. We also adapted InkTag's two-hash-versions scheme to maintain consistency after crashes. We discuss rollback attacks in §7.2.

**Threads and synchronisation** To prevent the host from exploiting the application, for example by allowing two threads to concurrently acquire a mutex, the shield implements a form of user-level scheduling [3, 37]. At startup, it creates a fixed number of threads according to the desired level of parallelism (typically, the number of hardware threads). These operate as virtual CPUs supporting an arbitrary number of application threads inside the enclave. Besides multiplexing application threads across the virtual CPUs, the shield's scheduler implements primitives for events, mutexes and semaphores. It maintains its internal state (run queues and synchronisation objects) using atomic instructions for safety, and uses the untrusted interface's event and interrupt mechanisms to support suspending/resuming and signalling the virtual CPUs.

The untrusted host can deny service by delaying wake-ups or interrupts, but cannot cause the application to execute incorrectly. Moreover, its ability to exploit latent race conditions in the application by delaying guest execution is severely curtailed by the multiplexing of application threads (which it cannot observe) onto virtual CPUs.

Table 1: Summary of component sizes

| | LoC[a] | Size |
|---|---|---|
| Drawbridge LibOS | millions[b] | 209 MB[c] |
| Shield module | 23,095 | 180 kB |
| Untrusted runtime | 7,446 | 52 kB |
| SGX driver | 4,520 | 41 kB |

[a] Lines of code counted by David A. Wheeler's "SLOCCount".
[b] See Porter et al. [47] for a breakdown (of a previous version).
[c] We report file size for all binaries in the LibOS; the subset that is loaded depends on the application, but is usually much smaller.

**Miscellaneous** The shield handles calls for entropy generation (using `RDRAND`, a secure source of randomness) and dynamic loading/relocation of application binaries. Our loader does not yet implement address-space layout randomisation [7]; this is planned for future work.

Process creation is not supported. While not inconceivable, it would be extremely complex and expensive to implement on SGX, requiring the creation of a new enclave (in a separate address space), and communication with it over untrusted channels. One advantage of the Windows OS is that surprisingly few applications use child processes [6, 47]. For those that do, it is often sufficient to run the "subprocess" in a different portion of the parent's address space (i.e., in the same enclave), since the API has no `fork()` operation; this is supported by the LibOS.

# 5 Implementation

Table 1 reports the size of various components in our current prototype. Besides implementing the shield and untrusted runtime, we added SGX support to our host OS (Windows 8.1) by writing a driver and making some kernel changes. The driver implements SGX kernel-mode operations: allocating and mapping EPC pages, and creating and destroying enclaves. It is trusted by the host, but untrusted by enclave code. We also modified the host kernel to enable efficient mapping of EPC pages to user-mode. This was necessary, because EPC regions appear as reserved device space to the kernel, and the existing

driver APIs for mapping device memory to user-mode did not anticipate the need for efficient page-granular mapping and protection changes. We also implemented support for debugging an enclave using the SGX debug mechanisms [28, Chapter 7]. Finally, we made minor modifications (249 lines of code) to the LibOS to avoid using shared memory within the picoprocess,[2] which SGX does not support.

## 5.1 Application deployment and attestation

Haven applications are deployed similarly to cloud VMs, with an extra attestation step we now describe. A user constructs a disk image containing application and LibOS binaries and data, and then encrypts it symmetrically, retaining the key. The encrypted VHD and shield binary are sent to the cloud provider. The shield is not encrypted, but its integrity will be verified.[3] The cloud provider establishes a picoprocess, and loads the untrusted runtime, which then creates an enclave and loads the shield module. While the shield is loaded, the SGX hardware attestation mechanism (§3.1) is used to measure (i.e., compute a secure hash of) its code and initial state. The shield receives two startup parameters: a structure of untrusted parameters chosen by the host, such as addresses of downcall functions in the untrusted runtime, and trusted parameters chosen by the user, such as configuration options and environment variables, which form part of the enclave's measurement.

After initialising itself, the shield generates a public/private key pair, and then uses its parameters to establish a network connection with the user – this may be a machine physically under the control of the user, or another enclave in the cloud. In either case, the shield uses the SGX attestation mechanism to produce a quote containing its public key which it sends to the user, proving that it has been correctly loaded and executes in an SGX enclave. If the enclave's measurement is as expected (i.e., if the shield was loaded correctly with the desired parameters), the user encrypts the VHD key using the public key contained in the quote, and sends it back to the shield; any tampering with the shield binary is detected and subverted at this point. Assuming it was loaded correctly, the shield may now decrypt the VHD key using its private key, and use it to access the contents of the VHD, allowing it to continue to load the LibOS and application.

