SGXFault: An Efficient Page Fault Handling Mechanism for SGX Enclaves

Omais Shafi Pandith
Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
Email: omais.shafi@cse.iitd.ac.in

***Abstract*—Trusted Execution Environments (TEEs) such as Intel SGX are becoming a commonplace for the security in server processors. Intel SGX provides the guarantees of confidentiality, integrity and replay attack protection for a specific region of memory. However, the downside of it is the performance bottleneck due to the context switching overhead and the page faults for the larger memory footprint applications. In this paper, we propose a scheme *SGXFault* which ensures for major part of the execution, the pages are available in the secure memory when needed. We do this decision at the last level cache (LLC) by locking the blocks of the frequent pages to the LLC using Cache Lockdown mechanism. In addition to this, we do a page-level prediction based prefetching when there is a miss in the LLC. Using the combination of both the approaches, we are able to outperform baseline Intel SGX and recent competing scheme by around 18.6% and 17.8% respectively.**

I. Introduction

It is important to protect a user application from different security attacks, especially for privacy-sensitive applications that store and maintain user accounts and passwords. Moreover, these applications are deployed in the cloud environments which itself poses a threat to protect the privacy of the data. To address such issues, Trusted Execution Environments (TEEs) [1]–[4] have been released by the vendors. TEE is an isolated region that provides the privilege to execute the code in a secure manner. TEEs have been embedded into many processors and are available via hardware extensions. Intel Software Guard eXtensions (SGX) [1], for instance, has been implemented into Intel processors since 2015.

Intel SGX [1], [5] provides a set of instructions that allow the code to execute securely inside a region of memory called as *enclave*. The data inside the enclave is encrypted by the memory encryption engine, which is located at the last level cache's boundary. The typical size supported by Intel for enclave is maximum 128MB and thus an application of maximum size 128 MB can be fitted inside the enclave. For the applications with a larger memory footprint, the size of the enclave is not sufficient. In such cases, the pages have to fetched on-demand from the untrusted memory to the secure memory (enclave). This on-demand fetching triggers a page fault in Intel SGX and forces the application to move out of the enclave known as asynchronous exit. This copying of pages from the untrusted memory to the secure memory has significant overheads associated with it. Therefore, if the application has a large memory footprint, there will be significant number of page faults associated with it leading to a massive performance overhead. The typical cycles that each page fault encountered uses is around 64,000 cycles [6], [7] which is huge. Some recent works such as *Eleos* [8] reduces the performance overheads associated with page faults by introducing an in-enclave application level paging. On similar lines, Liu. et al [7] propose *DFP* (Dynamic Fault History-Based Preloading),the recent competing work, uses history based page prefetching and source level code instrumentations to mitigate the overhead of page faults. We, however, do not need to augment the source code. Instead, we argue that not all the pages in any application are used frequently. We exploit this and aim to maintain the frequent pages in the caches for most of the execution time using the Cache Lockdown strategy.We maintain a counter for every page that keeps track of the frequency of the page used in the execution. In addition to this, we also ensure that we prefetch the non-frequent pages from the untrusted memory before they are accessed by keeping the track of all the pages accessed in the recent history compared to the page faults pattern done by [7]. We observe that we have significantly lesser wrong page replacements compared to the recent competing scheme *DFP*. We show that we are able to outperform the competing schemes, *Baseline-SGX* and *DFP* by 18.6% and 17.8% respectively without any compromise in the security for a suite of 7 server workloads. The key contributions of the paper are:

❶ We propose a scheme that categorizes the pages into frequent and non-frequent pages using the counter maintained for each page.
❷ For the frequent pages, we ensure that they are present in the caches for most part of the execution using the Cache lockdown strategy.
❸ For the non-frequent pages, we do the prefetching decision at the LLC only which ensures that the pages are available in the secure memory before they are accessed.
❹ By using both cache lockdown and prefetching strategies, we are able to outperform *Baseline-SGX* and *DFP* by 18.6% and 17.8% respectively.

The rest of the treatise is organized as follows. Section II provides a brief background and the motivation, Section III discusses the related work, Section IV describes our design, Section V evaluates our design, and we finally conclude in Section VI.

