Implementing Support for Intel SGX on the Barrelfish Research Operating System

Alex Upshaw

The University of Texas at Austin

Abstract

Intel SGX is an instruction set extension attempting to provide secure and trusted remote computation by furnishing protected execution and data environments for application software, called “enclaves”, with hardware confidentiality and integrity protection against attacks by even privileged software. In this work, we attempt to further the state of implementation of SGX support on the Barrelfish research operating system, a multikernel operating system designed like a message-passing, replicated-state, distributed system over the individual cores of a machine in order to deliver greater scalability with increasing core counts. A motivation for this work has been extending SGX support to Arrakis, another research operating system built from Barrelfish and designed to take the kernel out of the common data path of server software to improve performance. We are interested in exploring whether this kernel-bypass architecture lends itself to any stronger security properties then what can be achieved with more conventional operating systems.

1. Introduction

As public cloud and other hosted computing services grow more prevalent, concerns about the security implications of outsourcing the handling of one’s sensitive or critical data to third parties grow more important. Even if one trusts a provider not to act maliciously, one should still be concerned about the possibility that an attacker will compromise the provider’s infrastructure and machines. To address this problem, various solutions have been proposed over the years to the problem of providing secure trusted computing, typically rooted in rusted hardware.[2] Intel SGX is one such solution. It aims to allow software to create trusted “enclaves” in memory that are hardware-protected from offenses against their confidentiality and integrity, even from more privileged software.

Although SGX’s ability to protect against more privileged software sounds fairly impressive, it is by no means a panacea. As such, we are curious whether we will be able to find room for improvement over what security properties conventional operating systems achieve with SGX by implementing it on Arrakis, an operating system already designed to let user software go about its business without involving the kernel more than it needs to. Although data sent to I/O can already be encrypted under SGX, we are particularly interested in seeing whether removing the kernel from the general-case I/O path could help SGX applications keep their I/O metadata confidential as well.

Here, we begin working towards supporting SGX on Arrakis by expanding support on Barrelfish, the operating system from which Arrakis is derived.

2. Background

2.1 Intel SGX
Intel Software Guard Extensions (SGX) is a comparatively recent extension to the x86 instruction set offered by Intel, beginning with the Skylake microarchitecture in 2015. It has the goal of allowing software to protect its confidentiality and integrity from attacks by privileged software. To achieve this, SGX enables the creation of protected execution and memory environments referred to as “enclaves”. SGX introduces new memory access controls that disallow even privileged software from accessing protected enclave memory.

### 2.1.1 Enclave Structure

Code fetches from unprotected memory are disallowed when executing in enclave mode; therefore, at least the executable part of an enclave must reside in protected memory. Beyond that, however, what the enclave keeps in protected memory is up to the developer, as, in enclave-mode execution, reads and writes to both protected and unprotected memory are possible.[1] It would, however, be wise to keep anything that is important or confidential in protected memory.

**The Enclave Page Cache (EPC)** is the area of protected memory in which enclaves reside (figure 1). Details of what actually backs the EPC are considered implementation specific by Intel, but the EPC memory can generally be configured within certain parameters by the BIOS at boot time.[1] The CPU protects EPC memory from any unauthorized accesses. Although how it does so is also not explicitly specified by Intel, an example they give of what they might do (which can be presumed to be typical) is to encrypt EPC pages while they are stored in DRAM.[1]

The EPC is logically divided into 4KB pages that are allocated and freed by privileged software. Pages currently allocated and belonging to a specific enclave instance are “valid”, and other EPC pages are “invalid”.

Mechanisms also exist for overcommitment of EPC pages. The supervisor may evict and reload EPC pages with specific instructions. On page eviction, the CPU will ensure the contents of pages is encrypted before they are written out to unprotected memory. Furthermore, a combination of
message authentication codes and version numbers are used to verify that evicted EPC pages have not been tampered when system software tries to re-load them.[1]

Although the primary use of EPC pages is to securely back protected enclave memory, pages must also be allocated for certain sensitive SGX data structures that reside in protected memory, for which there are three specific EPC page type designations: SECS, TCS and VA.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
<td>8</td>
<td>Size, in bytes, of the virtual address range (in the enclave’s context) containing the enclave’s protected EPC memory.</td>
</tr>
<tr>
<td>BASEADDR</td>
<td>8</td>
<td>Base of the virtual address range (in the enclave’s context) containing the enclave’s protected EPC memory.</td>
</tr>
<tr>
<td>SSAFRAMESIZE</td>
<td>4</td>
<td>Size, in pages, of one SSA frame needed to save the enclave’s state on an asynchronous exit.</td>
</tr>
<tr>
<td>MISCSELECT</td>
<td>4</td>
<td>Bit vector specifying which extended features the enclave uses whose state will need to be saved and restored when entering and exiting the enclave asynchronously.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>16</td>
<td>Attributes of the enclave, including (1) whether it has been initialized, (2) whether it is a debug enclave (enables two SGX instructions, EDBGRD and EDBGWR, which are useful for debugging enclaves but wholly compromise their security benefits), (3) whether this enclave runs in 32-bit or 64-bit mode, and (4) XFRM, a bit mask specifying which extended features the enclave expects to be enabled and disabled when it runs.</td>
</tr>
<tr>
<td>MRENCLAVE</td>
<td>32</td>
<td>A cryptographic hash measuring how the enclave was built, its virtual address space layout, and its contents. This hash defines the identity of the enclave, is used to detect if system software tampered with it during construction, and may be used in key derivation.</td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>32</td>
<td>A cryptographic hash measuring the public RSA key used to sign an identity certificate provided when the enclave is initialized. This hash defines the identity of the enclave’s creator/owner and may be used in key derivation.</td>
</tr>
<tr>
<td>ISVPRODID</td>
<td>2</td>
<td>A product version ID number provided in the identity certificate that may be used in key derivation.</td>
</tr>
<tr>
<td>ISVSQN</td>
<td>2</td>
<td>A security version ID number provided in the identity certificate that may be used in key derivation.</td>
</tr>
</tbody>
</table>

**Table 1**: SGX Enclave Control Structure (SECS) fields of interest.[1] Note fields not of interest for this discussion are intentionally omitted.

