SecSched: Flexible Scheduling in Secure Processors

Omais Shafi
Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi,110016
omais.shafi@cse.iitd.ac.in

Janibul Bashir
Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi,110016
janibbashir@cse.iitd.ac.in

ABSTRACT

Trusted execution environments (TEEs) are an integral part of modern processors because security has become a very important concern. However, many such environments are bedeviled by the high cost of context switches, particularly when there is a switch from secure mode to non-secure mode owing primarily to cache pollution and TLB-flushing overheads. State-of-the-art implementations create a secure shared memory channel between a thread running in secure mode and a thread running in non-secure mode, which invokes system calls on its behalf. We argue that this is inefficient, and it is possible to reduce the overheads significantly by efficiently storing the context of secure threads and intelligent scheduling. In this paper, we propose a new scheduling algorithm *SecSched* that uses Cuckoo filters to capture the context of a thread. We schedule threads with similar contexts on the same core to leverage the effects of the locality. Our algorithm requires minimal hardware enhancements that are limited to maintaining a Cuckoo filter per core and a thread with the addition of few performance counters per thread to keep track of the miss counts. We show that with these minimal changes we can increase the performance of a suite of OS-intensive workloads by 27.6% with a minimal area overhead (around 0.04%).

CCS CONCEPTS

• **Security and privacy → Security in hardware**; **Systems security**; • **Software and its engineering → Scheduling**.

KEYWORDS

SGX, Hardware security, TLB flushing, Scheduling

ACM Reference Format:
Omais Shafi and Janibul Bashir. 2020. SecSched: Flexible Scheduling in Secure Processors. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414631

1 INTRODUCTION

Security is an integral part of modern processors. Major vendors such as Intel and ARM have added security enhancing modules to their processors. Intel introduced the SGX secure enclave [8] in 2015, and similarly ARM introduced TrustZone [4, 25, 35] as a security module in their ARM processors. Such environments are known as *trusted execution environments* (TEEs). They keep the code and data of secure programs in a dedicated region in memory in an encrypted format. It is not possible for untrusted programs to read or modify the state of encrypted programs. Along with TEEs another class of solutions known as trusted platform modules (TPMs) [5, 21, 22] are also becoming very popular, particularly in embedded environments. A TPM is a separate hardware module that contains secure keys, can perform cryptographic operations, and can authenticate/attest messages. Processors such as IBM Power8 [27] and IBM PowerEn [18] contain such dedicated modules. In this paper, we shall look at **improving the performance of TEEs without compromising on security**.

Specifically, we shall consider TEEs such as the Intel SGX platform. After SGX was introduced in 2015, we have seen some major efforts in the architecture community to improve its performance [24, 31, 34]. Researchers have identified two primary sources of the performance loss in Intel SGX: the mandatory encryption and decryption performed by the memory encryption engine for off-chip traffic, and the overheads of entering and exiting secure enclaves (encrypted regions of main memory). The solutions include faster cryptographic algorithms [19, 31], improved scheduling mechanisms [24, 34], efficient integrity verification [31], and dedicated methods to protect the TLB and page tables [24].

Our aim in this paper is to outperform the state of the art by targeting the time overhead due to the migration between secure and non-secure mode. We quantify the overheads of the Intel SGX by analyzing the performance of SGX on few microbenchmarks and create a performance model that captures the factors responsible for performance degradation in Intel SGX. Prior work *Hotcalls* [34] pins the threads to cores whenever a system call is encountered inside the enclave, thus reducing the overheads of enclave exits. However, it introduces the core idleness in the system. *SGXKernel* [33] uses an in-enclave multithreading by migrating application threads to these idle cores. Migrating random threads to the cores may result in significant pollution in the caches and TLBs. We intend to mitigate the overheads of both the schemes by proposing a new scheduling algorithm, called *SecSched* that migrates the threads to the idle cores by obtaining the working set similarity using Cuckoo filters [11]. Additionally, we keep a thread based count of the misses in the lower level cache and use this information to further schedule our threads efficiently. *SecSched* uses two sub-algorithms- Fast Matching and Slow Matching. The Fast Matching algorithm is for code like interrupt handlers that finds if the few recently accessed pages are part of a core's Cuckoo filter. The Slow Matching algorithm is for normal threads, that reduces the core idleness by using the Cuckoo filter and miss count information to map threads to

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414631

cores. Our scheduling algorithm improves the performance over the competing works by 27.6-54.1%. Let us briefly summarize our key contributions:

❶ We created a performance model and derived the major factors responsible for performance degradation in Intel SGX.

❷ We propose a scheduling mechanism that does the working set similarity (using the Cuckoo filters) based thread scheduling and the smart work stealing, thereby reducing the core idleness to nearly zero.

❸ We intelligently incorporated the lower level cache miss details into our scheme to further enhance the performance.

The rest of the treatise is as follows. Section 2 provides a brief background, characterization of the micro-benchmarks and OS benchmarks on a real SGX machine and a simulator respectively is presented in Section 3, Section 4 describes our design, Section 5 evaluates our design. We discuss the related work in Section 6 and finally conclude in Section 7.

2 BACKGROUND

Most modern secure processors run critical code inside trusted execution environments (TEEs). TEEs provide *confidentiality* and *integrity* of both the code and data. There are two popular environments in this space: Intel SGX [8] and ARM Trustzone [4, 25, 35]. We focus on Intel SGX because it has more features making it more suitable for server processors [12, 36]. We first study the background of Intel SGX with the associated bottlenecks and then we discuss the *Cuckoo filter* data structure that is an integral part of our design.

2.1 Intel Software Guard Extensions (SGX)

Software Guard Extensions (SGX) are a set of CPU instructions in Intel x86 machines that enable the user code to execute securely in trusted memory regions called *enclaves*. An enclave is a sandbox in which programs execute securely regardless of the operating system (OS) or the hypervisor. All the data destined to the enclave from the CPU is encrypted using the encryption engine inside the memory controller called the *Memory Encryption Engine(MEE)*(see Figure 1) [8, 13].

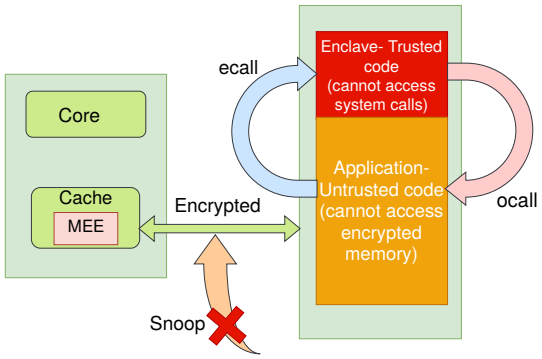


Figure 1: Communication between the processor and the enclave

2.1.1 Process of Creating an Enclave. At boot time the BIOS reserves a part of the physical memory, which is only visible to the SGX engine of the processor. This is known as *Processor Reserved Memory* (PRM). Whenever any user process wishes to create a secure enclave – a part of the PRM (limited to 128 MB) is used to create this enclave.

The enclave is divided into three parts: *EPC (Enclave Page Cache)*, *EPCM (Enclave Page Cache Map)*, and the *Meta-data store*. The EPC stores all the code and data pages of the secure application. The EPCM is used to store the mapping between the virtual address and the physical address. It is a page table for the secure application. There is a need to store this page table in the secure memory because if it is in the control of the OS, then it may maliciously modify the mapping thus compromising the security of the application. Finally, the metadata store keeps information for ensuring confidentiality and integrity.

2.2 Life Cycle of an Enclave

2.2.1 Enclave Creation. If an application wishes to create a secure enclave, it needs to follow these steps. First, it calls the ❶ *ECREATE* instruction, which initializes the SGX Enclave Control Structure (SECS) in the EPC, then it calls the ❷ *EADD* instruction to set the size of the enclave, and add relevant code and data pages to the enclave.

