Secure Multi-threading in Keystone Enclaves

Stephan Kaminsky

May 17, 2021

Secure Multi-threading in Keystone Enclaves

by Stephan Kaminsky

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Committee Examination:

Committee:

[illegible]

Professor Dawn Song
Research Advisor

5/14/2021

(Date)

* * * * * * *

[illegible]

Professor Krste Asanovic
Second Reader

5/17/2021

(Date)

Secure Multi-threading in Keystone Enclaves

Stephan Kaminsky
University of California Berkeley
skaminsky115@berkeley.edu

ABSTRACT

There has been a growing popularity of running applications in Enclaves. Enclaves protect an application and its data against a un-trusted and possibly malicious operating system. Because of this, trusted execution environments have been created to combat this such as Intel's SGX and ARM TrustZone. As the enclave market has begun to mature, there has been a shift in research to integrate existing legacy applications in enclaves to protect them in the cloud [2]. Some of these enclave platforms such as SGX are limited on the size of their enclaves due to hardware limitations while other platforms like ARM TrustZone only have a single secure zone. Keystone is another framework which creates customizable TEEs based on the RISC-V architecture. It does not share the same listed limitations as SGX or TrustZone though currently does not support an enclave running on multiple harts. This limits enclaves which want to be secure and performant. This paper analyzes the different approaches of secure multi-threading in Intel's SGX and ARM TrustZone to design a model for secure multi-threading in Keystone Enclaves. The design is robust enough to allow for support of thread isolation inside an enclave which is useful in edge computing networks.

1 INTRODUCTION

Trusted execution environments (TEEs) have been growing in popularity to protect sensitive code and data. Projects such as Haven have been working on integrating legacy applications inside of SGX enclaves to protect them from computation in the cloud [2]. As code is pushed into enclaves to help protect its execution and data, multithreading can be difficult to implement thus performance can be hurt. This is a problem as many applications are now relying on multithreading to improve the performance.

Performance programming relies on multiple parts to ensure that the code is executed correctly. This requires that synchronization and scheduling do not attempt to exploit latent vulnerabilities in the code. Additionally, giving full control over scheduling to the operating system may allow it to manipulate the code into executing a different path than it normally would have. This is an issue since the enclave's purpose is to protect the data and general execution from the operating system.

In this paper, we show a modification to the Keystone framework to enable secure multithreading in RISC-V operating systems. We propose a method for the Security Monitor (SM) to delegate control to an enclaves supervisor for enclave scheduling to decrease the size of the SM in addition to allowing the enclaving to managing its own synchronization. This model also prevents a malicious operating system for scheduling enclave threads in its own ways as the enclave supervisor determines which enclave thread it wants to execute when CPU cycles are given to it. Additionally, this model allows for an enclave to not be limited to a static number of threads in its execution.

The approach outlined in this paper pushes a lot of decisions to an enclave for how it wants to manage its own CPU cycles. Intel's SGX has a static number of threads which must be defined as entry points for an enclave [11]. This model gives the scheduling control to the operating system which may then be able to exploit synchronization bugs in the enclave. Additionally, SGX has some of the synchronization primitives controlled inside the operating system which may make an enclave enter critical sections with multiple threads. Our model keeps all of the scheduling and synchronization decisions inside of the individual enclave to prevent the operating system from gaining too much control over the enclaves control flow. ARM's TrustZone does allow for a separate secure operating system to schedule trusted threads. The problem with this model is there are only two zones, a secure zone and an untrusted zone. The Keystone model which we propose will allow for each enclave to have its own trusted zone so that bugs in one thread will be unable to affect other enclave threads.

This paper is structured as follows. We first review the related work in how other TEEs handle scheduling in enclaves in addition to vulnerabilities to their designs. We then review the existing Keystone structure and build upon the design to enable multithreading to be possible in this framework. We next review the requirements we need based off of some vulnerabilities which were found in the other platforms. Next, we review how each of the components needs to be modified so that it will be able to support multithreading correctly. We then evaluate how this design follows all of the requirements we set out to complete. Finally, we review future ideas we have to add to this model.

2 RELATED WORK

As chips reach limits to how much faster they can become each year, programmers are forced to find other methods to increase performance. Multi-threading has become one popular and efficient method to improve performance. Because multi-threading can create big improvements in a programs performance, different related enclave platforms implemented their own take on multi-threading.

2.1 Intel SGX

Intel SGX allows developers to secure enclaves which are isolated contexts inside their applications [11]. This isolation protects the applications data from being viewed by other processes running on that same machine. To deploy an enclave, an application developer would use the SGX SDK.

Intel SGX and the SDK have support for multithreading and other synchronization mechanisms. Enclaves must have at least one entry point at which it may be entered. SGX supports multithreading by having multiple entry points which are allowed to be entered concurrently. Because of having a static number of entry points on initialization, an SGX enclave has a static number of threads which it may run.