The most important structure in specifying an enclave is an **SGX Enclave Control Structure (SECS)**. SECSs are EPC-resident data structures for meta-data used by hardware to describe the enclave. It serves as the CPU’s primary description of and reference to each enclave and cannot be directly accessed by software. The SECS for each enclave is created by the CPU at enclave-creation time in an EPC page allocated for the purpose by system software.[1]
The SECS records basic information, configuration, and other parameters which are initialized based on input from system software (see Table 1). Parameters of interest include a base address and size specifying the range of the enclave’s protected memory, enclave attributes, bit vectors defining what extended features are used by the enclave, and some information pertaining to saving the state of enclave threads.[1] The virtual memory range specified here for the enclave’s protected memory is referred to as the “ELRANGE”. While executing in an enclave, the CPU will assume that all memory in ELRANGE should be protected and all memory outside it should not. The size of this range must be a power of 2, and its base address must be aligned to that power.[1] These constraints are imposed in order to allow the CPU to quickly check whether a virtual address referenced by the enclave is from protected or unprotected memory.[2] Enclave attributes include whether this is a “launch enclave” (see 2.1.2.2), whether it is a debug enclave, and whether the enclave is intended to run in 64-bit or 32-bit mode.[1] If this is a debug enclave, then two instructions, EDBGRD and EDBGWR are enabled for this enclave. These instructions allow non-enclave software to read and write data in the enclave’s protected memory, which can be useful for debugging, but also happens to shatter all security benefits the enclave is intended to provide.[1] For that reason, it is particularly important that production enclaves are not marked as debug enclaves. The bit vector specifying the set of extended processor features used by the enclave is called the XSAVE Feature Request Mask (XFRM).[1] It is necessary for the enclave to specify and enforce the set of extended features it uses, which is important because enabling and disabling certain features may change the semantics of certain instructions, which could allow malicious software to attack enclaves by entering them with unexpected feature configurations.[2]

An enclave’s identity is also established and verified by SECS fields. Fields of specific interest include MRENCLAVE, MRSIGNER, ISVPRODID and ISVSVN. MRENCLAVE is a value representing the enclave’s identity, and MRSIGNER represents the identity of the developer or owner of the enclave; they are both 256-bit cryptographic hashes.[1] MRSIGNER measures the public RSA key used to sign a “SIGSTRUCT” provided and inspected when the enclave is initialized, which is essentially a developer’s identity certificate for the enclave.[1] What MRENCLAVE measures is somewhat more complicated. Its contents are initialized when the enclave’s SECS is first created, updated every time a new page is added to the enclave and measured, and finalized when the fully constructed enclave is initialized. Although the exact algorithm for computing MRENCLAVE will not be discussed in this work, it is known the hash will be depend on: (1) the exact sequence of enclave-creating and modifying instructions used in the enclave’s construction, (2) the size of the enclave’s SECS, (3) the size of its “SSA Frames” (to be described below), (4) the offset of every EPC page added to the enclave’s protected memory (relative to the base address of ELRANGE), (5) the R/W/X permissions of each such page, and (6) the offset (again relative to ELRANGE’s base) and contents of every 256-byte region of the enclave’s protected memory that system software chooses to measure (see 2.1.2.1 for more information on this process).[2] ISVPRODID and ISVSVN are, respectively, a developer-provided product ID and a security version number for the enclave provided by the developer in the SIGSTRUCT. SIGSTRUCT also specifies the value of MRENCLAVE the developer expects it to have if system software constructed and measured the enclave to the developer’s expectations without tampering with it.[1]

A Thread Control Structure (TCS) is needed by SGX for each thread that will run in an enclave. Like the SECS, a TCS is also a protected EPC-resident data structure used by hardware to keep metadata and state information. When a thread enters a created enclave, it must specify a TCS that it will run under. Only one thread may run under one TCS at any given time; when a thread enters the enclave under a TCS, that TCS is marked as busy, and this designation is not cleared until the thread leaves the enclave.[1] Attempts to enter an enclave specifying a TCS that is already busy will fail; however, it is possible to support concurrent threads in a single enclave by creating multiple TCSs for it.[1] TCSs may be created in empty EPC pages allocated by system software with certain supervisor
instructions. Each TCS also contains an entry address where all threads running on the TCS will enter the enclave.[1] This entry address is specified in the TCS rather than accepted from the threads themselves to prevent malicious software from trying to enter an enclave at an unexpected point in order to attack it, for example, by return-oriented programming.[2]

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>8</td>
<td>State of execution on this TCS, including whether or not it is busy (that is, whether there is a thread currently executing in the enclave under this TCS). Entering an enclave with a TCS that is already busy is disallowed.</td>
</tr>
<tr>
<td>OSSA</td>
<td>8</td>
<td>Offset of the base of the SSA stack, relative to the base of the enclave, where the CPU will store thread state if a thread must exit the enclave asynchronously.</td>
</tr>
<tr>
<td>CSSA</td>
<td>4</td>
<td>Index of the current slot in the SSA stack.</td>
</tr>
<tr>
<td>NSSA</td>
<td>4</td>
<td>Total number of slots in the SSA stack.</td>
</tr>
<tr>
<td>OENTRY</td>
<td>8</td>
<td>Offset, relative to the base of the enclave, of the enclave’s entry point when a thread enters under this TCS.</td>
</tr>
<tr>
<td>AEP</td>
<td>8</td>
<td>Virtual address for the Asynchronous Exit Pointer (AEP) set by the thread entering the enclave, where control will be transferred in the event the enclave must be exited asynchronously.</td>
</tr>
</tbody>
</table>

Table 2: Thread Control Structure (TCS) fields of interest.[1] Note fields not of interest for this discussion are intentionally omitted.

One important thing contained by each TCS is a base address, current index, and size for a stack of Save State Areas (SSAs).[1] As enclaves are preemptable when they run, it is possible the CPU may need to exit the enclave unexpectedly (called an “AEX”, see section 2.1.2.3). In such a case, it will need to save some thread state information, and it will need to save it inside the enclave to keep it secure. Each TCS therefore points to a stack of SSAs, generally contained in regular EPC pages, for use by the thread running on that TCS. The information saved by a single AEX occupies one SSA frame. The size of the SSAs for a given enclave is recorded in its SECS and initialized directly from system software’s input.[1] The size must, however, be at least large enough to accommodate the amount of state information produced by a single AEX; exactly how much that is may vary depending on another SECS field, called MISCSELECT, which specifies what extended feature state will be saved for the enclave on an AEX.[1]

The third type of special EPC-resident data structure is the Version Array (VA). Each VA page is subdivided into 512 8-byte fields, each of which can be used to store a version number for an evicted EPC page.[1] When evicting an EPC page, system software must allocate a free slot in a VA for the page’s version number to be recorded. The slot will be cleared when the evicted page is successfully re-loaded.