2.2.2 Entering and Exiting an Enclave. Once the enclave initialization is done, the user-mode application can execute the code in the enclave by invoking the *EENTER* instruction that switches the CPU to the secure mode. Likewise, the *EEXIT* instruction is invoked by an application to exit the enclave.

Note that the application has secure and non-secure parts. The transition between them happens with the help of the *EENTER* and *EEXIT* instructions. In the literature [8] the procedure for entering an enclave is known as an *ecall*, and the procedure for leaving an enclave is known as an *ocall* (Figure 1). We shall henceforth use the terms *ecall* and *ocall* only.

SGX sadly has a major limitation. It does not allow the application to make system calls in secure mode. This means that there will be frequent *ecalls* and *ocalls* in OS intensive applications. These procedures have significant performance overheads. *The main focus of this paper is to reduce such overheads.*

2.3 Performance Overheads in Intel SGX

The major factor for the performance overhead in Intel SGX is the context switch from the secure region(enclave) to the untrusted region of the main memory. This involves the overheads of making and returning from a function call, bookkeeping by the processor to enter the secure mode, and TLB flushing while leaving the enclave. It is important to keep in mind that all secure pages have a copy of their address translations in the EPCM, otherwise the OS can maliciously alter the page tables.

In Intel SGX, we need to flush the TLB every time there is a context switch [8, 24, 34]. This is because Intel does not store bits in the TLB to indicate that a given page is supposed to be accessed by an secure or a non-secure region. Hence, whenever there is a secure/non-secure mode switch we need to flush the TLBs. Along with TLB flushing, there is also some degree of cache pollution

introduced by running both the kernel and the user code on the same core.

2.3.1 Methods to Mitigate these Overheads. Given that a mode switch is so expensive, researchers have proposed various mechanisms to reduce the overhead of invoking system calls in the secure mode. All of these approaches follow the same approach, which is to create a proxy thread outside the secure mode that executes system calls on its behalf. The secure thread communicates with the proxy thread via a shared memory mechanism, and uses it to pass both arguments as well as return values.

Shortcoming of prior work: For every secure thread there is a dedicated non-secure thread that executes system calls on its behalf. Moreover, to avoid costly TLB flushes secure threads are typically pinned to cores (increasing core idleness) or secure threads are migrated across the cores. Additionally, the overhead of communicating data with the non-secure proxy threads is significant. The aim of this paper is to propose a mechanism to mitigate such overheads.

2.4 Cuckoo Filter

A Cuckoo filter [11] is a data structure that has $O(1)$ insertion and deletion time as opposed to the commonly used Bloom filter [14, 16, 29] that does not efficiently support deletion. It consists of an array of buckets where each item has two candidate buckets determined by the two hash functions. Each entry of the array(say B) contains the fingerprint $\mathcal{F}(e)$ of the item e. To insert an entry (e), the location $i1$ is accessed by computing $\mathcal{H}(e)$. If it is empty, $\mathcal{F}(e)$ is inserted in $B[i1]$. otherwise, location $i2 = i1 \oplus \mathcal{H}(\mathcal{F}(e))$ is accessed. If $B[i2]$ is empty, $\mathcal{F}(e)$ is inserted. Cuckoo filters have a beautiful property, where given a location, we can easily find its alternative location in a commutative fashion:

$$i1 \oplus \mathcal{H}(\mathcal{F}(e)) = i2$$
$$i2 \oplus \mathcal{H}(\mathcal{F}(e)) = i1$$

If $B[i2]$ is not empty, we displace the entry, $\mathcal{F}(e')$ in $B[i2]$ and insert $\mathcal{F}(e)$ in its place. The displaced entry must be having an alternative location given by $i3 = i2 \oplus \mathcal{H}(\mathcal{F}(e'))$. We try to insert the displaced entry in $B[i3]$. This procedure is continued till we either find an empty entry, or reach a threshold in terms of missed attempts.

3 ANALYSIS OF SGX

3.1 Overview

The main aim of this section is to ❶ analyze the main agents responsible for performance degradation in Intel SGX, and ❷ derive various design insights in order to decrease the effect of such agents. We divide this section into two parts. In the first part, we run some micro-benchmarks on the actual hardware in order to find the latencies involved with the various SGX specific functions. We analyze the effect of different parameters on the latencies associated with such operations. In the second part, we incorporate the details derived from the real hardware results into our simulator and run some standard benchmarks. Based on the simulation results, we derived some insights that can be used to design a system in order to decrease the latencies associated with various SGX operations.

Note that we are proposing a scheme that involves some **hardware modifications** and as a result a simulator is necessary to evaluate the efficacy of our scheme.

3.2 Hardware execution

3.2.1 Setup. For actual hardware execution, we are using the Intel Core i7-6700 machine loaded with Ubuntu Linux 16.04 (see Table 1 for hardware parameters). We ran various micro-benchmarks on the system and used a read time stamp counter instruction (*rdtscp*) to note down the execution time in cycles. The micro-benchmarks (listed in Table 2) are designed so that we can determine the actual latencies associated with SGX specific functions such as *ecalls* and *ocalls*, system calls and function calls executed (while in secure mode), and data transfers in and out of the secure enclave (1KB data transfers).

Parameter	Value	Parameter	Value
Cores	8	Frequency	3.40 GHz
Glibc version	2.23	SGX SDK	Version 2.5
Private L1 i-cache/d-cache	32KB	L2 cache	256KB
L3 cache	8MB	Enclave size	128MB

Table 1: Hardware parameters

Microbenchmark	Description
Empty-ecall	Enter the enclave with no parameters and immediately return
Empty-ocall	Move from secure to non-secure mode
System-call	Exit the enclave, make a system call, and re-enter it
Function-call	Invoke a function in the secure region (called from the non-secure region)
Ecall-buffer-transfer	Pass a 1KB buffer to/from the secure enclave

Table 2: List of micro-benchmarks

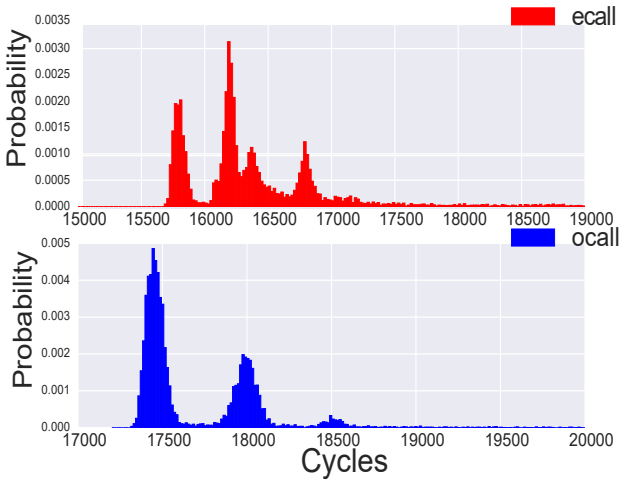


Figure 2: Distribution of ecall/ocall cycle latencies

3.2.2 Latencies associated with SGX platform. We ran all the micro-benchmarks on the hardware and measured the associated overheads. We first ran the micro-benchmarks *Empty-ecall* and *Empty-ocall*, making repeated *ecalls* and *ocalls* for 10000 measurements, respectively. Subsequently, we use *rdtscp* to measure the overheads associated with each benchmark. Based on the results, we plotted the probability density fucntion of the execution times of *ecalls* and *ocalls*. The plot is given in Figure 2. From the plot, it is clear that the latencies associated with the ecall and ocall varies from 15,500 to 17,000 cycles and 17,300 to 18,500 cycles, respectively (similar values reported by other works [34]).