SGX offers synchronization such as mutexes and condition variables. SGX developed a hybrid approach for managing these primitives. This means that lock variables are maintained inside the enclave while system calls are outside. This means that synchronization primitives may results in an enclave exit [11].

Additionally, the operating system has fine control over when an enclave runs. It can also send signals which will stop an enclave at a current point and return control to the operating system. It also may control which thread will be executing in an enclave.

These features give the operating system control over different aspects of the enclave. The operating system is able to start and stop SGX enclaves, it has full control before ecalls to SGX enclaves and during when and SGX enclave makes an OCall, and the operating system is able to interrupt and resume any SGX thread.

Given these abilities of the operating system, it may be able to manipulate the execution of an enclave. Since enclaves may now run with multiple threads, they may have some typical synchronization bugs such as atomicity violations, order violations, and data races. For example, an attacker would just have to find an exploitable synchronization bug, interrupt and schedule specific enclave threads, and then determine experimentally when to interrupt and schedule enclave threads [11]. This can be done quite precisely with page faults though is also able to be done by sending Linux signals to transfer control flow back to the operating system [11].

Since the operating system has full control over the enclaves running threads, it can also exploit the order in which threads are scheduled to manipulate a machine learning model. The operating system can poison asynchronous training models which are executing inside an enclave by pausing the execution of a thread while the learning rate is very high and only rerunning it when the global learning rate is much lower [8]. This causes a sharp change to which feature would be selected, causing the model to point to a possibly adversary controlled label.

2.2 ARM TrustZone

ARM's TrustZone is another model for creating secure enclaves. They achieve this by have two separate operating systems, a secure operating system and an untrusted operating system [7]. This model does fix some of the vulnerabilities which have been shown in SGX as the untrusted operating system does not have full control of which secure application is executing in the secure OS's environment. A problem with this model is a vulnerability in the secure operating system which is exploited by a malicious secure task may be able to get access to another secure tasks data. While this is not idea. ARM TrustZone was a step in the right direction to give TEEs while giving the ability to multithread.

3 KEYSTONE

Keystone is another model of creating TEEs in RISC-V. It currently does not support multiple harts from executing a single enclave at this time.

3.1 RISC-V Privileged Modes

RISC-V defines 4 different privilege levels: Machine Mode, Hypervisor Mode, Supervisor Mode, and User Mode (these are ordered from most privileged to least privileged) [10]. To create a system which is able to securely create TEEs, Keystone utilizes machine, supervisor, and user mode. We provide a description of all three privileges.

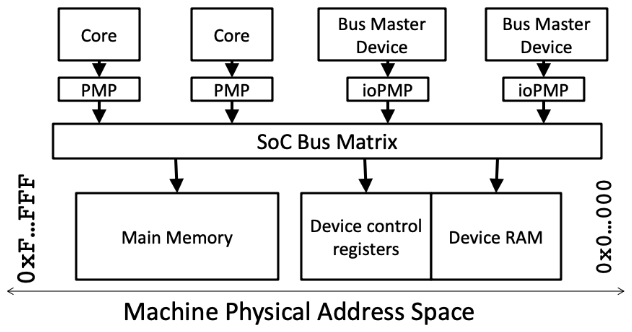

Figure 1: An illustration of PMP in the RISC-V memory hierarchy. Picture from Krste Asanovic, *UC Berkeley CS152*, presented 2019. [1]

3.1.1 Machine Mode. RISC-V Machine mode has access to all instructions. It also has access to special Control and Status Registers which control the system and give vital system information [10].

M-mode also controls the PMPs and ioPMPs to contain the active context inside a physical partition. Figure 1 depicts where in the memory hierarchy the the PMP validation lives. When the machine boots up, M-mode has access to the entire system. M-mode can restrict access for different privilege levels to have access to regions of memory. The PMP address encodes the address of a contiguous physical region and include configuration bits to specify the r-w-x permissions, and two addressing mode bits. It can even restrict M-mode access to regions which can only be reset with a machine reset [1]. This system is helpful as PMPs can be dynamically swapped to run different security contexts on a hart. The important thing to note is there are separate PMP contexts per hart meaning different harts can have different security contexts. Additionally, I/O devices have a set of ioPMPs which will restrict their DMA to specific regions similar to how the PMPs in each hart operate [1].

In Keystone, we have created a small and lightweight Security Monitor (SM) which keeps PMP entries in sync and manages all enclave operations.

3.1.2 Hyper-visor Mode. At the time of writing this paper, the Hyper-visor Mode specification has not been finalized yet [10]. Eventually when it has been finalized, Keystone will need the equivalent of the Security Monitor in the Hyper-visor as well especially since there will be a need to virtualize the PMP registers if there are multiple supervisors running on the system. Future work in determining the best model will be completed after the specification has been finalized.