The CPU records the state of allocation of individual EPC pages in an Enclave Page Cache Map (EPCM). The EPCM is in on-die memory and never accessible to software; it is necessitated because the CPU cannot trust supervisor-written page tables alone for maintaining correct state under the SGX threat model. The EPCM has one entry for each EPC page, recording whether or not the page is valid, which enclave the page belongs to (if it is valid), what type of page it is (either regular, SECS, TCS, or VA), where in the enclave’s virtual address space the page is mapped, and read/write/execute permissions for the page.[1] The EPCM is managed by the CPU and updated
automatically when instructions changing the state of EPC pages are run; it cannot be accessed or modified by software; the CPU uses it in address translation and access control for enclave pages.

2.1.2 Enclave Creation and Life Cycle

Enclave software is distributed in the clear with certain digital signatures to verify its integrity. Allocation of space for enclaves, their creation, and their initialization is handled by supervisor-mode system software. Once they are created, enclaves can be entered, exited and resumed from user mode. Destruction of enclaves and freeing of resources they occupied is again handled by system software.

SGX introduces two additional opcodes to x86-64 machine language, ENCLS and ENCLU. ENCLS is a privileged instruction used by system software to perform functionalities related to managing enclaves. ENCLU is a user-mode instruction for actions in normal enclave operation. Which enclave operation is to be performed when ENCLS or ENCLU is executed is specified by a further code passed through EAX.[1] Note that, in addition to user space code obviously being prohibited from using the privileged ENCLS instructions, supervisor code is barred from using the unprivileged ENCLU instructions, which will fault in that case. This means that enclaves are not allowed to execute while in mode.

2.1.2.1 Enclave Allocation and Construction

Initial enclave creation is done by system software with the ENCLS instruction ECREATE. When calling ECREATE, system software specifies the address of an empty EPC page it has allocated for the SECS of the new enclave. The CPU then initializes the SECS in the specified page.

Once the SECS has been initialized by ECREATE, system software uses the ENCLS instruction EADD to load each page of the enclave into the EPC. The system software specifies the address of an empty EPC page and a pointer to a structure describing the contents of the page to be added.[1] In addition to specifying the SECS of the enclave to which the new page is being added, the structure gives an address in unprotected memory from which to load the contents of the new EPC page. If EADD is successful, the CPU updates its EPCM to reflect the allocation and addition made. In addition to regular EPC pages, any and all TCSs needed by the enclave must be created and loaded with EADD. TCSs are also loaded from a source page in unprotected memory containing a TCS structure with a defined layout; the instruction will fail if this structure does not specify a legal TCS.[1]

Although each call to EADD updates the MRENCLAVE value in the enclave’s SECS to reflect the fact that a new page was added to an enclave and where it is, the supervisor is expected to call EEXTEND, another ENCLS instruction, to update MRENCLAVE in order to reflect the actual content that has been loaded.[1] When calling EEXTEND, system software specifies the SECS whose cryptographic measurement is to be updated and a base address in EPC memory where the content should be measured. Each call to EEXTEND only updates the measurement to reflect 256 bytes of data after the base address provided, so system software is expected to call it iteratively to cover all the contents of the enclave that need to be measured. A thorough and oft-cited analysis of SGX, called “Intel SGX Explained”[2] suggests that the reason EEXTEND is a separate instruction from EADD and that it only measures 256 bytes at a time is that cryptographic hashing is a fairly expensive, but Intel requires each individual SGX instruction to execute within a strict latency budget.

2.1.2.2 Enclave Verification and Initialization

Before it can run, an enclave must be initialized with the ENCLS instruction EINIT. This initialization finalizes the enclave’s construction, including finalizing the value of MRENCLAVE. Further calls to EADD and EEXTEND to add more content to the enclave are disallowed after this
initialization, and entering the enclave to run it is disallowed prior to this initialization. Before setting a field in the enclave’s SECS to mark it as initialized (and therefore allowing it to be run), EINIT will scrutinize the enclave to verify that system software has not tampered with it during its construction; EINIT will fail if it finds something wrong with the enclave. System software must provide EINIT with two pointers to structures in unprotected memory used in verifying the enclave, specifically a SIGSTRUCT and a EINITTOKEN, which essentially serve as identity certificates for the enclave.[1]

**A SIGSTRUCT is the developer-issued identity certificate for an enclave.** It includes (1) a public RSA key for the developer, (2) a bit vector specifying which extended processor features the enclave uses, (3) a bit vector specifying the enclave’s attributes, (4) a field, called ENCLAVEHASH, specifying the value of MRENCLAVE expected if the enclave was built by system software to the developer’s specifications, (5) product and security version number IDs for the enclave, and (6) a signature over the SIGSTRUCT’s head and body with the included RSA key.[1] EINIT will fail if MRENCLAVE’s final value does not match the specified ENCLAVEHASH, if the enclave’s extended feature set and attributes do not match those specified in the SIGSTRUCT, or if the SIGSTRUCT’s RSA signature does not match its actual contents under the provided public key.[1] The purpose of this verification with SIGSTRUCT is to establish who the enclave’s developer/owner is and that the enclave has been constructed correctly by system software, without tampering, to the developer’s expectations. EINIT computes a hash measuring the developer’s public RSA key included in the SIGSTRUCT, called MRSIGNER.[1] If EINIT does not fail, it will store MRSIGNER in the enclave’s SECS, and the value will be used to represent the enclave’s owner for purposes such as attestation and key derivation.