For a *System-call* micro-benchmark, the average latency is 43,328 cycles. It is because each such call includes an *ecall* and an *ocall*, and 80% of system call latency is accounted for these calls. Similarly, each function call inside an enclave includes one *ecall* and as a result the latency associated with a *Function-call* micro-benchmark is close to the *Empty-ecall* latency. In an *Ecall-buffer-transfer* micro-benchmark we are transferring the data between the secure and non-secure memory. Each such transfer includes the overheads associated with an *ocall* and the data encryption scheme (while transferring data from secure to non-secure: data-in) or the overheads associated with an *ecall* and the data decryption scheme (while transferring data from non-secure to secure: data-out). Table 3 lists the latencies associated with each micro-benchmark.

Micro-benchmark	Mean latency (cyc)
Empty-ecall	16223
Empty-ocall	17872
System-call	43328
Function-call	20428
Ecall-buffer-transfer - 1KB	data in: 19297 data out: 21336

Table 3: Average latency of microbenchmarks

We want to mention here that there is no clear reason about the overheads associated with the *ecalls* and *ocalls*. Even the Intel SGX documentation is not clear about the same [8]. As a result, we tried to develop a model linking the overheads associated with SGX with the system parameters (cache and TLB) as described in the following section.

3.2.3 Effect of system parameters. To determine the parameters that are responsible for performance overhead in the SGX specific functions, we used *perf* [9] tool to read the value of different system parameters while running micro-benchmarks. The entire execution is divided into fixed size durations, called epochs (10ms), and in each epoch the *perf* tool is used to capture the system parameters. Our aim is to model the effect of such parameters on the variation of CPI of an Intel SGX as compared to a vanilla execution. We start with different variants of the *System-call* micro-benchmark. We ran these variants on both the SGX and non-SGX machine and note down the performance slowdown and the variation in the system parameters. Accordingly, we created a training dataset. Using this dataset, we did a multi-linear curve fitting in order to link the variation in the performance with the variation in the system parameters (similar to stall based performance model [10, 15]). The fitted curve is given by Equation 1.

$$\begin{aligned} CPI_{\text{sgx}} - CPI_{\text{nonsgx}} = 192.18 \times f_{\text{mem}} \times \Delta TLB_{\text{mr}} \\ +6.96 \times f_{\text{mem}} \times \Delta L1_{\text{mr}} + 12.68 \times f_{\text{mem}} \times \Delta L2_{\text{mr}} + \\ 186.3 \times f_{\text{mem}} \times \Delta L3_{\text{mr}} \end{aligned} \quad (1)$$

Here, f_{mem} is the fraction of memory instructions, *mr* refers to the global miss rate (#misses per memory instruction). Note that in our fitted model, we initially included all the relevant parameters. However, most of these parameters have a coefficient close to zero and as a result we dropped such terms in our final equation.

In order to test our fitted model we created a testing dataset by running different variants of micro-benchmark *Function-call* and some variants of *System-call* (not included in training). We used this testing data in order to find the accuracy of the fitted model. The percentage error is in the ballpark of 9-13%, which for any statistical model is acceptable. Thus, we can say that our model captures the reasons responsible for slowdown in SGX and clearly increase in the TLB miss rate (because of TLB flushing) is one of the major reasons responsible for the performance slowdown as shown in Table 4. Note that the L3 miss rates for different micro-benchmarks do not differ much between the vanilla and SGX execution and hence we do not consider this parameter in our design.

Parameters	Testbenchmark 1(%)	Testbenchmark 2(%)
$\Delta\mathbf{TLB_{mr}}$	5.13	4.92
$\Delta\mathbf{L1_{mr}}$	8.16	7.04
$\Delta\mathbf{L2_{mr}}$	1.12	1.17
$\Delta\mathbf{L3_{mr}}$	0.1	0.12

Table 4: Variations of different architectural parameters

Insight1: The TLB miss rate and the cache pollution are the major factors that affect the performance of any OS application in an Intel SGX environment.

3.3 Architectural simulation

In this section, we discuss the characteristics of some standard workloads when run on an SGX platform. We use architectural simulator to run the benchmarks and derived various insights. Please note that we incorporated the overheads associated with SGX specific operations (derived in Section 3.2) in our architectural simulator and then executed each benchmark.

3.3.1 Setup. We use a cycle-accurate architectural simulator, Tejas [26]. The simulator has been widely used in the architectural community and has been rigorously validated against native hardware. It uses the Qemu [6] tool to capture the application and OS level traces at the granularity of individual instructions and I/O operations. The latencies associated with different memory structures are calculated using Cacti 6.0 [23] and have been incorporated into the simulator. The detailed architectural parameters are given in Table 5. Note that we have used the standard benchmarks from the Sysbench suite [17].

3.3.2 Instruction Breakup. Figure 3 shows the instruction breakup of OS intensive benchmarks. We divide the entire execution of a benchmark into two sets of instructions: application and OS. *File-server* primarily consists of file system operations, and hence it has

Parameter	Value	Parameter	Value
Cores	32	Frequency	3.2 GHz
		Technology	14 nm
Out of Order Pipeline			
iTLB	64 entries	dTLB	64 entries
Private L1 i-cache, d-cache			
Latency	3 cycles	Block size	64
Associativity	8	Size	32 kB
Private L2 cache			
Latency	8 cycles	Block size	64
Associativity	8	Size	256KB
Shared L3 cache			
Latency	60 cycles	Block size	64
Associativity	8	Size	8MB
Main Memory Latency		200 cycles, 4 mem. controllers	
Cryptographic and Hashing Units			
2 PRESENT hashing units	3.2 bits/cyc	-	-

Table 5: Simulation parameters

a high percentage of OS instructions (95%). Likewise, *Mailserver* also executes a lot of system calls and thus the fraction of system call handlers is quite high (85%). In comparison, *Apache* comprises system call instructions related to web server operations (socket-create, etc.). As a result, more than 60% of this benchmark is composed of OS based instructions. The other two benchmarks, *DSS* and *OLTP* have a large number of instructions for performing application-related operations (search operations and reading database records). Their percentage of OS instructions is relatively low (around 20%). The *ISCP* decrypts the entire data that it reads over the network-socket leading to a higher percentage of application footprint(around 71%) in it. The instruction breakup of *OSCP* is somewhat similar to that of the *ISCP* benchmark primarily because the nature of both the benchmarks is similar.

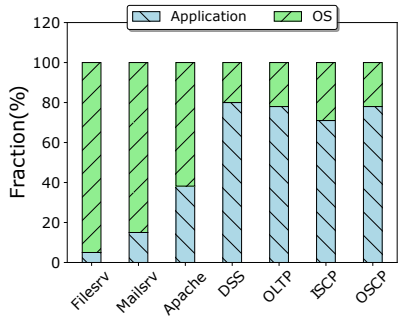
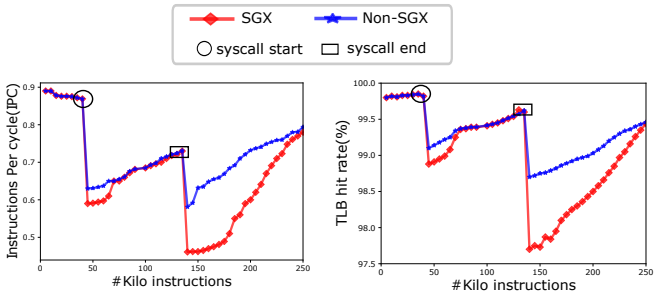


Figure 3: Instruction Breakup

Insight2: The breakup of application and OS instructions tends to be highly variable.