3.1.3 Supervisor Mode. The supervisor is equivalent to the mode an OS executes in. It has privileged access to modify the page table, page-in and page-out, flush the TLB, set interrupts, and more. It is **not** able to accesses physical memory addresses which violate the CPU's PMP entries nor is it able to change the PMP entries. This means that the OS can no longer access an enclave's memory once the Security Monitor has created an enclave.

Keystone adopts a model of giving the OS full control over the resources given to an enclave on creation and also having execution time control over the enclave. The SM will set interrupts so that it can still allow the enclave to run for a specified amount of time. There are some security implications with this model such as some well known single-stepping attacks in other platforms which could also target this model [3]. Because of that, the SM has a minimum for what a timer for an enclave can be to make single stepping attacks much more difficult.

3.1.4 User Mode. A traditional application runs in this mode. It has the most limited access to the instruction set. In Keystone, enclaves have a user mode component which is the enclave application. Enclaves also have a supervisor component (runtime) which allows for the application to access the SBI interface that can trap into the SM.

3.2 Keystone Attack Model

The Keystone framework must trust the PMP specification as well as the PMP and RISC-V hardware implementations to be bug-free. There is another group currently working on formally verifying the Keystone Security Monitor (SM). A developer using Keystone can trust the SM only after verifying the SM measurement is correct, signed by trusted hardware, and has the expected version. The enclave application (eapp) only trusts the SM and runtime (the supervisor component of an enclave), the runtime trusts the SM, and the SM only trusts the hardware. [5]

Keystone was written so that it can operate under different threat models, each requiring different defense mechanisms. Keystone's goal is to protect the confidentiality and integrity of all the enclave code and data at all points during execution. They break down the four classes of attackers as [5]:

(1) **Physical Attacker:** They can intercept, modify, or replay signals which leave the chip. Assumed to not be able to modify components inside the chip package.
(2) **Software Attacker:** They can control host applications, the untrusted OS, and network communications. They can launch adversarial enclaves, arbitrarily modify any memory not protected by TEE, and add/drop/replay enclave messages.
(3) **Side-channel attacker:** They can get information by passively observing interactions between the trusted and untrusted components via some side channel.
(4) **Denial-of-service attacker:** They can take down the enclave or host OS. Note that Keystone allows for the OS to DoS an enclave as it can refuse services to user applications at any time.

3.3 Keystone Components

Figure 2 depicts the complex overview of they Keystone setup and privilege levels. Because of the number of privilege levels, Keystone is composed of multiple components to create its protection guarantees.

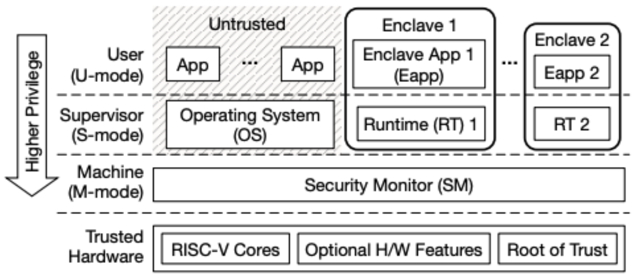

Figure 2: Keystone overview of the system setup and privilege levels. Includes components such as untrusted host processes, the untrusted OS, the security monitor, and multiple enclaves (each with their own runtime and eapp. Picture from *Keystone: An Open Framework for Architecting TEEs*, published 2019. [5]

3.3.1 Security Monitor. The security monitor is the primary component which enforces memory isolation. It utilizes the PMP registers so that different applications are not able to read/write/execute data from enclaves. Since PMP entries can be dynamically reconfigured during execution, Keystone is able to create a new region or release a region back to the OS. During the boot of the SM, Keystone configures the first (highest priority) PMP entry to cover its own memory regions and then disallows all access to its memory from all other modes. It then covers the last (lowest priority) entry to cover all of memory with all permissions so that the OS has default full permissions to all memory regions which are not otherwise covered by a PMP entry [5].

Since each core has its own complete set of PMP entries, it is the job of the SM to also keep other cores PMP entries up to date. To achieve this, cores are able to send inter-processor interrupts (IPIs) to each other during execution to immediately stop execution (when not in M-mode) to update its PMP entries. This ensures no core will have stale abilities to access newly restricted sections of memory.

The SM is also responsible for managing the enclave life-cycle. When the OS wants to create an enclave, it will load the enclave binary into physical memory, sets up the enclaves page tables and allocates the physical memory. The OS then tells the SM to create the enclave in a specific space. The SM will ensure the enclave binaries are correct, then it will walk the OS-provided page table to ensure there are no invalid mappings and only unique virtual-to-physical mappings. The SM will finally has the page contents along with the virtual addresses and configuration data which is used to verify the enclave has been correctly loaded [5].