From this point onward, communication with the outside world, and therefore access to any secrets contained in the VHD, is under application control. Typical server applications supporting SSL-encrypted connections may be configured using certificates and keys stored directly in the VHD, and accessed over the cloud provider's untrusted network. For future work, we are planning to add support for encrypted virtual private networks between a user's enclaves (or trusted hosts), providing a secure network to applications that require one.

## 5.2 Enclave entry/exit

In Haven, an application performs most of its execution in the enclave, calling out to the untrusted host only for system services. This is the opposite of the typical SGX usage model of untrusted code calling into an enclave [24]. To perform an upcall, the untrusted runtime loads the upcall parameters into specific registers, and invokes `EENTER`, which does not return to its caller but instead delivers control to the shield entry point inside the enclave.

To perform a downcall, the shield passes arguments in registers while clearing any unused registers (to prevent leaking secrets), stores the return address and stack pointer inside the shield's thread record, and invokes `EEXIT` with a target address of the relevant downcall handler. SGX leaves the enclave and executes the untrusted handler, which first loads a stack pointer before calling C code. When the downcall returns (generally, after a system call) its results are delivered to the enclave by `EENTER`, which re-enters the enclave at the shield entry point.

The shield must disambiguate the different entry causes. To do so it inspects the SGX thread structure, which identifies whether an exception occurred, and its own thread record, which records whether a downcall was in progress. It then reloads the stack pointer, and either calls (on an upcall) or returns to (on a downcall) C code.

Parameters that are passed by reference (e.g., I/O buffers) cannot be located inside the enclave, since they are inaccessible to the host. Instead, the shield allocates a "bounce buffer" from a memory region outside the enclave, and copies the parameters appropriately. We are considering (but have not yet implemented) an optimisation to the file system to encrypt directly into the bounce buffer on writes, and decrypt from it on reads; this would reduce the copy overhead for most file I/O.

## 5.3 Exception handling

When a page fault occurs within an enclave, SGX saves the register context and fault information to an in-enclave data structure (see §3.1). It then delivers an exception

[2] Code that relied on making multiple mappings of a shared memory section within the process was changed to use a single virtual address.

[3] If confidentiality of the shield was desirable, a smaller trusted bootloader could be used whose only task would be to perform attestation at startup and then decrypt and load the shield binary.

to the host OS, which may choose to handle the fault (e.g., by lazily updating a page table) and resume execution, or to report it to the user process. In the latter case, the untrusted runtime upcalls the shield in the enclave, which must then determine the true cause using the information and register context provided by SGX. The shield exception handler performs sanity-checks to ensure that the exception is valid and should be reported to the LibOS. These include checking that: an exception actually occurred (as reported by SGX); the instruction is in the enclave but not the shield module (which should never fault); and the fault type (read, write, or execute) is consistent with the page's expected permissions.

The shield prepares to deliver the exception to the LibOS, by copying the context and cause in a format defined by the Drawbridge ABI, and modifying the context to run the LibOS exception handler. Haven must now resume the modified context. Unfortunately, SGX only allows `ERESUME` outside an enclave, so the shield must `EEXIT` to a small (untrusted) stub that immediately resumes the enclave, restoring the context. We discuss the performance implications of this additional exit later, in §7.3.

Most other exceptions are handled similarly to page faults. The one special case is illegal instructions: as we describe later in §5.4, some user-mode instructions are illegal in an enclave. The trusted exception handler decodes and emulates these, by modifying the processor context and advancing the instruction pointer.

## 5.4 SGX limitations and workarounds

In addition to the need for dynamic memory allocation, we encountered three architectural limitations with SGX as initially specified [28] that made it impossible to run existing application and LibOS binaries. We summarise these issues, our workarounds, and proposed changes. Working with Intel, these changes are now incorporated in the revised SGX specification [29].

**Exception handling** SGX allows an enclave to handle its own exceptions by reporting the exception cause and register context securely in the TCS. However, while the registers are always saved, not all exception causes are reported to the enclave. For example, hardware interrupts are of no relevance to the enclave and may reveal private information about the host configuration, so they are not reported. As originally specified [28, §2.6.3], the list of reported exceptions included program faults such as division by zero, breakpoint instructions, undefined instructions, and floating point / SIMD exceptions, but not page faults or general protection faults. This prevented an enclave from handling these faults without trusting the host for information such as faulting address and access type. However, page faults are commonly handled by user-mode code, for example in demand loading or stack allocation. General protection faults are less common, but may also occur, e.g. in a LibOS emulating privileged instructions. SGX now reports these faults to the enclave [29].