3.3.3 Slowdown with SGX. In Intel SGX, each system call invocation in an enclave introduces the cost of context switching in SGX which includes TLB flushing and the cache pollution (as discussed in Section 3.2.3). The TLB flushing is a costly operation as page mappings for an application have to be fetched again from the main memory when the system call returns. Figure 4 shows the performance degradation along with the TLB hit rates for a system call with SGX and without SGX for a representative execution of a *Fileserver* benchmark. We see the performance of SGX degrades compared to non-SGX (Figure 4a) set up when the system call returns which is the direct result of the degradation in the TLB hit rates in the SGX mode as shown in Figure 4b.



a) Performance impact of TLB flushing on a system call

b) Impact of TLB flushing on TLB hit rate

Figure 4: Impact of TLB flushing on the performance and the TLB hit rate (Fileserver benchmark-representative execution)

Additionally, for OS intensive workloads the typical SGX architecture executes both the application and the OS code on the same core. This eventually leads to L1 cache pollution as both user and kernel code evict each others' code from the caches (as shown in Figure 5), thus adds to the performance degradation. We take into account *SameTask* and *OtherTask* tasks that provide us the insight of the evictions. *SameTask* refers to the pollution caused by the similar kind of tasks. For example: an OS task evicting other OS task and similarly an application task evicting other application task. In comparison *OtherTask* refers to the eviction caused by the dissimilar type of tasks. For example: an application task evicting the OS task and vice-versa. We observe that for all the benchmarks, the eviction of data from the caches is mainly because of *OtherTask* (around 74.2%). This means mainly the application and kernel data evict each others' data from the caches.

We finally simulate all the overheads of SGX for OS intensive workloads and we observe that the performance degradation in SGX is around 73.3% (shown in Figure 6). The TLB flushing amounts to 48.2/73.3% of the slowdown while the rest 25.1% is attributed to the L1 cache pollution. Therefore, we require a scheme that mitigates the effect of enclave exits and the cache pollution in OS intensive workloads.

Insight3: The performance degradation in Intel SGX for OS intensive workloads is 73.3% because of the enclave exits and L1 cache pollution.

4 SECSCHED DESIGN

4.1 Overview

Intel SGX by default flushes the TLBs upon a context switch because it does not store bits in the TLB to indicate that a given page is only supposed to be accessed by a secure or a non-secure region. The flushing of TLBs is very expensive as shown in Section 3.3.3. In addition to this, for OS intensive workloads, the OS and the application instructions compete for the space in the caches. Weisse

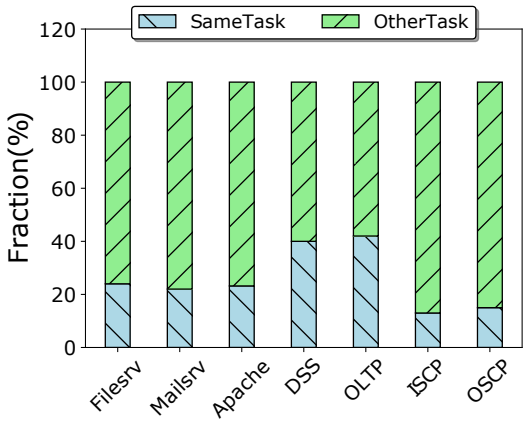


Figure 5: Breakup of evictions

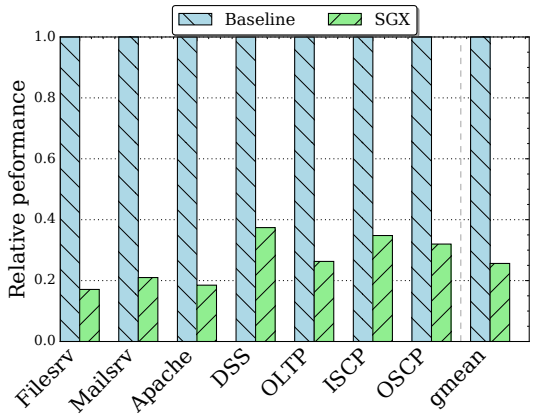


Figure 6: Performance overhead in SGX

et.al [34] try to reduce context switches altogether by pinning threads to cores but it introduces the core idleness in the system. *SGXKernel* mitigates the overhead of *Hotcalls* by migrating the threads to the idle cores randomly from the pool of threads but it leads to the destructive interference as the threads are migrated randomly. To mitigate the issues of *SGX*, *Hotcalls* and *SGXKernel*, we propose *SecSched* that schedules threads across the cores by computing the similarity in the working set of the threads and the cores using the Cuckoo filters. Additionally, we maintain the count of the misses of the LLC for each thread and use this miss count information to efficiently schedule threads across the cores.

4.2 Maintaining Miss Count

We maintain a 16-bit miss count (chosen experimentally) for each thread that keeps track of the lower level cache(LLC) misses for a particular thread. The miss counts for all the threads are maintained in a shared cache based structure called as *MissCountTable (MCT)*. The MCT is an array of 64 entries containing a 16-bit miss count for each thread where each entry of the table is indexed by the thread id.

Updating the miss count: Whenever a thread encounters a memory instruction, it sends the request to the L1 cache. Along with the request, thread id (6 bits) is sent. If there is a L1 miss, the request along with the thread id is propagated to the L3 cache (similar approach done by [20]) and if there is a miss,the request to fetch the block is sent to the main memory. At the same time, the *MCT* table is indexed by the thread id and the corresponding miss count of the thread is updated. Note that updating the *MCT* is not in the critical path. In case the miss count for a thread saturates (reach 65,535), we halve the miss counts of all the threads to maintain the consistency across the miss counts of all the threads.

4.3 Scheduling using Miss Counts

We consider a system with 32 cores, and 32 cache banks. The cores and cache banks are organized as a chess board (similar to [3]). We consider a set of applications $\mathcal{A}$, where each thread has two modes: application and OS. In the application mode it executes application code, and is considered to execute securely using our secure hardware. The execution in the OS mode is not secure. These are standard assumptions [8, 30, 34].

We run a dedicated secure scheduler in software, which resides in a permanent enclave of its own. The kernel's scheduler invokes it with a special instruction. This secure scheduler maintains two queues per core: waiting queue (waiting for some external event like a DMA message) and a ready queue. We map a thread in the ready queue to a core by supplying the mapping as a *hint* to the kernel's scheduler, which the latter considers with a very high priority. To create this mapping we add a Cuckoo filter to the context of each thread, and also to each core.

In *SecSched*, we maintain one Cuckoo filter of $\mathcal{N}$ entries per thread, and one per core (2KB per thread and a core). We insert an entry after a TLB miss. The Cuckoo filter is a part of a thread's context. In our experiments, we found $\mathcal{N}$ = 1024 as the optimal value as we shall see in Section 5. The size of the fingerprint is 16 bits (chosen experimentally) in our design to reduce the false positive rate of the Cuckoo filter. We use the light weight PRESENT [7] block cipher as a one-way hashing engine in hardware to generate the fingerprint and the hash. Consider the process of generating the fingerprint $\mathcal{F}$ ($\mathcal{H}$ is generated in a similar manner). The physical page id is 36 bits. We concatenate this with a 28 bit random initialization vector, to generate a 64-bit string. In addition, we have a 80-bit random key. PRESENT generates a 64-bit cipher text after 20 clock cycles. The fingerprint is the 16 LSB bits of the generated cipher text. Since we are using this scheme for hashing, we wish to minimize the time taken, and thus PRESENT is a suitable candidate. Additionally, Cuckoo filters disallow false negatives. When an entry cannot be inserted into the Cuckoo filter, we have the notion of partial deletion, where we delete 50% of the elements in the array. This achieves a tradeoff between the probability of successful insertion and keeping a record of past history.