The SM is also in charge of executing an enclave. This means it needs to set the PMP entries and then transfer control to the enclave entry point. Additionally, the SM is responsible for destroying enclaves. It is important that the tear-down of the SM clears the enclave memory region before returning the memory to the OS as to not leak information. Furthermore, it will free all enclave resources, PMP entries, and enclave metadata. The SM also supports the easy integration of optional plugins with different security protections which are not in the base Keystone.

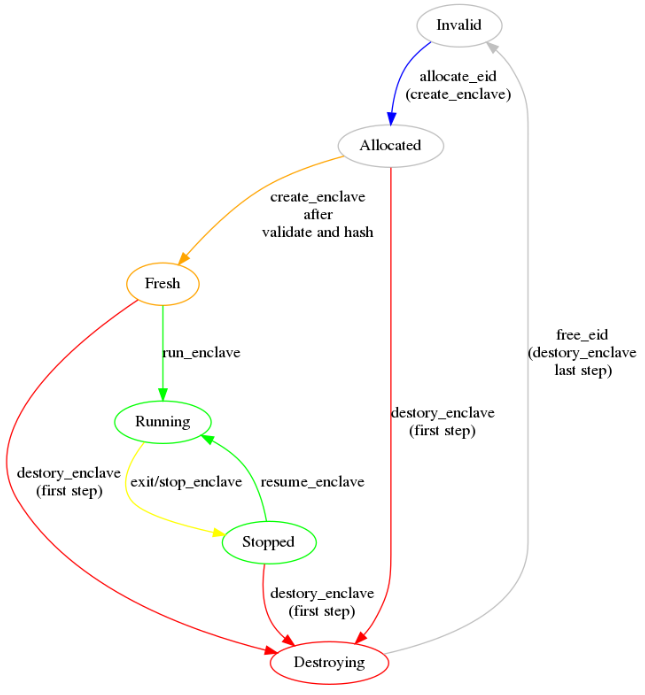

Figure 3: This is the current enclave life cycle inside the SM. Picture from Stephan Kaminsky *Keystone*, published 2019. [4]

3.3.2 Keystone Runtime. The Keystone RT is the supervisor which runs inside of an enclave. It is a lightweight supervisor which provides a basic set of call interfaces similar to how the OS provides a limited set of functions to an application. The RT is responsible for performing edge calls on an enclave on behalf of the enclave application's needs to perform read/writes outside the enclave (i.e. write to the host's stdout stream). Additionally, it proxies system calls to the host OS. Keystone is able to utilize existing defenses to prevent Iago attacks via this call interface. The benefit of this RT is it allows for developers to port over their legacy apps without needing to build their own supervisor to support their existing applications [5].

3.3.3 Keystone Driver. The Keystone Driver is a Linux driver which manages the enclaves which the OS has launched. It understands the system call interface to create/run/destroy enclaves. The untrusted application will interact with it to launch an enclave, of course, this is assuming the OS permits it. The driver will ensure that any execution time, which is currently given to the untrusted application, will be given to the eapp which the application is executing. Additionally, the driver ensures that there is an easy interface so that an eapp can make system calls which need to be serviced either by the OS or by the untrusted application [5].

3.4 Enclave Life Cycle

Figure 3 depicts the life cycle of an enclave inside of the SM. The SM has a fixed number of enclaves it can support which is determined upon the SM boot. Because of this limited size, there exists a metadata structure created for each enclave. We will refer to these structures as eids (though eid does refer to the enclave ID). We will now explore the different stages of an enclave [4].

3.4.1 Invalid. In this state, an enclave structure is un-allocated but available to be allocated. This state will change to allocated once the create enclave functions allocates the eid.

3.4.2 Allocated. Once an enclave has allocated the eid, the enclave structure is said to be allocated. This prevents multiple enclaves to be initialized to fill the same enclave structure. Once allocated, the enclave can go to two different states. Either it can go to the 'Destroying' state which may be caused for various reasons such as a failure to validate and hash the enclave. If the enclave finishes its validation and hashing, it now becomes 'Fresh'.

3.4.3 Fresh. An enclave enters the fresh state once it has been hashed and validated from the allocated state. The fresh state indicates that an enclave is ready to be executed but has not been executed so it will need to be further set up to be able to execute correctly. From the 'Fresh' state, an enclave can either go to the "Destroying" state if destroy enclave is called on it. Otherwise, if run enclave is called, it will set up the state of the enclave and then transfer the enclave state into the 'Running' state.

3.4.4 Running. The running state indicates that the enclave is being currently executed on at least one hart. It may be entered either by the run or resume call. Once all threads have been suspended, it will transfer to the 'Stopped' state. Do note that the 'Running' state is one of the few states which cannot directly go to the destroying state. This is because there could be disastrous issues if one hart starts destroying an enclave while the other is executing it. Because of this, the enclave must be stopped before it is able to be destroyed.