The purpose of the EINITTOKEN, however, is not as clear as that of the SIGSTRUCT. According to the Intel System Developer’s Manual[1], the nominal purpose of the EINITTOKEN is to “verify that the enclave is permitted to launch”. Furthermore, this token is to be generated somehow by a special “launch enclave” (which can be initialized without its own EINITTOKEN). Assuming one has gone through a launch enclave to obtain an EINITTOKEN, it will give the enclave’s attributes, feature selection, MRENCLAVE, and MRSIGNER; EINIT will fail if the values given in this token do not match those of the enclave being initialized. Furthermore, for EINIT to accept the EINITTOKEN as valid and initialize the enclave, it must also contain a message authentication code produced with a special secret, called EINITTOKENKEY, which is only accessible to the launch enclave.[1]

**“SGX Launch Control” refers to the process of getting an EINITTOKEN from the launch enclave.** Documentation about this process is rather meager in the Intel Manual[1], although it appears that it is intended to facilitate the enforcement of a launch-control policy. At first glance, this might appear to be a redundant feature. System software is already responsible for allocating space for and loading enclaves, and enclave software is distributed in the clear, so it is fairly easy to conclude that system software can already enforce an arbitrary launch control policy without using a launch enclave (The “SGX Explained” analysis[2] also comes to this conclusion, and makes the argument in much greater depth than I have here that SGX Launch Control is an unnecessary feature). One interesting constraint on the launch enclave that is mentioned in the manual’s[1] brief section on launch control is that the hash of the public key used to sign the launch enclave’s SIGSTRUCT must match a hash stored in IA32_SGXLEPUBKEYHASH, a set of model-specific registers. By default, these registers contain a hash of Intel’s enclave signing key. It can therefore be inferred that SGX launch control is intended to enforce Intel’s launch control policy, whether system software and the enclave developer like it or not. The “SGX Explained” analysis[2] goes very far in corroborating the fact that SGX launch control is intended to serve Intel’s interests, likely in opposition to the interests of their users. The analysis examines some incriminating SGX-related patents of Intel’s, which suggest that, at least at some point (possibly early in SGX’s development, however), Intel considered using SGX launch control to enforce a licensing scheme requiring that enclaves only be allowed to run if their authors have a business relation with Intel[2].
It appears that there is at least some hope that a workaround for SGX launch control may be offered, however, as the current version of the Intel Software Developer’s Manual[1], does mention that certain processor models may allow the BIOS to reconfigure the IA32_SGXLEPUBKEYHASH to authorize the use of a non-Intel launch enclave.[1] Furthermore, an Intel open source project providing SGX support for Linux,[3] includes a reference launch enclave for modification and use with a “Flexible Launch Control (FLC)”. We presume Intel intends to add this feature with future processors, although we are not aware of any specific statements or documentation about what models will or do support FLC.

2.1.2.3 Enclave Operation

Enclaves are entered from unprotected user code in the enclave’s context. EENTER, an ENCLU instruction, accepts the virtual address of a valid, non-busy TCS that the entering thread will run under.[1] If EENTER does not fault (for example, because a busy TCS is given, because the enclave has not yet been initialized, etc.), then the CPU will flush all cached linear-to-physical mappings, switch to enclave execution mode, load the value of XFRM from the SECS into XCR0 (switching to the set of enabled extended features expected by the enclave), set the TCS as busy, and jump to the entry address specified in the TCS.[1] Before loading the XFRM value, however, EENTER will back up XRC0’s value so the feature set the caller was using before entering the enclave can be restored later. One thing that the EENTER instruction itself does not switch is the stack pointer; enclave software is responsible for changing its stack pointer on entry to point to a stack inside the enclave. Most registers also are not automatically changed on entering the enclave, so they can be used by the user application invoking the enclave to pass parameters in.

Enclaves return to their unprotected user context with EEXIT. This causes the CPU to exit enclave-mode, return to normal user mode, restore the backed up value of XRC0 for the caller, and make a jump. However, unlike EENTER, which always jumps to an entry point specified by the TCS, EEXIT accepts a target address from the enclave software to jump to when it exits.[1] If the enclave wishes to return control to the caller like an ordinary function call, EENTER automatically outputs the instruction pointer value that follows EENTER to RCX; however, it is the responsibility of the enclave software to record this value for use when exiting. The fact that EEXIT can jump to an arbitrary location outside of the enclave could allow the developer to define and implement a convention to enable the enclave to make calls to procedures outside the enclave,[1] although, in this case, the developer would also need to establish a convention for how such procedure calls return to the enclave, given that they must enter it through EENTER again at an address specified in the TCS. EEXIT does not change most general-purpose registers. This means they can be used to return information out of the enclave, but it also means that it is the responsibility of the enclave to clear any information from its registers that it considers confidential before exiting, and backup and restore restore any registers that are considered callee-saved under the convention between the enclave and its caller.

Although the CPU’s general-purpose registers can be used to pass parameters and return information between the enclave and its caller, it should be noted that they are not the only way. In particular, since the enclave is allowed to read and write unprotected memory (although not fetch code from it), arbitrary regions of unprotected memory in the enclave-calling process’s virtual address space can be used for data transfer between its unprotected application software and the enclave.

Although it is confidentiality and integrity-protected, enclave software is still preemptable. [1] To handle this, the CPU may perform an Asynchronous Enclave Exit (AEX) in the event that it there is a fault, interrupt, or other event that calls for the CPU to switch control to a handler outside the enclave. When an AEX occurs, the CPU will back up the enclave thread’s registers to an SSA frame in the SSA frame stack referenced in the thread’s TCS.[1] Because these registers may contain confidential information, the CPU will also load a synthetic state before it switches out of enclave
mode and transfers control to an external handler. The synthetic state is designed to clear all information that might be confidential from the registers, and it is also designed so that system software’s handler of the relevant event will give control to a user space AEX handler in the process that entered the enclave.[1] To support this, EENTER takes as a parameter an Asynchronous Exit Pointer (AEP), which gives an address for the caller’s AEP handler. EENTER also makes backups of the caller’s RSP and RBP when it is called.[1] During an AEX, once the CPU has backed up the enclave’s state to an SSA, it will load the backed-up values for the caller’s RSP and RBP, and also load the given AEP to RIP, thereby pointing the synthetic state towards the enclave-calling thread’s stack and provided AEX handler.[1] Although most information about the enclave’s internal state is preserved, in the event the AEX was triggered by a page fault in the enclave, the virtual address causing the fault is left mostly intact, but with the low 12 bits cleared.[1] This allows system software to determine which page the fault occurred on (which is useful because system software may evict pages from the EPC), but does not unnecessarily disclose where in the page the enclave was specifically accessing.