**Permitted instructions** SGX disallows in-enclave execution of instructions that may cause a VM exit or change software privilege levels [28, §3.6]. Unfortunately, three of these instructions are commonly encountered in LibOS and application binaries: `CPUID`, `RDTSC`, and `IRET`.

**`CPUID`** This instruction queries processor features, generally to test for extended instructions. SGX prevents its use within an enclave, because a virtual machine may be configured to trap and emulate it, but emulation is impossible since the enclave's registers are not visible to the hypervisor. Instead, the processor signals an invalid instruction, and Haven's exception handler emulates `CPUID` using static knowledge of features available on SGX.

**`RDTSC` and `RDTSCP`** These instructions return the cycle counter, and are commonly used as a low-overhead time source, e.g. to measure hold-time in adaptive spinlocks. The initial version of SGX prevented their use because, like `CPUID`, they may cause a VM exit. However, unlike `CPUID`, they are not feasible to emulate: first, there is no reliable source of time, and second, most uses of `RDTSC` rely on its low overhead, which emulation cannot achieve. Instead, the SGX specification was revised to permit these instructions if VM exiting is disabled [29].

**`IRET`** Nominally an "interrupt return", this is used at the end of an exception handler when restoring processor state. It pops registers including instruction and stack pointers, and returns in the new context.[4] Since `IRET` can also change protection level, SGX disallows its execution in an enclave. Haven presently emulates `IRET`, but this adds overhead to exceptions, as we discuss later in §7.3.

**Thread-local storage** For legacy reasons, thread-local data on x86 is accessed via FS or GS segments. On SGX, `EENTER` and `ERESUME` load private FS and GS base addresses from the TCS. However, because a TCS is immutable once created, the addresses must be known at startup. Instructions exist to change FS/GS (`WRFSBASE`, `WRGSBASE`), but these could not reliably be used in an enclave, since the changes were not saved on asynchronous exits, and `ERESUME` restored FS and GS from the TCS.

As a result of this limitation, it was not possible to context-switch a TCS between application threads, and

[4] `IRET` is used since, to our knowledge, it is the only user-mode instruction that can restore a complete context, including volatile registers.

therefore impossible to perform user-mode scheduling within an enclave. As a workaround, the Haven prototype maps application threads 1:1 onto TCSs and host threads.[5] Although the shield's scheduler still ensures the correct behaviour of all synchronisation primitives, the host's ability to control scheduling of application threads makes it more likely that a malicious host could exploit application-level bugs by arbitrarily delaying threads.

This problem has been addressed in the revised SGX specification [29], but an implementation was not yet available to us, so our prototype relies on the workaround.

### 5.5 Unimplemented aspects

Our prototype implements the full design, with three exceptions. Rather than the attestation mechanism, we send the VHD key in the clear. We also built a simplified version of the disk integrity scheme that has equivalent performance, but cannot detect all block rollback attacks. Finally, we "emulate" `CPUID` by exiting the enclave to execute the instruction. We do not expect these shortcuts to materially impact performance.

## 6 Performance Evaluation

We developed and tested Haven using a functional emulator for SGX provided by Intel. However, in the absence of an SGX CPU or cycle-accurate emulator, we must do our own performance modelling. Our approach is to measure Haven's sensitivity to key SGX performance parameters.

In this section, we first describe our performance model, before reporting results for two typical cloud applications: Microsoft SQL Server, and Apache HTTP Server. Our performance experiments were run on a system comprised of a 4-core Intel Core i7-4700HQ CPU running at 2.4GHz with 8GB of 1600MHz DDR3 RAM, a 240GB SSD (Intel SC2CW240A3), and a gigabit Ethernet interface (Intel I217-LM) running Windows 8.1 Pro. We used this mobile-optimised platform, because it permits us to adjust the DRAM frequency and timings, allowing us to simulate a variable memory penalty for SGX.

### 6.1 Performance model

To model performance, we assume that an SGX implementation will perform the same as a current CPU, except for (i) additional costs (direct and indirect) of SGX instructions and asynchronous exits, and (ii) an additional memory penalty (latency and/or bandwidth of cache misses) for memory encryption when accessing EPC.

Many SGX instructions are executed only at enclave startup, and are therefore irrelevant to the performance of long-running server applications. We also assume that the EPC will be large enough to hold the working set of our applications, and therefore do not model the overhead of paging it to backing store. The only remaining direct overheads for Haven performance on SGX are the instructions for dynamic memory allocation (§3.1), and transitions into and out of enclave mode: `EENTER`, `EEXIT`, `ERESUME`, and asynchronous exits. The dynamic allocation instructions only check and update page protection metadata. The transitions are documented as requiring a TLB flush [28] and also perform a series of checks and updates.