3.4.5 Stopped. The stopped state can only be entered via the exit or stop enclave functions. It means the enclave has executed at some time in the future but is not currently executing on any hart of the CPU. Once stopped, the enclave can then proceed to the 'Running' state if resume is called on it or if it can destroy itself.

3.4.6 Destroying. An enclave in any state other than 'Invalid' (because it doesn't make sense to destroy something which is already destroyed) and 'Running' (you should not be able to destroy an enclave which is currently executing on another hart) can go to the Destroying state. An enclave will be brought into the 'destroying' state immediately at the start of the destroy enclave function. This is meant to prevent an enclave which is being destroyed to suddenly start executing. In this step, the SM will clear all of the enclave memory, free any enclave resources, and finally clear the enclave eid so that it may be reused in the future. Once destroy enclave completes, the eid goes to the 'Invalid' state.

3.5 Keystone Current Limitations

Keystone offers a great initial framework for creating, validating, and securing enclaves. Even with all of that, it is currently is lacking the ability to allow multiple harts to enter a single enclave. Additionally, the current design of Keystone requires that the SM keeps track of every single thread for every enclave. This can significantly bloat our trusted computing base which can make it more difficult to verify in addition to requiring a larger amount of memory.

Another issue with the design of Keystone is the lack of ability for the enclave supervisor to set pending interrupts for itself. This design severely limits any ability for the enclave supervisor to do any scheduling of its own.

4 DESIGN

4.1 Requirements

AsynchShock[11] and Game of Threads[8] have shown some of the pitfalls of giving too much control to the operating system. Because of that, we must look at what we will require our system to be able to protect against.

❶ **The operating system should not be able to control the page tables of an enclave when an enclave is executing.** This will prevent the operating system from learning about where inside an enclave, the enclave is at in its execution.

❷ **The enclave will have a constant and well defined entry points which do not change from enclave to enclave or during the execution of the enclave.** This requirement forces the operating system to not be able to pick and chose what part of the enclaves code it will be continuing to execute.

❸ **The attacker will not be able to control which thread is running or pause out specific threads.** This is so the operating system will be unable to force an enclave to preserve values which may have already been confirmed as fine to use but will cause unexpected side effects if ran later in the code.

❹ **All synchronization must happen inside the enclave.** This will make it so the operating system is not able to maliciously modify primitives which will cause multithreading invariants the enclave is relying on to be violated.

❺ **The enclave should not be limited to a certain number of static threads nor should the operating system be able to determine how many threads are executing if that is unwanted by the enclave.** This requirement gives the enclaves flexibility in their runtime and gives programmers more flexibility in how they design their EAPP.

4.2 Security Monitor

The goal for the security monitor was to abstract out as much about multithreading as possible. The goal is to reduce the trusted computing base while still enforcing all security guarantees which where previously given. To do so, the way how we view the enclave lifecycle had to change to support the case of having multiple harts running at the same time.

4.2.1 Enclave Metadata. The enclave metadata block contains a lot of useful information about an enclave. This includes the current state it is in, saved registers which can be restored when it is running, and other metadata. This model works great if you only have one thread executing at a time in an enclave. This does take up a decent amount of space as you need to save all of the registers when halting an enclave. Additionally, this model increases the trusted computing base as the Security Monitor now has to handle the full context switch, since it now has to handle saving or restoring the enclaves registers before it enters the enclave. Given the current model, an approach to enabling multi-hart execution would require us to either ❶ statically allocate N harts of save space per enclave or ❷ dynamically allocate space in the Security monitor, such as a linked list where a node holds a pointer to the next saved thread data in addition to the save data for a thread.

Approach ❶ still allows us to give an enclave dynamic thread allocation which is a goal of this project. The issue is the number of simultaneous threads which it can have has a static limit which the Security Monitor will have to be fully aware of and decide on when the machine boots. This creates limitations for an enclave which may want to execute many threads.

Approach ❷ removes the static allocation limitation of ❶ as we would be able to allow an enclave to have virtually unlimited thread executing in addition to not requiring all the static data to be allocated on boot.[1]

These approaches have limitations for an enclave. Since the Security Monitor has to manage the threading, it needs to understand more about an enclave. The Security Monitor now must decide which thread should be executed next or it can give that control to the enclave such as the model of SGX. This is not advised as shown by the attacking thread scheduling [8][11]. These approaches also increase the trusted computing base which can lead to an increase in size of the Security Monitor in addition to more possible locations to have bugs.

While there are drawbacks to these approaches, they does have some benefits which should be noted. They would keep the enclave sizes smaller as the enclave does not need to support its own scheduler. Additionally, it does not have the extra overhead required to save a thread on an interrupt as we would have to re-enter the enclave when interrupted to save the thread before we return back to the host.