4.3.1 Finding Similarity between Cuckoo Filters. To find the similarity of working sets between a thread and a core, we need to find the intersection between Cuckoo filters. This can be easily done in $\theta(N)$ time by considering an entry at a time of one Cuckoo filter and searching for it in the other one (see Figure 7). We compute the intersection by performing the bitwise-and operation between the two 16-bit fingerprints. The size of the intersection is an indication

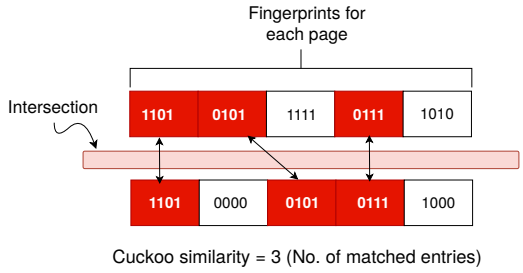


Figure 7: Intersection of two Cuckoo filters

of the similarity in working sets. Let us refer to this as the *Cuckoo similarity*.

4.3.2 Fast/Slow Matching with Miss Counts. Our scheduling mechanism *SecSched* uses two algorithms - *Fast Matching* and *Slow Matching* (refer to Figure 8). Fast Matching is used when a system call is encountered in a thread. It migrates the thread to some other core with an objective to minimize the cache pollution. Slow Matching is needed to reduce the core idleness of the system with an added objective of mitigating the effect of the cache pollution.

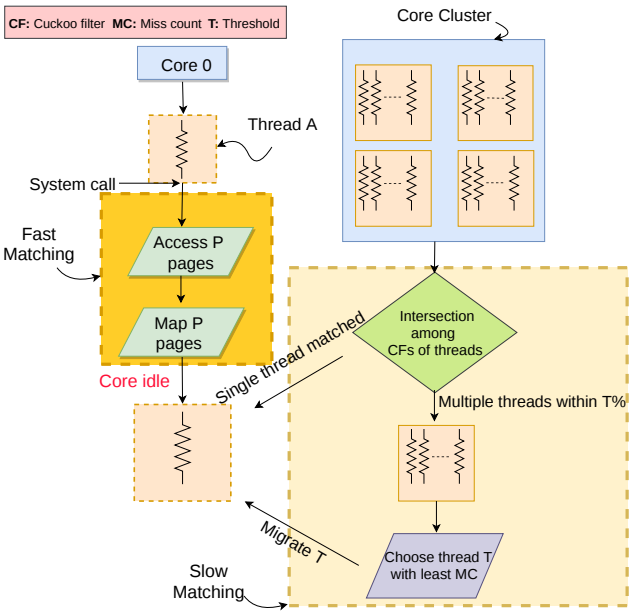


Figure 8: *SecSched* design

Fast Matching

Whenever a system call is invoked in a thread, we run that particular secure thread till we access $\mathcal{P}$ pages (as shown in Figure 8), in the new privilege level. In our experiments, we found $\mathcal{P}$=4 as the optimal value. We run these page accesses on a new TLB called as *VictimTLB* consisting of entries equal to $\mathcal{P}$ and additionally, we lock all the entries in the default TLB to avoid any changes by the kernel. Note that this *VictimTLB* is maintained for each core. During these accesses, the system call executes some libraries which gives us the hint of the working set in the near future. We compute the Cuckoo similarity in hardware between the thread whose context was switched (set $\mathcal{P}$) and the Cuckoo filters of other threads running on other cores. We choose the core that has the highest Cuckoo similarity. The intuition is that $\mathcal{P}$ contains the relevant part of the library and kernel code that are predictive of the actions of the thread in the future. If a core contains the pages in $\mathcal{P}$ then it most likely contains the blocks that the thread will need in the near future as well. We call this scheme as *Fast Matching*. When the system call thread finishes executing on the other core and returns back, it is sent to the waiting queue and waits for its turn to be executed. There can be a possibility that a system call finishes executing in less than $\mathcal{P}$ pages. In that case, we do not migrate the thread to the other core. Instead, we let it run and access the pages using the *VictimTLB* to avoid the migration overhead. The *Fast Matching* process takes roughly 0.8μs at 3.2 GHz (mapping of 4 pages on one among the 32 cores takes 4 X 20 X 32 = 2560 cycles), where 20 cycles is the hash latency(refer to Table 5), which is insignificant. We only do this for context switches from the application to system calls and interrupt handlers. Since the core from which the thread is migrated becomes idle, we propose to use another algorithm to reduce this core idleness which we refer to as *Slow Matching*.

Slow Matching

In *Slow Matching*, we use a work stealing based mechanism to migrate threads from other cores to this core in order to reduce the core idleness (refer to Figure 8). We rank waiting threads based on their Cuckoo similarities with the Cuckoo filter of the idle core. If there is a clear winner, then we choose that thread; however, if the values are close, then there is a need to use another metric as a tie-breaker. **In *SecSched*, we consider all the similarity values that are within $\mathcal{T}$% of the maximum similarity, and then in that set we choose that thread that has the least miss count.** The overhead of *Slow Matching* is significant- around 410μs because we need to match the Cuckoo filter contents (1024 entries) of the present core with all the 64 threads(64 X 20 X 1024=1,310,720 cycles at 3.2 GHz). Since the overhead of Slow matching is significant, this approach can be used for regular tasks, not for interrupt handlers. Moreover, since *Slow Matching* takes into consideration the Cuckoo filter similarity of the threads, the application(secure) threads will be scheduled on the application cores only and the same holds for the kernel threads. This ensures that the kernel threads cannot modify the contents of the application (secure) threads and hence there is no need for TLB flushing. Moreover, to avoid starvation, similar to the *STARVATION_LIMIT* parameter in Linux, we introduce a threshold, which is defined as the number of entries in the ready queue multiplied by 10 ms. If a task waits for more than that period, then we raise its priority to maximum, and immediately schedule it when we get a chance (next syscall or interrupt).

4.4 Hardware Modifications

SecSched requires few hardware modifications that need to be incorporated for the final design. They are 1) Maintaining Cuckoo filter per core and a thread, 2) a *MissCountTable (MCT)*, 3) a *VictimTLB* that holds the $\mathcal{P}$ number of pages in our Fast matching algorithm, 4) Hash unit(we use PRESENT) to compute the hashes and 5) special assembly instructions to load and store fingerprint values.

5 EVALUATION

In this section, we evaluate our design decisions and compare our final scheme with the state-of-the-art schemes, *Hotcalls* [34], *SGXKernel* [33] and a baseline implementation, *Baseline_enc*, which implements *Intel SGX* [8]. The simulation infrastructure is described in Section 3.3.1 and the architectural parameters are given in Table 5. A system running OS intensive workloads typically has a lot of threads that remain idle because they wait for data from secondary storage. Hence, to evaluate the effect of scheduling on the overall performance of the system, it is a standard practice to spawn more threads as compared to the number of cores [14]. As a result, we spawn 64 threads on 32 cores (2X load factor [14]) for evaluating the performance of our scheme. We define performance as the inverse of the execution time. Note that we simulate each benchmark with a warm up period of 30 million instructions and then with the followup simulation of 1 billion instructions.

5.1 Synthesis Results

We designed our new hardware structures in VHDL and used the Cadence Genus tool [1] to estimate the area overhead. The designs were fabricated with a 14nm technology. Table 6 shows the size per structure and the associated overheads. We observe that the area overheads for our new proposed hardware structures are minimal with *MCT* having the least area overhead as it is shared across the cores. Moreover, the time to access these structures is typically less than a cycle.

Hardware Structure	Area(in μm^2)	Total Area Overhead
VictimTLB	≈ 92	0.00073%
Cuckoo Filter	$\approx 4,700$	0.04%
Miss Count Table (*MCT*)	≈ 300	$7.5 * 10^{-5}$%

Table 6: Area overheads of different structures

5.2 Parameter Selection

The main aim of this section is to decide the optimal value for different parameters used in our scheme such as size of Cuckoo filter, number of pages to be accessed in *Fast Matching*, and the threshold value in the *Slow Matching*.