The instruction ERESUME must be used to re-enter an enclave after an AEX. Like EENTER, ERESUME takes the virtual address of the TCS and an AEP to call if another AEX occurs. If ERESUME is successful, then the CPU will switch to enclave mode, the enclave thread’s state will be loaded from the SSA where it was saved, and it will resume execution where it was interrupted.[1] Alternatively, the AEX handler may choose to EENTER or ERESUME somewhere else in the enclave with a different TCS (if one is available). This can theoretically allow the user space AEX handler to invoke another exception handler within the enclave itself to attempt to deal with the cause of a fault, although establishing a particular convention for such a procedure is left to the developer.[1] If one were to attempt such a thing, it is also of note that, assuming the SSA is in a regular EPC page within the enclave, the enclave can examine the SSA frame from which a fault may have originated in addressing whatever the issue may be.

2.1.2.4 Enclave Page Eviction and Return

SGX allows system software to overcommit the EPC and evict and restore its pages.[1] Pages evicted from the EPC are written out to unprotected memory, where system software may further evict them to disk if necessary. Encryption and other protections (including storing a version number of each evicted page in a VA page) are enforced by the CPU to ensure that EPC page eviction does not compromise the confidentiality or integrity of protected memory.

If an enclave tries to access one of its EPC pages that has been evicted, it generates a page fault like it would for a normal memory page. As described in section 2.1.2.3, system software can still determine which page such a fault occurred on if it wishes to re-load the page in reaction to such a fault.[1]

Before evicting an EPC page, system software is expected to allocate a slot in a VA page in the EPC for storing the evicted page’s version number and to unmap the page from page from the enclave’s context.[1]

Once the page is unmapped, system software should next use the ENCLS instruction EBLOCK, passing it the virtual address of the EPC page to be evicted. EBLOCK causes the page to be marked as blocked in the EPCM, which will prevent the creation of any new TLB translations referencing the page.[1]

System software is expected to run the ENCLS instruction ETRACK before it actually evicts the page. It must give the virtual address of the SECS page corresponding to the enclave to which the EPC page being evicted belongs. ETRACK causes the processor to set up some internal state to track whether the TLB has been flushed for logical cores executing the enclave specified by this SECS. It is necessary for hardware to track this because system software is expected to ensure that all
TLBs possibly referencing the page being evicted have been flushed, but system software is not trusted under SGX’s security model, so it must be verified that it actually does this. Once hardware is tracking this invalidation, system software needs to actually induce it. In particular, system software must issue an inter-processor interrupt to each logical processor executing code in this enclave, which will trigger an AEX, flushing the TLB.

**Once these preparations have been made, the instruction EWB is used to actually make the eviction.** EWB accepts three virtual addresses from system software, one pointing to a structure describing both the page in unprotected memory where the evicted page is to be written out to and a 128-byte buffer for storing metadata about the evicted page, one pointing directly to the EPC page being evicted, and one pointing to the empty slot in a VA page that has been allocated for this page’s version number. EWB will fail if it sees that the state information set up by ETRACK suggests that the relevant TLB(s) have not been flushed. If EWB succeeds, it will write the encrypted contents of the evicted page to unprotected memory as requested and record a version number for it in the VA slot. EWB also writes a Paging Crypto Metadata (PCMD) structure to the 128-byte buffer system software provided. In addition to metadata, the PCMD contains a message authentication code used to verify that neither the contents of the evicted page nor the contained metadata have been tampered with. The version number stored in a VA page is essentially a nonce to prevent malicious system software from evicting a page from the EPC then trying to re-load an older version of it.

System software may choose to batch multiple page evictions into a single cycle of invalidating and flushing TLBs for efficiency reasons by issuing multiple EBLOCK commands, followed by one or more ETRACK commands and inter-processor interrupt TLB shootdowns, followed by multiple EWB commands.

**To re-load an evicted page, system software uses one of two ENCLS instructions, ELDB or ELDU.** Both instructions take essentially the same three pointers as EWB takes, except in this case the buffers in unprotected memory for the encrypted page contents and PCMD, and the VA slot are populated rather than empty, and the page in the EPC is empty rather than valid. ELDB or ELDU will fail if the message authentication code and version number do not validate given the page metadata, encrypted contents, and version number, suggesting system software tampered with it while it was evicted. The difference between ELDB and ELDU is that, if successful, ELDB will re-load the page in a blocked state, whereas ELDU will re-load it in an unblocked state.

Additional considerations for EPC paging include that evicting a SECS page is not allowed until all other pages in its enclave have been evicted, that re-loading a page belonging to an enclave is not possible while that enclave’s SECS is evicted, that it is possible to evict VA page if another VA page is available to store a version number for the evicted one, and that it is not possible to re-load an evicted EPC page while the VA page containing that page’s version number is also currently evicted itself.

### 2.1.2.5 Enclave Deconstruction

To tear down an enclave that is no longer wanted and free the EPC pages it occupies, system software repeatedly calls the ENCLS instruction EREMOVE for every EPC page in the enclave. EREMOVE simply takes the virtual address of an EPC page and marks it as invalid (that is, unallocated). EREMOVE can fail, however, in the event that there is still a thread executing in the enclave whose page is being deallocated. EREMOVE will also fail to deallocate a SECS page if there are still any EPC pages that belong to that SECS, so an enclave’s SECS must be the last part of it that is deallocated.

### 2.2 Barrelish
Barrelfish is a research operating system with an architecture described as a “multikernel”. It is designed for scalability on multicore systems, and, at a high level, it can be described as having a local microkernel on each core of the system, a distributed system of user-space monitors that maintain replicated state, rather than shared state, and a number of traditional OS services provided by user-space processes.

Barrelfish is inspired by observation of a number of hardware trends that more traditional monolithic-kernel operating systems seem to be ill-suited to deal with efficiently. Specifically, core counts are increasing, and architectural tradeoffs found among modern systems are growing more diverse. These trends pose at least two challenges to traditional monolithic kernels. First, increasing core count increases the latency costs of cache-coherence, thereby making shared memory and lock contention less scalable. Second, traditional operating systems often rely on being hand-tuned and optimized to perform well on the architectures they run on, which grows increasingly intractable as architectures grow more varied, and even the rate at which new hardware diverges from old hardware increases.