Even with these benefits, its limitations make these approaches less than ideal. Another model is to push all of the thread allocation and scheduling to the enclave. This approach would require the Security Monitor to save the address of the entry point for the enclave which will be (re)entered whenever the enclave is scheduled to execute or continue executing. The Security Monitor also keeps track of a second entry point which would be executed to handle interrupts.[2]

4.2.2 Enclave Life Cycle. The enclave lifecycle is designed to ensure that only valid operations happen on an enclave. There would be a vulnerability if you could destroy an enclave at the same time as it is running in another core. This could cause it to change its execution behavior externally, something not specifically done by the enclave itself, which breaks one of the invariants of our enclaves. Thus, special care has to be done to ensure that multiple harts running does not leak data.

Not all stages needed to be changed to have a secure enclave lifecycle. So long as the stage is atomically checked and changed,

[1]This is virtually unlimited as the Security Monitor has a fixed allocated size on boot. What this allows is for a smaller amount of data to be allocated in the Enclave metadata list as it would point to dynamically allocated nodes of a linked list per enclave containing the thread information.

[2]This handles both cases where the enclave scheduled an interrupt which fired and the case where the OS scheduled interrupt was fired. The enclave supervisor is given free reign on deciding how it wants to handle those cases so long as it returns back to the Security Monitor before the watchdog timer interrupt occurs if it was an OS interrupt.

there should not be issues of multiple harts performing some operation on an enclave which should only have one hart executing on it. For example, an enclave enters the Allocated stage after the SM attempts to allocate an enclave ID by searching the enclave metadata list for an enclave in the Invalid state. Once it finds one, it changes to the allocated state. If multiple harts are running create enclave at the same time, the spin-lock surrounding the check will prevent multiple from allocating the same enclave. Additionally the state change to Allocated happens before the spin-lock is released so that two harts are unable to grab the same enclave metadata block.

The next state which can be affected by multiple harts attempting to start an enclave is the transition from the Fresh state to the Running state. Because the run_enclave call performs some important enclave setup stuff not performed by create_enclave, it is important that it is executed only once. To accomplish this, when this function is called, it grabs a lock and the searches for the enclave (because it is passed in an enclave ID). Once it finds the enclave ID, it will check to see if it is in the Fresh stage. If it is in the fresh stage, it will change the state to Running and then increment a thread counter. This is important as we now need to keep track of how many harts are currently executing the same enclave. Finally, we release the lock once we have done the verification to ensure that only one hart can perform the check and state change, if needed, at a time. If we failed to meet those checks, due to two different harts executing run_enclave on the same enclave, we will return up with an error saying the enclave was not Fresh. Thus this system will ensure that we only have a single hart ever execute run_enclave for the current enclaves lifecycle (until it eventually becomes Invalid).

The next two states are Running and Stopped. They needed some changes to support multiple harts executing them. Since it would be bad if we attempted to destroy a running enclave, when a hart calls stop_enclave, it will decrement a thread counter. Only if the thread counter reaches 0 will the enclave transition to the Stopped state. Otherwise it will stay in the running state as at least one other hart is executing within the enclave.

We finally have the Destroying state. This state can only be entered when an enclave is in the Stopped state as that means no hart is actively executing that enclave. This state must also only have one thread executing it at a time otherwise we may leak data or cause undefined execution behavior. To solve this issue, we acquire the enclave lock, check that the enclave is in the Stopped state, and finally change the state to Destroying before we release that lock. Since the Destroying state can only be entered via that mechanism, we do not need to worry about another hart entering that state as it may only enter that if the enclave is in the Stopped state. Once it is has completed destroying the enclave, it will deallocate the enclave metadata block by changing the state to Invalid. Thus we have concluded the lifecycle of an enclave as it is back to a state which can be allocated.

4.2.3 Interrupts. Giving the enclave the ability to set interrupts is important to allow for the enclave's supervisor to manage its own threads. There are some limitations and careful considerations which must be made so that the enclave is unable to take control of a hart. To do this, we multiplex our interrupts so that the enclave can set its own interrupt which will be delivered to itself. Additionally, we also need to handle the original interrupt which the OS had set for the enclave.

The idea is for each enclave to have an interrupt entry point which the SM will enter if an interrupt has been made. There will be two types of timer interrupts which it will be handling. The first type of timer interrupt is one set by the enclave itself to manage its own scheduling. This interrupts is essentially delegated to the enclave so that it can process the interrupt.

The other type of interrupt would be one set by the OS to request the hart back from the enclave. In this case, the SM will set a new, short timer interrupt known as a watchdog timer interrupt and enter the SM's interrupt handler. The SM alerts the enclaves supervisor that this is a switch back interrupt. This will then allow for the SM to run the small bit of code necessary for copying and saving the threads data in addition to returning back to the SM. Once it returns back to the SM, the SM will finish resetting the state and return back to the host. If the enclave does not return back to the SM before the watchdog timer interrupt occurs, the enclave will be viewed as being delinquent and abruptly destroyed before returning to the OS.