5.2.1 Cuckoo Filter Characterization. We ran multiple simulations by varying the size $\mathcal{N}$ of Cuckoo filter from 256 to 2048. The variation in the performance of our design as compared to *SGXKernel* for different sizes is given in Column (1) of Table 7. It is clear that by increasing the Cuckoo filter size there is nearly 40% improvement in performance. However, beyond a certain point the improvement in performance saturates. Thus, we have chosen a Cuckoo filter with 1024 entries in our final design. Using a higher size Cuckoo filter increases the storage overhead with a gain of negligible amount in the performance.

The improvement in the performance with the increase in Cuckoo filter size is attributed to the increase in i-cache and d-cache hit rate as given in Column (2) and Column (3) (4-7% improvement). This improvement is because using a large Cuckoo filter enables us to capture the footprint of a thread much better as compared to a filter with smaller size.

Moreover, the false positive rate (falsely indicating the presence of a certain value when it is not actually present) decreases linearly with the increase in Cuckoo filter size (see Column(7) of Table 7). Thus, taking all the parameters into account, we chose $\mathcal{N} = 1024$ for our design.

5.2.2 Number of Pages in Fast Matching vs Performance. In the *Fast Matching* algorithm, we have to determine the number of pages ($\mathcal{P}$) that a system call should be allowed to access after an application makes the system call. We varied the $\mathcal{P}$ and calculated the effect of same on the performance of the system. The relative performance for different values of $\mathcal{P}$ (varied from 1-8) is given in Figure 9. It is clear from the figure that $\mathcal{P} = 4$ is the optimal value. For a smaller value of $\mathcal{P}$ we are not able to get the better hint of the working set of the system. However, for a larger value the working set of a system is better captured but at the cost of higher cache pollution resulting in performance degradation. Thus, there is a tradeoff between the how well the working set has been captured and the cache pollution. In our results, $\mathcal{P} = 4$ proves to be the optimal value and hence the same has been selected in the final configuration.

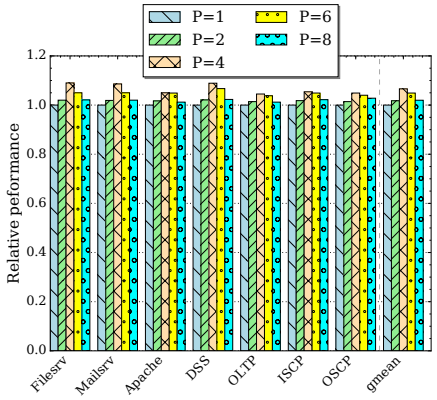


Figure 9: Relative Performance of *SecSched* with varying $\mathcal{P}$

5.2.3 Threshold Value in Slow Matching vs Performance. In the *Slow Matching* algorithm, when we have multiple candidates for the working set similarity, we choose a tie-breaker and decide on the basis of the thread's miss count. If the working set similarity difference among the threads is within a certain threshold, we choose the thread with the least miss count. We observe the performance of our *SecSched* by varying the threshold $\mathcal{T}$. There is a tradeoff between the estimation of the working set and the lower level misses (accesses to main memory) in this case. The aim is to choose that thread whose Cuckoo filter contents have the maximum similarity with the local thread and at the same time, the lower level misses for the thread are less. We found that the performance is optimal for the threshold value($\mathcal{T}$) of 10% (refer to Figure 10).

Summary: Based on the analysis, we decided the following values for different parameters in our design: **# entries in Cuckoo filter($\mathcal{N}$) = 1024, $\mathcal{P}$ = 4, and $\mathcal{T}$ = 10**. We use these values in our final design and then compared it with the state-of-the-art schemes as discussed in the following sections.

Benchmarks	Cuckoo filter size	Performance(%) inc. over SGXKernel	i-cache HR(%)	d-cache HR(%)	# syscalls	FM(%)	SM(%)	FP(%)	Performance(%) with miss counts
		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Fileserver	**256**	9.65	83.01	77.1	2708	0.07	1.32	24.4	1.3
	512	13.74	87.1	82.02	2708	0.13	2.65	20.5	1.9
	1024	17.4	93.36	88.9	2708	0.27	5.21	12.3	3.6
	2048	17.48	93.38	88.95	2708	0.35	7.8	11.3	3.62
Mailserver	**256**	10.4	82.5	79.01	2958	0.09	1.45	25.7	1.8
	512	14.56	88.0	82.1	2958	0.162	2.92	21.42	2.6
	1024	19.0	94.3	88.17	2958	0.3	5.92	11.92	3.82
	2048	20.8	95.6	90.67	2958	0.61	7.83	8.92	4.01
Apache	**256**	1.6	81.23	76.2	2434	0.05	1.15	32.03	0.46
	512	2.4	85.5	81.01	2434	0.09	2.31	21.6	0.78
	1024	5.12	89.3	88.2	2434	0.18	4.61	16.3	1.1
	2048	6.32	91.3	90.2	2434	0.32	5.98	12.05	1.41
DSS	**256**	13.53	82.3	78.8	1992	0.02	1.09	23.68	1.57
	512	18.51	87.1	83.0	1992	0.05	2.21	18.21	2.39
	1024	22.8	94.87	89.06	1992	0.09	4.38	9.68	5.42
	2048	24.9	95.92	91.22	1992	0.18	6.58	7.12	5.89
OLTP	**256**	4.38	77.2	81.1	1728	0.031	1.02	26.23	0.87
	512	7.18	81.2	84.01	1728	0.062	2.08	22.66	1.18
	1024	9.12	89.4	89.3	1728	0.11	4.23	15.2	1.88
	2048	10.13	91.1	91.68	1728	0.22	6.28	10.05	1.97
ISCP	**256**	4.89	79.2	80.4	1691	0.028	1.01	24.21	0.68
	512	8.23	84.2	84.5	1691	0.065	2.03	20.36	1.5
	1024	11.1	91.4	89.9	1691	0.09	4.13	14.29	2.7
	2048	12.39	93.2	93.58	1691	0.2	6.1	9.05	2.83
OSCP	**256**	4.94	78.94	81.45	1687	0.024	0.99	22.13	0.76
	512	8.11	83.78	85.01	1687	0.049	1.92	19.26	1.44
	1024	10.1	89.43	89.12	1687	0.081	3.84	13.9	2.56
	2048	11.49	91.23	91.87	1687	0.18	5.78	8.86	2.65

HR → Hit rate, *FM* → Fast match overhead, *SM* → Slow match overhead, *FP* → False positive rate

Table 7: Impact on different architectural parameters for different Cuckoo filter sizes

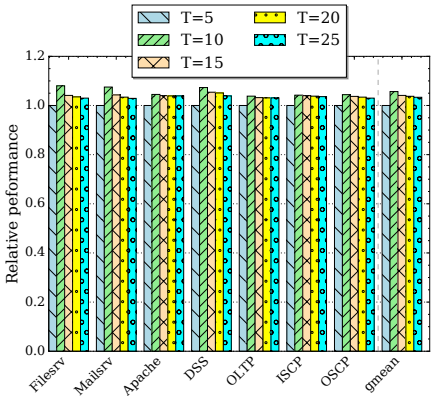


Figure 10: Relative Performance of *SecSched* with varying $\mathcal{T}$

5.3 Analysis

We study four micro-architectural parameters to understand the behaviour of different schemes for different kinds of benchmarks and based on these parameters, we infer the performance for each scheme:

❶ TLB hit rate (refer to Figure 11)
❷ i-cache hit rate (refer to Figure 12)
❸ d-cache hit rate (refer to Figure 13)
❹ core idleness (refer to Figure 14)

Baseline_enc runs both the application and OS threads for OS intensive workloads on the same core; the application code and kernel code evict each others' data from caches, thus leading to a decrease in the i-cache and d-cache hit rates (refer to Figures 12 and 13). In addition to this, on each system call invocation, TLB flushing is done. The frequent TLB flushes lead to the decrease in the TLB hit rates (Figure 11) as the contents are to be read back from the main memory. Since the application and the OS threads run on the same core , the core idleness is minimal in *Baseline_enc* (refer to Figure 14).