Barrelfish restructures the OS as a distributed system among the individual cores. Furthermore, it employs three design principles to address these challenges: (1) making all inter-core communication explicit, (2) structuring the OS to be hardware-neutral, and (3) considering most OS state to be replicated between cores, rather than shared. Making the OS hardware-neutral reduces the burden of adaptation to and optimization for ever more diverse different architectures. Maintaining a replicated state with explicit message passing allows inter-core communication to be asynchronous, which is much more scalable as core count increases than attempting to synchronously maintain shared state on an ever-increasing number of cores. A further advantage of Barrelfish’s architecture is that it facilitates transferring insights from the field of distributed systems, which is growing more relevant because single computers are coming to resemble networked systems. For example, modern multiprocessors already internally perform message passing rather than sharing a single interconnect for scalability. Beyond better exposing and facilitating reasoning about the latency of inter-core communication in modern multiprocessors, the messaging passing abstraction also facilitates extension of support to systems without cache coherence or even shared memory.

Barrelfish consists of a minimal kernel on each core, called a “CPU driver” along with a user-space monitor instance. In order to achieve comparative hardware-neutrality, interfacing with the hardware is delegated to the CPU driver, and policy is delegated to the monitor. The monitor (and other user-space processes) can thus implement all system policy in as hardware-agnostic a manner as possible, while the CPU driver is responsible for dealing with the hardware at a low level and providing an abstracted interface for its basic functions to the monitor. The monitors are responsible for maintaining global replicated state, which they do with a distributed agreement protocol.

As the CPU drivers have no shared state and are responsible almost exclusively for mechanism and not policy, they are free to be comparatively simple. CPU drivers on Barrelfish are, in fact, event-driven, single-threaded, and non-preemptable. One particularly important function provided by the CPU drivers is efficient inter-core communication, which Barrelfish uses heavily because it is a distributed message-passing system. It is important that the CPU driver be responsible for this because efficient inter-core communication tends to be reliant on architecture-specific optimizations. Barrelfish’s CPU drivers are optimized to deliver cache-line sized messages between cores through a shared memory region while minimizing interconnect utilization needed to transfer this message.

Processes in Barrelfish are represented by dispatcher objects, which are core-local; a process must have a dispatcher on each core on which it might execute. Communication in Barrelfish occurs between dispatchers, thereby directly exposing inter-core communication. Dispatchers are run by the CPU driver on their core through an upcall interface they provide; typically, the dispatcher will also run a user-level thread scheduler on top of this interface.
Barrelfish uses a system of capabilities in enforcing state consistency.[4] Capabilities are kept and enforced by the CPU drivers, and all memory allocation and virtual memory management is performed by user-mode software referencing these kernel capabilities. Although the capabilities are core-local kernel objects, the monitors implement a method to transfer them between cores.[4]

2.3 Arrakis

Arrakis is research operating system built upon the Barrelfish research operating system.[5] Although we have not yet reached the point of adapting Barrelfish’s SGX support to Arrakis, a brief discussion of Arrakis may be warranted as ultimately adding SGX features to Arrakis has been a motivation for our work on SGX support in Barrelfish.

Arrakis is designed to use virtualization technologies to remove the need to enter the kernel in common-case I/O operations to improve performance without sacrificing the protection normally afforded by an OS.[5] Arrakis is described as operating in the control plane for I/O, whereas the data plane is mediated by virtualization-enabled hardware without the active assistance of the kernel. Arrakis exploits Single-Root I/O Virtualization (SR-IOV) support, that is, I/O devices that are capable of presenting themselves to software as multiple configurable virtual devices. Arrakis creates a virtual device for each user space process that the process can access directly as if it were accessing physical hardware. This allows the process to go directly to the device for its I/O needs without needing to go through the kernel.[5] Furthermore, Arrakis can still enforce OS-like protection features in deciding how these devices are multiplexed through the configuration of the virtual devices; the controls will then be enforced by the hardware without the need to involve the kernel in the normal I/O path. Although the cost of entering the kernel to do I/O has historically been negligible because I/O devices have been so much slower than the CPU, the performance benefits of removing the need to enter the kernel from the normal I/O path can be substantial for I/O-intensive workloads running on fast I/O hardware such as a 10Gb network card. Severalfold performance improvements have been demonstrated with Arrakis in such cases.[5]

Arrakis enables largely unmodified applications to take advantage of direct I/O by providing POSIX-compliant user libraries implementing a user network stack and file system. It is, however, possible to gain a bit more performance by adapting applications to a native Arrakis interface that enable asynchronous I/O and true zero-copy I/O, which are not possible within the POSIX standard.[5]

3 Support for SGX on Barrelfish

We received a branch of Barrelfish with partial SGX support from ETH Zürich as a starting point for our implementation. ETH Zürich undertook this development on physical hardware; however, for want of SGX hardware and to facilitate early-stage debugging, we have been developing on an emulator with partial support for SGX, specifically a version of QEMU modified within our department.

3.1 Support Previously Implemented at ETH Zürich

The work towards SGX support that was done at ETH Zürich includes definition of Barrelfish capabilities for the EPC and its pages, SGX feature detection code, addition of new system calls to the x86_64 CPU driver to handle ENCLS instructions as capability invocations on EPC pages, definition of a user-space library for SGX support, and partial implementation of the system calls and of the library.
The new EPC capability types are fairly straightforward; first, one large capability is defined for the entire EPC, which is created from the capability for the physical address space (PhysAddr) in the event that SGX support was in fact detected. The feature detection code also determines what range the EPC occupies in physical address space and therefore what range the EPC capability should be created with. From this EPC capability, EpcReg capabilities are created for the individual pages in the EPC. From EpcReg capabilities, specialized types of EPC capabilities can be created, including capabilities for SECSs, TCSs, VAs, and SSAs. There are also capability types for the virtual-to-physical mappings of each of each of the EPC page types.

With respect to the ENCLS system calls, ECREATE and the debug instructions EDBGRD and EDBGWR were fully implemented, and EADD was mostly implemented, although the system call did not yet validate the input for non-regular EPC pages (TCS pages in particular) before executing the actual EADD instruction with it. Functions had also been set up for EEXTEND and EINIT, but not implemented.

The user space SGX library defines five functions for the creation and operation of enclaves, sgx_create, sgx_remove, sgx_start, sgx_resume, and sgx_exit. Of these functions, sgx_create had a partial implementation, and the other four had no implementation (other than placeholder return statements). The sgx_create function’s partial implementation accepted an “sgx_enclave” structure to be initialized as the enclave is created as well as an ELF image of the enclave to be loaded. The existing implementation validated the provided ELF, allocated a page for and filled out the input structure for the enclave’s SECS, allocated pages for a TCS and for the enclave’s contents specified by the ELF, called ECREATE to create the SECS, then EADD to add the pages in the enclave. After that, it returned. Notably, it did not handle measuring the enclave with EEXTEND and initializing it with EINIT. Another deficiency in sgx_create is that, although it initialized the data structures for all the enclave’s pages and it called EADD for each one of them, it did not actually load the contents of the enclave’s ELF file into the buffers in unprotected memory it allocated to be passed to EADD, and, consequentially, all the pages it added to the enclave were empty.