By default, the only interrupt which is ever set is by the SM given what the host requests.[3] Given just this interrupt, when it is triggered in this model, it will enter the enclave's interrupt handler and tell it to save its state and return so it can return control to the OS. The model proposed here allows for the enclaves supervisor to set its own interrupt. If an enclave sets an interrupts, the SM will have to verify that the time which the interrupt will happen is before when the OS's interrupt will occur.[4] If the time which the enclave supervisor requested is before the original interrupt, the OS interrupt time will be saved so that the next interrupt can be the time given by the supervisor. When the SM has to handle an interrupt, it will check to see if there exists a saved OS time for an interrupt, if it see that one exists, it had determined the interrupt is set by the supervisor of the enclave and will forward the interrupt to the enclave to handle. If the OS's saved timer interrupt is not there, then it has determined it is the OS who set the interrupt and the enclave needs to be saved and then control should be returned to the host. If the supervisor requests an interrupt which is after the OS interrupt, then the interrupt is ignored (and the enclave is notified). This is done so that the SM does not need to store state about an enclave/thread which is running in the enclave.

4.3 Enclave Supervisor

The enclave supervisors also needed to be adapted to support multithreading.

4.3.1 Entry Points. The enclave supervisor will now have two different entry points. The first entry point will be where the enclave will set up and run itself. When this entry point is called, the enclave supervisor will execute the scheduler which will pull a thread off of the scheduling queue to then execute. It will also, possibly, schedule

[3]Note that Keystone requires that an enclave is allowed to run for a minimum time to prevent single stepping an enclave.

[4]In addition to a small amount of time to prevent senseless repeated interrupts by an enclave right before the existing OS interrupt. This small amount of time is to allow the scheduler to successfully schedule another thread before the interrupt would occur.

an interrupt so that it can switch its own threads if the OS gives the enclave a lot of time to execute.

The second entry point will handle all of the interrupts which the enclave must handle. Mainly, it will have to deal with two different interrupts: the enclave received an interrupt set by ❶ itself or ❷ the OS. If the enclave set its own interrupt and it was triggered, then the supervisor will simply save the current thread data to it thread control block and then execute the scheduler to pick the next thread to activate. If the interrupt was set by the OS, the enclave supervisor will then save the current executing thread's data to the thread control block and return back to the SM so that it can return to the OS.

4.3.2 Scheduler. Since the scheduler is completely abstracted away from the SM, its implementation can be to however the EAPP author wants. An example scheduler is a simple round-robin priority scheduler which supports multiple queues so that we can set threads on the blocked or available to execute queue. This two queue approach helps ensure that a blocked thread does not waste CPU cycles. The priority aspect of this scheduler helps ensure that synchronization does not cause deadlock or other slowdowns for happening as the priority of a thread may be elevated if it holds a lock which another, higher priority, thread may be requesting. Blocked threads may be caused by the attempt to acquire a lock which is already held or an OCALL which needs to be made. The EAPP author has the flexibility to design it as they need.

When the scheduler is entered, it will schedule a thread and set a timer interrupt with the SM. The SM will return if that timer interrupt is successfully saved or if it failed. If it failed, this allows the scheduler to decide what it wants to do next as the OS has a sooner interrupt than the scheduler. The default behavior is to ignore that it failed and act as if it is a success and run the thread for the smaller time-slice.

An optimization can be made to ensure senseless enclave supervisor interrupts are not given if there is only one thread running. To do this, the enclave's scheduler will not schedule an interrupt on the hart if it detects that only one thread exists (both in the ready and blocked queues). If there is only one thread, than an interrupt will not assist it. Once the enclave thread makes a call to create a new thread, the scheduler will then set an interrupt for the hart so that it can switch between threads. This approach will always work in giving more optimal timings as the currently executing thread is the only method to add additional threads to execute in the current model. Because of this, we can prevent an enclave from wasting time switch between a single thread.

4.3.3 Interrupts. The enclave supervisor now has to be fully aware of certain types of interrupts. Specifically, we may have an interrupt which was triggered by the scheduler and an interrupt which is triggered by the OS. When the enclave is entered via the interrupt handler entry point, it will check to see if the interrupt was an OS interrupt or an enclave interrupt. If it was an enclave interrupt it will simply execute the scheduler which will re-queue the existing thread and execute another one (and set the respective interrupt). If the interrupt is determined to be from the OS, then it will just execute the code to save the current thread before returning to the operating system.

4.3.4 Synchronization. To assist with shared resources which threads may have, synchronization has been added to assist with this. To be specific, locks and semaphores which are thread aware exists to assist performant operation of the eapp. When a thread in the enclave attempts to acquire a thread, it will call the enclave's supervisor to assist it in getting the lock. While this does require some extra context switching, it gives a benefit of making the scheduler aware that a thread is now blocked if the requested lock is held. This allows the scheduler to put the thread on a blocked list so it does not waste CPU cycles.