For our evaluation, we implemented a second version of Haven that does not use SGX. Instead, it simulates SGX performance for the above critical instructions by busy-waiting for a configurable number of processor cycles, which we vary. In addition, for each enclave transition a system call is used to flush the TLB. Since the system call itself adds overhead not present on SGX, we view this as a conservative estimate for the performance of SGX. We cannot simulate the overhead of disallowed instructions such as `IRET` and `CPUID`, since there is no practical way to make them trap without SGX. However, we know from experience with the emulator that `CPUID` is only invoked at startup, and `IRET` is relatively rare (e.g., we observed around 100 `IRET`s per second for a web server workload).

Memory penalties for EPC access are more difficult to model. We simulate the impact of slower EPC by artificially reducing the system's DRAM frequency.

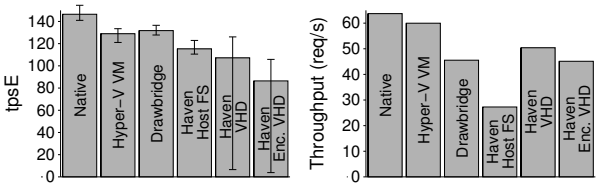
### 6.2 Application workloads

**Database** We run Microsoft SQL Server 2014, Enterprise Edition, and TPC-E [56], a standard online transaction processing benchmark. We use the default configuration for SQL Server when running natively or in a VM, but for Drawbridge and Haven we varied some parameters. Drawbridge does not support large pages or locked physical allocations, so we disabled them. We also limited the buffer cache to 6.5GB (the best-performing size), because the LibOS does not report physical memory usage, and the server's default behaviour led to excessive paging.

The TPC-E clients run on a single machine connected to our test system by a local gigabit network. We generated a database of 1000 customers[6] and left other pa-

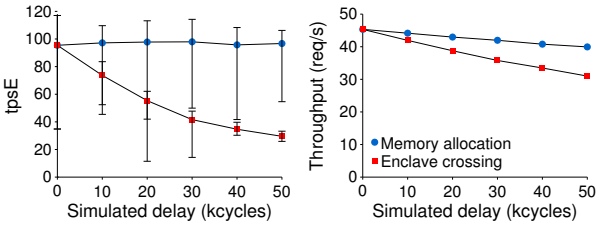
[5] This workaround is possible since the storage sizes for FS and GS are constant and Drawbridge does not choose their location [6, §3.1].

[6] Our database is smaller than the minimum for official TPC-E results (5000 customers), but sufficient to saturate our test system.




(a) SQL Server, TPC-E (b) MediaWiki on Apache

Figure 4: Performance breakdown




(a) SQL Server, TPC-E (b) MediaWiki on Apache

Figure 5: Sensitivity to SGX instruction overhead

rameters at the default settings. For each run we allowed at least 30 minutes of warm-up time, and then measured transaction performance for one hour, reporting the overall throughput. Error bars show minimum and maximum throughput over the run for a sliding 1-minute interval.

**Web server** We run Apache HTTP server[7] version 2.4.7 and PHP 5.5.11. We configured Drawbridge to run Apache's worker processes in the same address space (and enclave), and modified Apache's configuration to avoid using `AcceptEx`, which exposed a compatibility bug in the LibOS socket code. We installed MediaWiki 1.22.5 backed by a SQLite database, and enabled the Alternative PHP Cache for intermediate code and MediaWiki page data. We benchmarked the server using 50 worker threads on the client that repeatedly fetched the 14kB main page over persistent SSL connections for a period of 5 minutes.

## 6.3 Results

**Overall performance** We begin by comparing the performance of Haven to alternative host environments (none of which provide shielded execution). Figure 4 shows a performance breakdown for several configurations of each workload: native execution on Windows 8.1, in a Hyper-V VM, in Drawbridge, and three different configurations of Haven: one that trusts the host to implement the filesystem, the next using the private (VHD-backed) filesystem but not encrypting it, and finally the full system with VHD encryption and integrity protection enabled. In all Haven workloads, we flush the TLB on enclave crossings, but do not insert any additional delay for the SGX instructions. We verified that the server's CPU (and not network or storage I/O) is the bottleneck in all non-Haven runs, so these results give a reasonable indication of the overhead of the various software components.