II. Background and Motivation

Intel SGX (Secure Guard eXtensions) provide a set of instructions that allow the code to be executed securely inside the region of memory called as *enclave*. An enclave is a region where programs execute securely regardless of the Operating system or the hypervisor. Moreover, the secure context created by the SGX has only user level priviledges. In addition, all the data inside the enclave is encrypted using the Memory Encryption Engine [9]. Intel SGX gurantees the confidentiality and the integrity of the data using the SGX integrity trees [9]–[11] that is the hash of the counters used for counter mode encyrption in SGX. The counter mode encryption [12], [13] is a block cipher mode of encryption used by the Memory Encryption Engine for encryption the data in the enclaves.

The enclave is made up of three components - *Enclave Page Cache(EPC)*, *Enclave Page Cache Map(EPCM)* and *metadata*. The EPC is used to store all of an application's code and data. The EPCM is used to store the mapping between the virtual and physical addresses of the protected pages. Lastly, the metadata contains the information that is used to ensure the confidentiality and the integrity of the data.

The SGX hardware including EPC is managed by the Operating system even if it is not trusted. It should be noted that the application can only use a small physical enclave memory and if the application size is larger than the specified size (currently 128 MB), the EPC paging mechanism ensures that the pages are swapped from the untrusted memory to the enclave when there is a need. A page fault is caused each time an application accesses a page which is not present in the enclave.This page fault triggers a moving application from an enclave known as *asynchronous exit*, which is invoked by the *AEX* instruction. This page fault is handled by the untrusted OS and the page is swapped to the trusted region if the integrity and the confidentiality of the data is preserved and once the page is swapped, the EPCM is updated and if there is no space available in the enclave, the existing EPC page is evicted. The OS swaps in the faulty memory page into the enclave using *ELDB/ELDU* instructions and finally the application resumes its execution using *ERESUME* instruction. Thus each page fault triggers a page swap which needs to execute three instructions (*AEX, ELDB/ELDU and ERESUME*). These instructions incur an overhead of around 10,000, 44,000 and 10,000 cycles respectively as shown by exiting studies [6], [7], thus leading to the overhead of around 64,000 cycles for each page swap which is massive. The typical page swap in untrusted memory takes around 2000 cycles [14] and thus for SGX, the overhead of page faults can be significantly huge if the application has a large memory footprint.

III. Related Work

We shall refer to Table I specifically in this section; it summarizes the related work in the chronological order. *HotCalls* [6] introduces a shared unencrypted memory between the enclave and the untrusted memory. All the data transfer happens through this shared memory using the polling-pinning method. This is done to prevent costly enclaves switches. On the similar lines, *SecSched* [15] outperforms Hotcalls by introducing a Cuckoo filter based working set similarity scheduling that reduces the core idleness of *Hotcalls*. However, both of these schemes target the overhead of enclave exits and entry. They do not look at the overheads of the page faults in SGX

Similarly, *Morphable Counter* [10] and *VAULT* [11] try to reduce the overhead of integrity verification by introducing new data structures to provide this security primitive at a lower cost, which is a part of the significant overhead of paging in Intel SGX.

In addition, the same problem has been discussed by Eleos [8]. To alleviate the severe overhead of page fault and swap the page content with untrusted memory space, a software page management system has been devised to govern the existence state of the pages in the EPC. Despite the usage of the same encryption technique, maintaining the same security guarantee with hardware-implemented instructions (such as EWB and ELDU/ELDB) is problematic, particularly at the micro-architecture level. The recent competing work [7] recently published uses page preloading technique for migrating pages from the untrusted memory to the secure memory. It basis its decision on the history of the page faults observed, not on the entire memory trace and also the decision is taken at the enclave memory. Moreover, it uses a source level code instrumentation to implement software-based page preloading. We instead decide about the prefetching of the pages at the last level cache only and we take the entire trace of the memory to see the access of the pages leading to a better accuracy in our proposed design.

Shortcomings of the prior work: The competing schemes which target the overheads of page faults in SGX either use application managed paging or the page preloading based on page faults observed along with the some source code instrumentation to implement software-based page preloading. The main aim of this paper is to reduce the overheads of the page faults in SGX without the use of software managed paging or the source code instrumentation as done by the competing schemes.

IV. Implementation

A. Overview

Intel SGX provides a limited physical memory of maximum 128 MB for applications to use. If the footprint of the application is greater than 128 MB, the data has to be fetched from the untrusted memory leading to the page faults. The overhead of a single page fault is typically around 64,000 cycles reported in [7]. To mitigate the performance overhead of page faults, we propose *SGXFault* that categorises the pages into *frequent* and *non-frequent* pages. We use a scheme called as Cache Lockdown to lock the blocks of the frequent pages and for the non-frequent pages, we use a prefetching based scheme that prefetches the pages from the non-secure to the secure memory by taking the decision at the cache level. These two techniques combined together ensure that the overhead of page faults is reduced.