Hotcalls pins the threads to cores, thus avoiding TLB flushing. However, it does busy waiting that leads to the CPU idle time of roughly around 48% (see Figure 14). This is because the application threads wait for the OS part to finish using busy-wait loops. However, *Hotcalls* scores in terms of higher cache hit rates(refer to Figures 12, 13), which is expected because of the enforced locality.

SGXKernel unlike *Hotcalls*, does not keep the core idle when a system call is encountered in a thread. Instead, it ensures asynchronous system calls i.e, when a system call is encountered in a thread, till the result of the thread comes back, it schedules other application threads on the same core to ensure that the core does not remain idle. Each core has a runnable queue and these queues are re-balanced by periodically sending the threads from busy runnable queues to idle ones, thus ensuring that the core idleness is minimum(refer to Figure 14). However *SGXKernel* does not take into account the working set of the application thread. Since a new thread is scheduled on a core on a random basis, the new thread will pollute the caches and also evict the TLB entries of the thread that was running previously, leading to the lower TLB, i-cache and

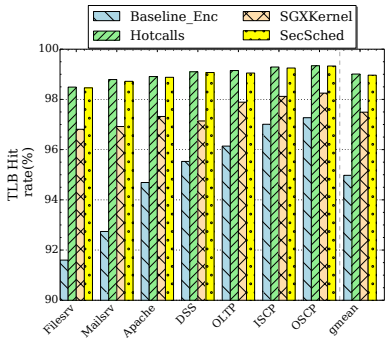


Figure 11: TLB hit rates

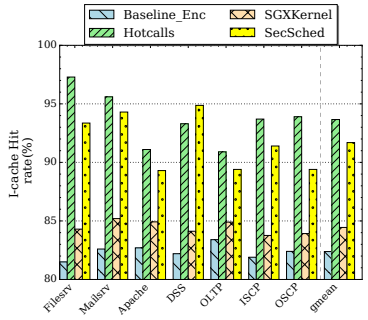


Figure 12: i-cache hit rates

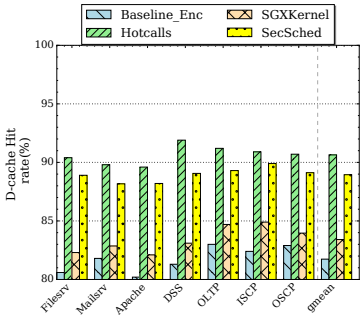


Figure 13: d-cache hit rates

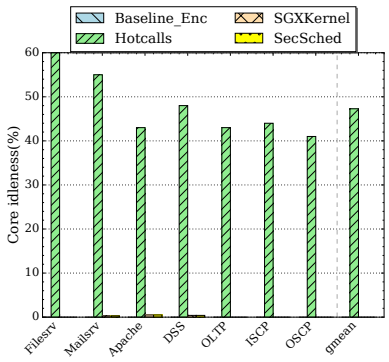


Figure 14: Core idleness

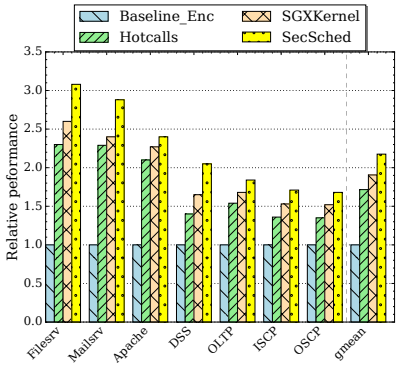


Figure 15: Performance

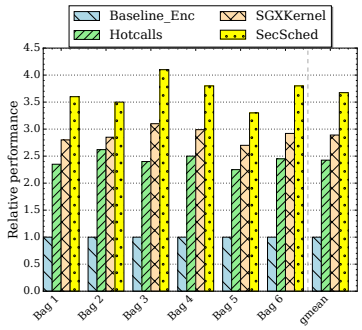


Figure 16: Relative performance for a bag of tasks

d-cache hit rates in *SGXKernel* than *SecSched* (differ by 1.5-2%, 7-8% and 5-6% respectively).

SecSched avoids the core idleness by a smart work stealing that steals the thread on the basis of working set similarity of the threads using the Cuckoo filters. This results in a negligible core idleness(close to 0%) in *SecSched* as shown in Figure 14. In addition to this, *SecSched* ensures that the application(secure threads) and the kernel threads run on separate cores, thus avoiding the need of TLB flushing resulting in high TLB hit rates than *SGXKernel* (1.5-2% more). However, since *SecSched* migrates tasks on the basis of matching filters(leading to destructive interference), the i-cache hit rates in *SecSched* are slightly lower than *Hotcalls* by 2-3%. Moreover, in *SecSched* the data locality of the OS threads and the resultant d-cache hit rate suffers as compared to *Hotcalls* (lower by 2.5%) because of thread migrations.

We want to mention here that in our experiments, we increased the number of threads to 128 by keeping the *MCT* size same as for the default setup. Increasing the number of threads can lead to the aliasing in the *MCT*. However, inspite of the aliasing, we did not observe any significant change in the performance for 128 threads.

5.4 Performance Comparison

We now observe the performance comparison among the different schemes based on the aforementioned micro-architectural parameters. The mean (geometric) improvement of *SecSched* over *Hotcalls* and *SGXKernel* is 25.8% and 15.93% respectively for OS intensive workloads(refer to Figure 15). Let us try to answer the reasons for the performance improvement.

SecSched does not keep the core idle, thus reducing the core idleness of the system compared to *Hotcalls* (Figure 14). Moreover, since it schedules threads on the basis of working set similarity, the cache pollution is almost minimal(Figures 12, 13). Infact it is better than *SGXKernel* because of the random scheduling of threads in *SGXKernel*. Additionally, since threads migration is random, the TLB entries are evicted by the random threads, leading to the decrease in TLB hit rates (Figure 11) in *SGXKernel* compared to *SecSched*, thus better performance. However, *SecSched* and *SGXKernel* have minimal core idleness(refer to Figure 14) since they migrate threads to the idle cores. In addition to these factors, our scheme *SecSched* performs better because of the miss count information in the *MCT*, used as a tie-breaker among the multiple threads in Slow Matching. We found using the miss count information, our scheme *SecSched* performs around 2-3.5% better than without miss count (Column(8) of Table 7) because it improves the throughput of the system as the thread with least miss count is chosen. Finally, to evaluate the overhead of the software scheduler, we found that the throughput (disregarding spin locks) is 0.8% more than what we get by back calculating from the performance data because of the overhead of the scheduler.

Let us study some benchmark specific trends. We see a significant improvement in the *Fileserver* and *Mailserver* benchmarks because in the *Hotcalls* scheme the idleness is high for both the benchmarks (see Figure 14). In addition to this, there is more destructive interference for *SGXKernel* in case of *Fileserver*, *Mailserver* and *DSS*, therefore the performance of *SecSched* is better for these benchmarks. The gains for *Apache* are much lower because of its low core idleness with *Hotcalls*. Moreover, the difference in the

icache, dcache and TLB hit rates between *SecSched* and *SGXKernel* is very less, thus performance gains are much lower.