5 EVALUATION

Given the design which has been outlined, we will now go through and show how we have implemented each of the requirements we set out to have.

❶ The operating system should not be able to control the page tables of an enclave when an enclave is executing. This is an important requirement which could give a malicious operating system vital information about what the enclave is currently doing. The Keystone framework already solves this problem by isolating an enclave with its own supervisor which will manage the page table for it. The operating system will initially set up the page table and give the free pages though all of that is validated when the enclave is initialized. Once the enclave is initialized, the operating system is no longer able to modify it which means it will not be able to gain access or change control flow of an executing enclave.

❷ The enclave will have a constant and well defined entry points which do not change from enclave to enclave or during the execution of the enclave. This requirement forces uniformity in addition to decreasing the SM's size as it does not need to do a lot of extra computation or use extra space depending on what the enclave wants to do. In this model, the SM is actually reduced in size and complexity as it no longer needs to keep track of every register for every enclave. It only needs to know the specific entry points and enter them when it wants to give CPU time to an enclave. This also gives the enclave much more free will in its ability to schedule its own threads without having to do massive communication with the SM. The SM, which is our TCB, now can be much smaller which means it is easier to validate in addition to having fewer moving parts.

❸ The attacker will not be able to control which thread is running or pause out specific threads. This goal is achieved by forcing all of the scheduling to happen inside of an enclave. The enclave supervisor has its own scheduler which will allow for its own, non-manipulated choice in what thread should execute. The operating system is only able to give CPU time to an enclave as a whole instead of specifying that the CPU time is for a specific thread. In addition, this model abstracts the threading from the SM as well as the SM no longer needs to worry about keeping track of which registers to restore when allowing an enclave to continue to execute.

❹ All synchronization must happen inside the enclave. Since an enclave has its own supervisor which has its own scheduler, all synchronization primitives are held inside the enclave. Since the enclaves supervisor is in charge of scheduling, it can safely and quickly manage critical areas of the code as to not allow for the operating system to manipulate the enclave into entering critical sections with multiple threads.

❺ **The enclave should not be limited to a certain number of static threads nor should the operating system be able to determine how many threads are executing if that is unwanted by the enclave.** In this model, the SM only needs to know about the entry points of an enclave. It does not need to understand how many threads or what threads should be executing inside an enclave. The only thing the SM keeps track of is how many CPU harts are executing a single enclave as to ensure that the enclave does not enter a state which it should not be in. The enclave supervisor is the one managing its own threads which means it may have as many threads as its own scheduler allows. In the model proposed in this paper, there are no limits to the number of threads which may be executing. In addition, if the enclave wants to hide the number of threads it has from the operating system, if the operating system was to give more harts than the enclave had threads, the enclave could hold the extra harts for the normal amount of time as to not leak to the operating system information such as that.

6 FUTURE WORK

For future work, we plan on adding configurable full isolation via virtual memory to enclave threads. This is a relatively easy feature to add as the enclave is already running in supervisor and user mode. The supervisor would be the one to manage the virtual memory so that individual threads are properly protected from each-other. This has useful application in edge computing[9][6] as one can isolate multiple lambda functions from each-other for more performant operations.

7 CONCLUSION

This paper presents a model which can be used to allow for secure multithreading in Keystone enclaves with minimal overhead. The approach used helps decrease the trusted computing base as some of the work which had to be done in the SM has been moved to the enclave. This helps shrink our trusted computing base in the SM which helps make verification easier. Our analysis shows that this new model allows for enclave to run multiple threads which an operating system is not able to maliciously control and manipulate. This allows for Keystone to support richer EAPPS with no changes to the SM as the scheduling has been fully delegated to the enclave's supervisor while still maintaining our security requirements.

REFERENCES

[1] Krste Asanovic. 2019. Lecture 9 – Virtual Memory. https://inst.eecs.berkeley.edu/ cs152/sp19/lectures/L09-VirtualMemory.pdf.

[2] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 267–283. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[3] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[4] Stephan Kaminsky. 2019. Keystone SM Enclave Life-cycle.

[5] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. https://arxiv.org/pdf/1907.10119.pdf.

[6] Nitesh Mor, Richard Pratt, Eric Allman, Kenneth Lutz, and John Kubiatowicz. 2019. Global Data Plane: A Federated Vision for Secure Data in Edge Computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1652–1663. https://doi.org/10.1109/ICDCS.2019.00164

[7] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (Jan. 2019), 36 pages. https://doi.org/10.1145/3291047

[8] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 35–52. https://doi.org/10.1145/3373376.3378462

[9] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[10] Andrew Waterman and Krste Asanovi ́c. 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture* (document version 1.12-draft ed.). RISC-V Foundation.

[11] Nico Weichbrodt, A. Kurmus, Peter R. Pietzuch, and R. Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*.