For SQL Server, the extra runtime layers and TLB flush on enclave crossings give Haven a 13% slowdown vs. Drawbridge. Furthermore, Haven's unoptimised FAT filesystem is a bottleneck for the I/O-intensive SQL workload. Besides a further 25% slowdown (with encryption), it shows significant drops in throughput when limited I/O bandwidth causes the server to periodically delay transaction processing to allow checkpoint writes to complete.

Drawbridge and Haven exhibit relatively poor networking performance with Apache, because all socket operations traverse a security monitor in a separate process. Moreover, Haven performs substantially (40%) worse than Drawbridge with the host filesystem, because of many small file operations that flush the TLB. The private filesystem avoids this and even outperforms Drawbridge, since the workload is read-intensive and served almost entirely from the buffer cache inside the enclave.

**Sensitivity to SGX instruction overhead** Figure 5 shows the sensitivity of our workloads to SGX overheads. We vary the delay for either dynamic memory management instructions or enclave crossings while keeping the other at zero; the TLB is flushed on enclave crossings in all cases. SQL Server is sensitive to crossing overhead, with diminishing effects beyond 30k cycles, but unaffected by memory allocation overhead because few allocations occur in its steady state. The web server's throughput is sensitive to both parameters, because memory is allocated in request handlers, but drops less overall.

**Sensitivity to EPC performance** We artificially reduced the memory performance of our system, by lowering the DRAM clock rate from 1600MHz to 1067MHz.[8] This slowed the memory system by a third overall, but only reduced TPC-E throughput by 21%, and web throughput by 7%. We conclude that memory-intensive workloads are sensitive to EPC performance, but note that our experiment over-estimates its effect, since only some of the system's memory accesses would go to EPC.

**Summary** For now, we must speculate about performance of SGX implementations, but find our results encouraging: for large, complex, CPU and memory-intensive applications such as SQL Server, and for OS-intensive applications like a modern web stack, even given

[7] We used the 64-bit VC11 build from www.apachelounge.com.

[8] We reduced the DRAM clock multiplier, but kept the delay times (such as CAS latency, expressed in clock cycles) unchanged.

our inefficient prototype and assuming 10,000+ cycles for SGX instructions, Haven's performance penalty vs. a VM is 31–54%. We suspect that significant classes of users will readily accept such overheads, in return for not needing to trust the cloud.

# 7 Discussion

This section discusses various issues, starting with an analysis of the trusted computing base. We then cover future work, suggest SGX optimisations, and discuss general hardware support for shielded execution.

## 7.1 Trusted computing base

As Table 1 shows, the trusted computing base (TCB) of Haven is substantial, because the LibOS includes a large subset of Windows. However, in contrast to the current cloud model, all code in the enclave, and thus all code in the user's TCB, is under user control. They may use any means to achieve trust, including scanning for malware, code inspection, etc., and update it at will.

Ultimately, our goal is not to minimise the TCB, but rather to give the user equivalent trust in the confidentiality and integrity of their data when moving an application from a private data centre to a public cloud. In this regard, Haven addresses two real threats: a malicious employee of the cloud provider with either admin privileges or hardware access, or a government subpoena.

Besides the software TCB, Haven also relies on the processor's correctness. While a feature like SGX undoubtedly adds complexity, hardware (even microcode) is extremely hard for an attacker to modify, and hardware vendors perform significant validation to ensure correctness.

## 7.2 Future work

**Storage rollback** Haven does not currently prevent rollback of filesystem state beyond the enclave's lifetime. It cannot avoid the following attack: the enclave is terminated (e.g., the host fakes a crash), and its in-memory state is lost. A new instance of the enclave accessing the VHD is guaranteed to read consistent data, but not necessarily the latest version. Protecting against such attacks requires secure non-volatile storage [45]. Such storage may be located on other nodes, but the cost of network communication on every write is likely prohibitive. Instead, we plan to communicate only on "critical" writes (e.g., transaction commits) to balance this cost against the likely risks.

**Untrusted time** Our prototype relies on the host for system time and timeouts. However, a malicious host may lie about the time or signal timeouts early. We are planning two mitigations. One is to ensure the clock always runs forward. The other uses the cycle counter as an alternative time source; after calibrating it via network time synchronisation, we can check for early timeouts.

**Cloud management** Besides isolation, virtual machines can be saved, resumed and migrated. However, the implementation of these features depends on the host's ability to capture and recreate guest state, something that Haven explicitly prevents. We aim to support similar features cooperatively, using prior work that implemented checkpoint and resume at the Drawbridge ABI level [6]. In its simplest form, the host could request the Haven guest to suspend itself, which it would do by capturing its own state to an encrypted image. The host may then establish a new enclave to resume execution on another node. Before gaining access to the encrypted image, the new guest would perform an attestation step, giving it the keys necessary to access the encrypted checkpoint image. If the guest failed to complete these operations in a timely manner, the host could simply terminate it.