5.5 Sensitivity Analysis

5.5.1 Performance without Miss Counts per Thread. Let us now look at the performance of our *SecSched* scheme if we don't take the miss count information in our scheduler. Column(1) of Table 7 shows the performance that we gain when we don't consider miss counts in the slow matching process. We observe that even without the miss count information, the gains are modest (mean:13.52%). This is mainly because in most of the cases the Cuckoo filter overlaps are very dissimilar, thus avoiding the use of miss count information in those cases.

5.5.2 Overhead of Fast/Slow Matching. We now look at the overheads of fast and slow matching when threads are migrated across the cores. Since the threads are migrated across the cores when a system call is invoked, we look at the number of system calls in column (4) of Table 7. The system calls vary from 1728 (*OLTP*) to 2958 (*Mailserver*). For each system call we first perform a fast match. In column (5) we show the overhead of fast matches, which is defined as the percentage of the execution time that is spent in these operations. Note that not all of them are on the critical path. Hence, the effect on the critical path will be lower than the time overhead that we calculate. We find the overhead of fast matches to be less than 0.27%, which is insignificant. Now, let us compute the overhead of slow matches in the same manner (see column (6)). Here the overhead varies from 1.02% (*OLTP*) to 5.92% (*Mailserver*). This follows the same trend as the number of system calls. However, the actual overhead is lower than 5% in most cases because of the nature of critical paths that we empirically observed in our programs.

Bag ID	Constituent Benchmarks
Bag 1	*Fileserver,Mailserver*
Bag 2	*Apache,DSS, OLTP,Mailserver*
Bag 3	*OLTP, Apache*
Bag 4	*Apache, DSS*
Bag 5	*Mailserver, ISCP, DSS, OLTP*
Bag 6	*Mailserver, Fileserver, OSCP, Apache*

Table 8: Suite of OS intensive workloads

5.5.3 Performance Impact for a Bag of Tasks. In any real world scenario, we can have many applications running at the same time. Thus, it is necessary to evaluate the behavior of our system in such scenarios. In this section, we compare the state-of-art proposals with our proposed schemes for a bag of OS intensive benchmarks. We equally divide the threads across the constituent workloads. We observed that as compared to *Hotcalls* and *SGXKernel*, *SecSched* performs 54.1% and 27.6% better respectively (refer to Figure 16). We observe that the contribution of miss count information in bag of workloads is comparatively significant(around 7-9%).

6 RELATED WORK

Switchless Calls in Intel SGX

Some recent works propose schemes to reduce the overhead of context switching in environments like Intel SGX for OS intensive applications. *Hotcalls* [34] introduces an unencrypted shared memory channel between the user code in the secure enclave and the untrusted code in the kernel responsible for executing system calls. This is done to prevent costly enclave switches. However, it leads to a lot of core idling particularly when the load across the OS and application core is unbalanced. *SGXKernel* [33] reduces the core idleness by migrating the threads randomly across the cores whenever a system call is invoked at any core. However, migrating the threads randomly pollutes the caches, thus leading to the reduced performance.

Eleos [24] introduces exit-less paging and exit-less system calls. It enables exit-less system calls by transparently assigning them to a remote procedure running in another thread(worker thread) similar to what Hotcalls does. *SCONE* [2] shows how to run docker containers inside the enclaves and provides techniques to improve the container performance similar to the approaches proposed by *Hotcalls* and *Eleos*. However, it needs changes to the kernel and the support of SGX aware libraries which we do not rely upon in our design. Tian et.al [32] proposes a worker based scheduling algorithm that adjusts the number of workers in response to the changing workloads. The analysis focuses only on the idleness of the CPU cores that are dedicated entirely for kernel code.

Scheduling for OS Intensive Tasks

SchedTask [14] divides the execution into sets of functions called *superfunctions* based on the similarity of instruction pages, and functionality obtained using the Bloom filter. Applications with a similar memory footprint are scheduled on the same core. SchedTask improves upon *FlexSC* [28] that introduced exit-less system calls for OS intensive applications. It executes different system call handlers on different cores to improve locality, and unlike SchedTask it lacks features for work stealing. Even though our approach is broadly inspired from these works, it is different in the sense that we use a lightweight approach to quantify the similarity in memory footprints using a Cuckoo filter, we do not require any modifications to the OS code, and we need not be aware of the inner details of the OS.

7 CONCLUSION

In this paper, we present a scheduling algorithm *SecSched* that reduces the overhead of context switches in Intel SGX. We propose a novel Cuckoo filter based scheduling that captures the similarity in the working set of the threads. In addition, we also maintain a thread based miss count information that acts as a tie-breaker when the Cuckoo filter contents are similar among multiple threads during the work stealing approach of Slow Matching. By incorporating these changes in Intel SGX, we are able to outperform the competing schemes *Hotcalls* and *SGXKernel* by 54.1% and 27.6% respectively.

8 ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments. We also thank Sandeep Kumar for providing valuable feedback on the initial version of the manuscript; it allowed us to immensely improve our paper.

REFERENCES

[1] [n.d.]. Genus Synthesis Solution. https://www.cadence.com.

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

[3] Anuj Arora, Mayur Harne, Hameedah Sultan, Akriti Bagaria, and Smruti R Sarangi. 2015. FP-NUCA: A fast NoC layer for implementing large NUCA caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 9 (2015), 2465–2478.

[4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.

[5] Sundeep Bajikar. 2002. Trusted platform module (tpm) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation* (2002), 1–20.

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[7] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. 2007. PRESENT: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 450–466.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[9] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18.

[10] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)* 27, 2 (2009), 3.

[11] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 75–88.

[12] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Secure live migration of SGX enclaves on untrusted cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 225–236.

[13] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security & Privacy* 6 (2016), 54–62.

[14] Prathmesh Kallurkar and Smruti R Sarangi. 2017. Schedtask: a hardware-assisted task scheduler. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 612–624.

[15] Tejas S Karkhanis and James E Smith. 2007. Automated design of application specific superscalar processors: an analytical approach. *ACM SIGARCH Computer Architecture News* 35, 2 (2007), 402–411.

[16] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.

[17] A. KOPYTOV. [n.d.]. SysBench : a system performance benchmark. *http://sysbench.sourceforge.net/* ([n. d.]). https://ci.nii.ac.jp/naid/10026762797/en/

[18] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. 2012. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 389–400.

[19] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. 2016. PoisonIvy: Safe speculation for secure memory. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 38.

[20] Wenming Li, Lingjun Fan, Zihou Wang, Xiaochun Ye, Da Wang, Hao Zhang, Liang Zhang, Dongrui Fan, and Xianghui Xie. 2015. Thread ID based power reduction mechanism for multi-thread shared set-associative caches. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 1–4.

[21] Thomas Morris. 2011. Trusted platform module. In *Encyclopedia of cryptography and security*. Springer, 1332–1335.

[22] Thomas Morris. 2011. Trusted platform module. In *Encyclopedia of cryptography and security*. Springer, 1332–1335.

[23] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009), 22–31.

[24] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 238–253.

[25] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 67–80.

[26] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*. IEEE, 47–54.

[27] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, and D Levitan. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.

[28] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 33–46.

[29] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review* 35, 4 (2005), 181–192.

[30] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2014. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 357–368.

[31] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 665–678.

[32] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*. ACM, New York, NY, USA, 22–27. https://doi.org/10.1145/3268935.3268942

[33] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: a library operating system optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference*. ACM, 35–44.

[34] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 81–93.

[35] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. ACM, 21–30.

[36] Dmitry P Zegzhda, ES Usov, AV Nikol'skii, and E Yu Pavlenko. 2017. Use of Intel SGX to ensure the confidentiality of data of cloud users. *Automatic Control and Computer Sciences* 51, 8 (2017), 848–854.