In addition to this implementation, ETH Zürich provided a simple test program for their code which calls sgx_create to load a simple enclave.

### 3.2 Bug Fixes and Migration to QEMU with Partial SGX Support

We have been developing SGX support for Barrelfish on a modified version of QEMU. QEMU is a popular processor and system emulator (although it could also be described as a type-2 hypervisor) that utilizes dynamic binary translation to provide CPU and platform virtualization without requiring hardware virtualization support, and to provide emulation of CPUs and platforms that may differ from the host platform. The version of QEMU we are using has been extended by others in our department (on a previous occasion) to emulate Intel SGX on systems that do not support SGX.

Additions to this version of QEMU include (1) enforcement of the additional access controls introduced by SGX that deny non-enclave software direct access to the EPC, (2) maintenance of several new data structures mimicking those used by real SGX hardware (such as an EPCM), and (3) addition of two routines (one for ENCLU and one for ENCLS) that the code generated during binary translation will invoke to emulate SGX instructions. These two routines, in turn, select one of a number of helper functions to call that correspond to the individual SGX operations, such as EINIT or EENTER. This structure also allows us to conveniently insert debugging code into the helper function for a specific SGX operation in order to determine why that operation is faulting or behaving unexpectedly when Barrelfish calls it. Furthermore, the emulator counts how many times each SGX instruction is run on it and prints these counts to the console when it is exited, which can be helpful for debugging.
Other than lacking support for certain SGX instructions which we are not yet interested in using, we describe this emulator’s support for SGX as “partial” because it does not actually enforce the cryptographic checks that EINIT will on real hardware before clearing an enclave to run. Although this does mean that we will still need to develop and test support for this authentication in order to migrate from this emulator back to real hardware, it is somewhat convenient for our purposes at the moment because it means we can test our SGX support without needing to create identity certificates for our enclaves, and it also frees us of the need to use a launch enclave to launch enclaves.

We have migrated the SGX support we were provided by ETH Zürich to run on our modified version of QEMU. Our first order of business was to get ETH Zürich's provided test to run on our emulator.

When we first attempted to build and launch this test on the emulator, Barrellfish failed to detect SGX support on the emulator. Initially, we assumed that this was due to a bug in the emulator’s responses to CPUID queries made by Barrellfish, so we begun carefully examining what responses Barrellfish expected to its queries and how the emulator was coming up with its responses. The root cause of the issue, however, turned out to simply be that QEMU was not being invoked to emulate an SGX-capable CPU. To remedy this, we added an argument requesting a Skylake processor be emulated to the QEMU command used by Barrellfish when building and running itself on QEMU (Barrellfish’s build system directly supports building an appropriate virtual disk image and launching it on QEMU).

Next, once Barrellfish correctly detected SGX support on our emulator, running the test on our emulator triggered a general protection fault. Unfortunately, although we have resolved it now, debugging this test and the underlying SGX support proved to be time consuming, in part due to initial unfamilariy with SGX and Barrellfish.

The instruction pointer dumped at the time of the fault suggested it occurred when memset was called. We also examined the stack to determine where and how memset was being called. Barrellfish does not have a convenient function to automatically reconstruct the logical call stack (like, for example, dump_stack in the Linux kernel); it only has a function to dump the raw data off the top of the stack in hexadecimal. To make this data more useful, we modified the function to dump a much larger section of the stack, and to compute unrelocated addresses for every value dumped, using logic that already existed for computing the unrelocated address corresponding to the instruction pointer. Initially, we interpreted these stack dumps by hand, comparing the unrelocated addresses seen in the stack with a disassembly of the binary file for the CPU driver. While interpreting these stack dumps, we quickly came to think it might be worthwhile to invest time in automating this process. We produced a simple Java program that compares the raw stack dump to the disassembly file and automatically appends function names to any line in the stack whose value, interpreted as an unrelocated address, falls within a function listed in the disassembly.

Misguided suspicions about the cause of this fault included that the fault originated from trying to execute the memset code, rather than trying to write to the memory it was invoked to set, and that the fault was somehow caused as a result of the address being written to not being correctly mapped into memory rather then a result of it being in protected memory. Contributing to this particular misguided inquiry was an initial misconception that the EPC itself is dynamically constructed by system code from unprotected memory by dynamically offering pages to SGX’s memory encryption engine.

The actual cause of this initial fault turned out to be a memset call in Barrellfish’s capability initialization code that attempted to zero-out the contents of an EPC page when the capability for that page is created. It appears this memset was copied from initialization code used for unprotected memory regions, some of which was reused in the EPC page initialization code. Attempting to zero-out an EPC page fails because the EPC is not accessible from outside an enclave. We have since removed this memset call to prevent the fault.
After resolving this particular fault, it was followed by several more general protection faults like it. These faults, too, were caused by attempting to zero-out specific types of pages in the EPC. We have now removed the memset calls inducing each of these faults.

A further issue we identified and resolved was that, prior to a loop that calls EADD for each page in the enclave, a computation of the number of pages the enclave has was commented out and replaced with the value 1, presumably to attempt only a single EADD for debugging purposes.

After resolving these issues, we were able to run the SGX test developed at ETH Zürich to termination without fault or exception, though it should be noted that this does not necessarily mean that the enclave’s construction was completely correct at this point, as it is possible to construct enclaves that cannot be initialized or that are unrunnable without causing an explicit fault prior to attempting enclave initialization or entry.

3.3 Further SGX Support We have Added to Barrelfish

We have added basic support for EEXTEND and EINIT on Barrelfish. Furthermore, we have added the necessary code to sgx_create to ensure the enclave’s contents are actually loaded.

Although the work previously done at ETH Zürich defined the system calls for EEXTEND and EINIT, they contained only placeholder code and had not yet been implemented. We have implemented them, which was fairly straightforward in that it just required defining parameters for, passing, and using all the values needed to execute each ENCLS instruction. One feature our implementation is lacking at this time is validation of the parameters received by the kernel from user code. After completing these system calls themselves, we extended the code in sgx_create to use them appropriately.