## 7.3 SGX optimisations

Besides the limitations identified in §5.4, two further opportunities exist to optimise SGX performance for Haven.

**Exception handling** As mentioned in §5.3 and §5.4, two aspects of SGX combine to substantially increase the overhead of exception handling: `ERESUME` and `IRET` are both illegal in an enclave. Haven's exception handler must `EEXIT` to a tiny stub that `ERESUME`s a modified context within the enclave. This then runs the LibOS and application exception handlers, which typically finish by executing `IRET` to restore the original context. However, this causes another illegal instruction exception. Overall, a single application exception (e.g., stack growth) results in two exceptions and eight enclave crossings. As there appear to be no insurmountable security implications, we suggest permitting (or providing equivalent replacements for) these instructions within the enclave.

**Demand loading** Haven's shield loads application and LibOS binaries. Modern systems typically load lazily: virtual address space is reserved, but pages are allocated and filled only on first access, in response to faults. Since other threads may also access the same pages while they are loaded, demand loading is done using a private memory mapping before remapping pages with appropriate permissions in the final location. However, since SGX

does not support moving an existing page, Haven must eagerly load all binaries. This adds time and memory overhead, particularly at startup. For example, running PowerShell until it displays a prompt causes 124MB of DLLs to be loaded, but only 4% of those pages are accessed.

The revised SGX specification includes an `EACCEPTCOPY` instruction [29], which allows a new page to be both allocated and initialised with a copy of data located elsewhere in the enclave before it becomes accessible to software. This should enable demand-loading, although we have not yet had the opportunity to experiment with an implementation.

### 7.4 Hardware for shielded execution

**Shielded VMs** As we noted in §2.1, SGX is the first commodity hardware that permits efficient multiplexing among multiple isolated programs without relying on trusted software. However, for many use-cases including cloud deployments, hardware capable of isolating full virtual machines (rather than portions of a user-mode address space, as in SGX) would be desirable from a compatibility standpoint: it would support complete guest operating systems. There are many performance and complexity-related challenges to building such hardware, including multiple levels of address translation, privileged instructions and virtual devices. However, if it were available, we suspect that a Haven-like shield module would be a suitable architecture to protect unmodified guest VMs from a malicious hypervisor, since the same trust issues addressed by Haven in the OS also arise in VM interfaces.

**Shielding without information leakage** Our definition of shielded execution (§2.1) requires confidentiality for intermediate state of the guest. As we noted (§3.1), SGX limits our ability to achieve this, because it exposes to the host information such as exceptions and page faults, and because side-channels such as cache footprint leak guest state information. At first glance, this concession seems necessary for the OS to dynamically manage resources. If all resources were allocated statically for the life of the guest, a host would have no reason to observe guest states. However, an OS relies on seeing application behaviour to efficiently multiplex resources over varying demands; e.g., by monitoring faulting addresses it can use page replacement algorithms to manage physical memory.

We conjecture that a hardware isolation mechanism supporting true shielded execution can in fact permit dynamic resource multiplexing by changing the role of the resource manager. Present mechanisms conflate determining the quantity of resources (e.g., the number of physical pages) to allocate with the selection of specific resources (the virtual-to-physical mapping). We propose decoupling these, giving the host control only over resource quantities, and allowing the guest to choose specific resources to relinquish when allocations change. For example, memory would be managed by allocating physical pages in the host, but allowing the guest to control its virtual mappings, and using self-paging [22] to permit oversubscription. The host may ask a guest to relinquish pages, and kill it if it did not meet a deadline. We anticipate that hardware could also support cache partitioning, achieving similar results to page colouring [64] without constraints on physical allocation; a host could flush and repartition caches without exposing guest access patterns.

## 8 Related Work

We survey related work in two areas: trusted hardware, and systems to isolate applications from an untrusted OS. Notwithstanding prior research [9, 12, 31, 34, 44], hardware security modules (HSMs) [1, 53], trusted platform modules (TPMs) [57] and ARM TrustZone [5] are presently the main hardware sources of trust on commodity platforms, and we focus on them.

**Hardware security modules** HSMs [53] are often used to protect high-value secrets (e.g., keys) in the cloud. An HSM is a protected computing element made tamper-proof using a physical barrier and a self-destruct mechanism to erase data if the barrier is compromised. Cloud HSMs such as AWS CloudHSM [1] offer APIs for key manipulation, signing, and encryption. As a result, the cloud user's keys are protected, but other data must still be transiently decrypted in a general-purpose node in order to use it. This reduces, but does not eliminate, the attack window compared to storing data persistently in the clear. As dedicated hardware, HSMs are also expensive.

**Trusted hardware** TPMs [57] are hardware devices included in many PCs supporting a similar attestation mechanism to SGX. The original approach to TPM-based attestation builds a chain of trust using progressive measurement of code during system boot, such as the bootloader, OS, etc. [50]. More recent CPU extensions enable the *late launch* and dynamic attestation of an isolated "secure kernel". This can reduce a platform's software TCB to just the late-launched code, a form of isolated execution, albeit one with two key drawbacks compared to SGX: vulnerability to relatively-simple hardware attacks including memory snooping, and lack of support for efficient multiplexing of distinct late-launch environments.

There are two general approaches to multiplexing TPM systems. The first, taken by Flicker [38], is to time-

multiplex the entire PC between secure kernels and an untrusted host OS. Unfortunately, because it uses a separate chip, TPM dynamic attestation is notoriously slow – Flicker's transition times are tens to hundreds of milliseconds for small modules. The second approach is to attest a trusted hypervisor or OS, which implements isolated execution in software [39, 49, 52]; the main downside for our scenario is that, regardless of its size, the hypervisor remains under the cloud provider's control. A cloud user may compare a TPM attestation to a known hash of the hypervisor binary, but we assume that the provider must be able to update the hypervisor (e.g., to patch security flaws, but also to insert arbitrary backdoors), and the user must ultimately trust them. This approach may be feasible given a hypervisor that is verified (down to binary code) to protect guest confidentiality and integrity, so that the attestation a user receives is meaningfully connected to a proof of the isolation mechanism, but current progress on OS-level formal verification is some way from this goal [23, 30].

A set of extensions in many ARM processors, TrustZone enables a "secure world" execution environment that is isolated from the OS [5]. Like the TPM, systems using TrustZone rely on software to multiplex the secure world; for example, to enable a runtime for security-critical components of mobile applications [51].

**Shielding apps from an untrusted OS** A number of systems seek to defend applications from a malicious OS. While XOMOS [35] used custom hardware, most recent approaches rely on the support of a trusted hypervisor. Proxos [54] runs isolated applications in a separate VM, but allows them to interact with a commodity OS. Overshadow [11] and SP$^3$ [60] pioneered transparent encryption of user memory when visible to the OS, protecting application data from direct tampering. CloudVisor [63] extended this technique to full VMs using nested virtualisation, while SecureME [12] accelerated it in hardware. More recently, InkTag [25] showed how to optimise the guest OS and protect persistent storage, and Virtual Ghost [14] used compiler techniques to implement a similar mechanism within the OS kernel.

However, systems based solely on protecting application memory from an untrusted OS are vulnerable to Iago attacks through the system call interface [10]. Systems such as InkTag [25] attempt to defeat Iago attacks by interposing on system calls (e.g., in a custom `libc`) and checking their results, but we feel that this approach is unlikely to be tractable for arbitrary applications given the complexity of modern OS interfaces – Linux today includes more than 300 system calls, and Windows well over 1000, as well as exceptions and asynchronous event mechanisms. Instead, Haven defeats Iago attacks by design, using a LibOS, shield module, and a substantially smaller (≈20 calls) mutually-distrusting host interface; it also avoids the need for a trusted hypervisor through SGX assistance.

**Cloud security** Finally, other research tackles the problem of removing trust from the cloud. Although fully homomorphic encryption schemes which allow arbitrary computation on encrypted data suffer intractably high overhead [20, 21], partially homomorphic encryption has been successfully applied in some domains; e.g., some database queries [4, 46] and MapReduce programs [55] can be implemented without ever decrypting data. While this cannot support existing applications, it also does not require trusted hardware.

MiniBox [33] combines the isolation of TrustVisor [39] with the sandbox of Native Client [61]. Like Haven, MiniBox achieves mutual distrust between application code and the host OS. Unlike Haven, MiniBox relies on a trusted hypervisor, and its isolated execution environment supports only small pieces of application logic, rather than complete unmodified applications.

PrivateCore vCage [48] is a virtual machine monitor implementing full memory encryption for commodity hardware by executing guest VMs entirely in-cache and encrypting their data before it is evicted to main memory. Although it relies on a trusted hypervisor, and thus cannot meet our security goals, it shares with SGX a resistance to memory probes and similar physical attacks.