Under our current implementation, sgx_create calls EEXTEND for every 256-byte region of every page added to the enclave. EEXTEND is called for the regions of each single page in ascending order of offset from the page base. These calls are made for each page after EADD for that page, but before adding the next page. This ordering is significant because the value of MRENCLAVE is sensitive to the exact order in which all the calls made to EADD and EEXTEND occur. If we are to add support for cryptographic validation of the enclaves at EINIT time, or if we move our current implementation to physical hardware, it will be necessary to ensure that the EADD and EEXTEND calls occur in the exact order the enclave developer expects them to in order to produce an MRENCLAVE value that matches their SIGSTRUCT. This could be achieved either by requiring developers assume, by convention, that the instructions will be executed in the order sgx_create currently executes them, or it could be achieved by adding functionality to accept, from the developer, and adhere to some file or structure explicitly specifying what order they expect enclave construction instructions to occur.

Once the enclave is constructed, sgx_create now calls EINIT to initialize it. Currently, placeholder values (specifically null pointers) are given for the SIGSTRUCT and EINITTOKEN; however, the emulator we are developing on accepts this because it does not actually enforce cryptographic validation of enclaves. If we wish to support this validation and/or migrate to physical hardware, we will need to do two more things prior to running EINIT. First, we will need to provide a means of accepting a SIGSTRUCT from the enclave’s developer with which to initialize the enclave. A program for aiding enclave writers in computing appropriate SIGSTRUCTs for our platform might also be warranted, as computing a SIGSTRUCT is non-trivial and is a function handled by Intel’s provided SDKs when developing enclaves for production operating systems (Linux and Windows). Second, we would need to implement some means of obtaining a correct EINITTOKEN for the enclave, possibly involving the addition of a launch control service to Barrelfish that runs the launch enclave and accepts
token requests. The difficulty of providing such a service is compounded by the fact that it appears launch enclaves must generally be signed by Intel (see section 2.1.2.2).

In addition to calling EEXTEND and EINIT, we have also extended sgx_create to actually load the enclave contents into the buffers it passes to EADD from the enclave ELF file in order to ensure the enclave content actually makes it in to the enclave. In addition to adding a call to an ELF-loading function already present in Barrelfish, we have also adjusted how sgx_create allocates memory for these buffers in order to support this. In particular, the buffers are now allocated on demand in response to callbacks from the ELF-loading function requesting memory for loadable program sections. Formerly, each page’s buffer was allocated independently around the time the structures referencing it were first initialized. We have changed this because the ELF-loading function requires some multi-page regions to be contiguous; our implementation creates each region requested by the ELF-loading function, then iterates through the structures for each EPC page that part of this buffer will be loaded into and points them towards this buffer.

4 Future Work

Somewhat obviously, we would like to finish implementing support for basic SGX features on Barrelfish. We would also be interested in running SGX on Barrelfish on physical hardware rather than on an emulator at some point; however, it would seem as though SGX Launch Control could wind up being a substantial hindrance to moving to physical hardware.

Furthermore, we are particularly interested in integrating SGX support with Arrakis features to explore whether Arrakis’s kernel-bypass architecture could facilitate any improvements in enclave security, especially in the area of I/O channels. We would like to see if it is possible to keep I/O metadata confidential from system software once the kernel has been removed from the I/O path.

5 Related Work

To the best of our knowledge, there has not been any work done on support for Intel SGX on Barrelfish and Arrakis outside of that previously done at ETH Zürich and the contributions of this work.

Generic support for SGX on other, more widely used, operating systems, specifically Linux and Windows, is provided by Intel. In addition to SGX kernel drivers, Intel also provides platform software and an SDK for SGX on both these platforms.

Beyond this basic support, various works have been created to offer further security, better performance, and ease of use for SGX on these platforms. Two projects that have been particularly influential are Haven[6] and SCONE[7].

Haven is a project from Microsoft Research intended to enable the protection of unmodified legacy binaries by running them in an enclave on Windows.[6] Haven achieves this by including a library OS that implements the Windows API in the enclave with the legacy binary. The library OS, in turn, relies on a shield module that implements very basic OS primitives through a mutually-distrusting interface with the host OS. It also provides secure persistent storage to enclaves through an encrypted virtual disk image.

Another reason Haven is notable is for inspiring further additions to SGX. SGX was originally conceived as a means to shield only some security-sensitive parts of larger applications. As such, under the original SGX specification, there are certain limitations that make it impossible to port arbitrary legacy binaries into enclaves in the general case (for example, certain instructions are not allowed when in enclave mode, and there is no way to dynamically change the permissions of a running enclave’s
EPC pages). However, the Haven project motivated a further extension to SGX, called “SGX2”, which Intel will support with future CPUs. SGX2 makes a number of changes needed to support running legacy binaries inside enclaves, perhaps the most significant of which is a mechanism for dynamically offering additional EPC pages to a running enclave.

SCONE[7] enables running Docker containers inside enclaves on Linux. Beyond providing a convenient means of migrating applications to SGX for Linux, SCONE also focuses on minimizing both the size of the trusted computing base (TCB) it requires and the performance overhead it incurs. SCONE exposes a standard library interface to containers, which it then supports by making system calls out of the enclave. Similarly to Haven, SCONE distrusts the results of the system calls and verifies them before returning them to the caller. Performance optimizations made by SCONE include user-level threading and asynchronous system calls, which are useful because the overhead of leaving and re-entering an enclave is considerable. SCONE also provides a number of “shields” (although it should be noted its use of the term differs from Haven’s) for protecting I/O. These include transparently encrypting files that are written out, networking through TLS connections terminated inside the enclave, and encrypting output to the console.

6 Conclusion

We believe that Arrakis’s kernel-bypass architecture shows promise for allowing SGX enclaves to achieve security benefits not available from conventional operating systems, such as confidentiality of I/O metadata. We have worked towards this objective by extending the implementation of SGX support on Barreelfish, starting from work that was previously done at ETH Zürich. Unfortunately, we have still not yet reached the point of successfully running an enclave on Barreelfish, and more work remains to be done.

Acknowledgements

We would like to thank Tyler Hunt for providing us with his modified version of QEMU with SGX support. We would also like to thank Marc Gähwiler and Philipp Gamper from ETH Zürich for their work in implementing SGX support in Barreelfish, which this work builds upon.

References