# 9 Conclusion

Today's cloud platforms offer many advantages, but these are often outweighed by the risks inherent in a hierarchical security architecture: the provider is trusted with full access to user data. To eliminate this risk, Haven implements shielded execution of unmodified server applications in an untrusted cloud host. Haven brings us one step closer to a true "utility computing" model for the cloud, where the utility provides resources (processor cores, storage, and networking) but has no access to user data.

## Acknowledgements

We appreciate the assistance and collaboration of Intel Labs, especially Matthew Hoekstra, Simon Johnson, Rebekah Leslie-Hurd, Frank McKeen, Carlos Rozas and Krystof Zmudzinski. We are also grateful to all who provided feedback, in particular Steve Hand, Jon Howell, Rebecca Isaacs, Rama Kotla, Bryan Parno, Oriana Riva, Emmett Witchel and the anonymous reviewers.

# References


[1] *AWS CloudHSM Getting Started Guide*. Amazon Web Services, Nov. 2013. http://aws.amazon.com/cloudhsm/.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *6th Conference on Innovative Data Systems Research*, Jan. 2013.

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

[6] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *EuroSys Conference*, pages 239–252, Apr. 2013.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, pages 105–120, Aug. 2003.

[8] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Report 2006/052, Cryptology ePrint Archive, 2006.

[9] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *16th IEEE International Symposium on High-Performance Computer Architecture*, Jan. 2010.

[10] S. Checkoway and H. Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2013.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2008.

[12] 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.

[13] Cloud Security Alliance. Government access to information survey. https://cloudsecurityalliance.org/research/surveys/#_nsa_prism, July 2013.

[14] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, 2014.

[15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 339–354, Dec. 2008.

[16] K. Fu, F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, 2000.

[17] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *9th IEEE International Symposium on High-Performance Computer Architecture*, pages 295–306, 2003.

[18] B. Gellman and L. Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. *The Washington Post*, June 2013.

[19] B. Gellman and A. Soltani. NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say. *The Washington Post*, Oct. 2013.

[20] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[21] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *32nd International Cryptology Conference*, 2012.

[22] S. M. Hand. Self-paging in the Nemesis operating system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, 1999.

[23] 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*, Oct. 2014.

[24] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[25] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: secure applications on an untrusted operating system. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2013.

[26] F. Hou, N. Xiao, F. Liu, H. He, and D. Gu. Performance and consistency improvements of hash tree based disk storage protection. In *2009 IEEE International Conference on Networking, Architecture, and Storage (NAS 2009)*, pages 51–56, 2009.

[27] J. Howell, B. Parno, and J. R. Douceur. How to run POSIX apps in a minimal picoprocess. In *2013 USENIX Annual Technical Conference*, pages 321–332, June 2013.

[28] *Software Guard Extensions Programming Reference*. Intel

Corp., Sept. 2013. Ref. #329298-001 http://software.intel.com/sites/default/files/329298-001.pdf.

[29] *Software Guard Extensions Programming Reference, Rev. 2*. Intel Corp., Oct. 2014. Ref. #329298-002.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[31] 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.

[32] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in HYDRA. In *5th ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[33] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, June 2014.

[34] 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.

[35] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[36] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, 2000.

[37] 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, Oct. 1991.

[38] 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.

[39] 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.

[40] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf, 2004.

[41] 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.

[42] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO'87*, pages 369–378, 1987.

[43] C. C. Miller. Revelations of N.S.A. spying cost U.S. tech companies. *The New York Times*, Mar. 2014.

[44] 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.

[45] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

[46] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[47] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinksy, and G. C. Hunt. Rethinking the library OS from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, Mar. 2011.

[48] PrivateCore. Trustworthy computing for OpenStack with vCage. http://privatecore.com/vcage/, 2014.

[49] 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.

[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, Aug. 2004.

[51] 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.

[52] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[53] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9):831–860, Apr. 1999. ISSN 1389-1286.

[54] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation*, pages 279–292, 2006.

[55] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. In *2013 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–286, 2013.

[56] *TPC benchmark E standard specification*. Transaction Processing Performance Council, June 2010. Rev. 1.12.0.

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

[58] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *EuroSys Conference*, Apr. 2014.

[59] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give *me*? In *5th International Conference on Trust and Trustworthy Computing*, pages 159–178, June 2012.

[60] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *4th International conference on Virtual Execution Environments*, pages 71–80, 2008.

[61] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[62] A. Yun, C. Shi, and Y. Kim. On protecting integrity and confidentiality of cryptographic file system for outsourced storage. In *2009 ACM Workshop on Cloud Computing Security*, pages 67–76, 2009.

[63] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles*, pages 203–216, 2011.

[64] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys Conference*, pages 89–102, 2009.