TABLE I: Summary of Prior Work

Paper	Venue	Remarks
Eleos [8]	EuroSys'17	Reduces overhead of context switches and page faults both. Uses application-managed paging.
HotCalls [6]	ISCA'17	Reduces overhead of context switches only using unencrypted shared memory.
VAULT [11]	ASPLOS'18	Modifications of metadata structures to increase size of EPC.
Morphable Counters [10]	MICRO'19	Modifications of metadata structures to increase size of EPC
SecSched [15]	PACT'20	Reduces overhead of context switches only using flexible Cuckoo filter scheduling.
RegainingLostSeconds [7]	Middleware'20	Reduces page fault overhead using page preloading and source code level instrumentation.
SGXFault	—	Reduces page fault overhead using prefetching and cache lockdown.

B. Keeping Track of Frequent and Non-Frequent Pages

We maintain a 8-bit (chosen experimentally) saturating counter per block in a TLB. We use a 64 entry TLB in our design. In addition to this, we also maintain a per page 8-bit counter called as *PageCounter* which is the average of the block counters. This will provide us the information about the overall usage of the blocks in the page. Therefore, the additional space required is around 4KB per core. When the counters for any of the blocks overflow, we half the counters of all the blocks for all the pages to prevent ageing. These counters will ensure that we have the information of the most frequent and the non-frequent pages (and blocks also) in our application.

C. Cache lockdown for blocks of frequent pages

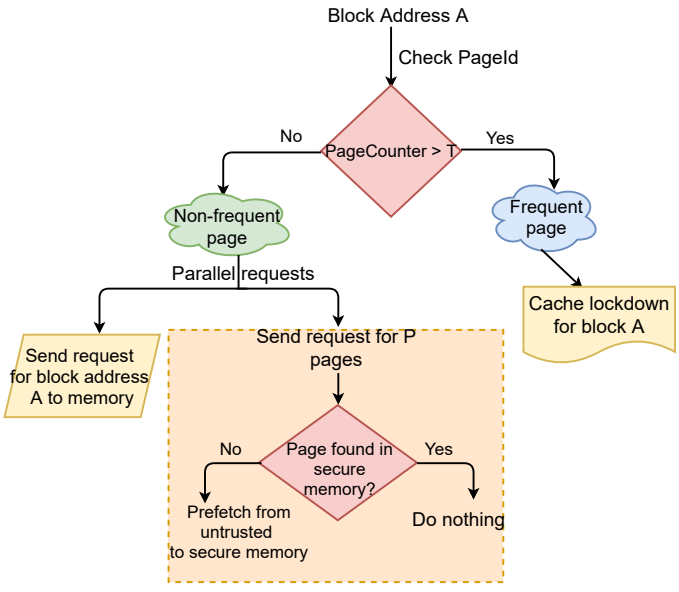

Fig. 1: Design of *SGXFault*

The main aim here is that for any given application, we want to ensure that the secure pages are always found in the EPC memory which will reduce the page faults. However, due to the limited size of EPC memory, this is not achievable. In such cases, the pages have to be fetched from the untrusted memory which will introduce significant performance overheads. We avoid this by separating the pages into two categories- *frequent* and *non-frequent*. Figure 1 shows the overall working of the *SGXFault* design. We use the per page *PageCounter* as described above to distinguish between the two categories. For each block, we get the corresponding pageId using the page offset and obtain the *PageCounter*. The pages with the count value greater than T are considered as the most frequent pages while the rest are considered non-frequent. The value of T is chosen experimentally to be 2^7. The blocks which belong to the frequent pages will be accessed frequently in the near future and thus we use a locking scheme called as *Cache Lockdown* [16], [17] for the lower level cache (LLC) to lock the blocks of the pages that are frequently used. This indirectly will ensure most frequent pages are locked in the cache and hence the request to these pages is not sent to the secure memory frequently. Note that we use a atomic instruction to lock a particular cache line in the cache. We want to mention here that we cannot lock all the entries of the cache (the frequent pages) as that will introduce performance degradation because the non-frequent pages will not get any space in the cache. To eradicate this issue, we use the Intel Allocation Technology [18]–[20] to partition the cache between the blocks of the frequent and the non-frequent pages. We allocate the cache based on the ratio of frequent and non-frequent pages. We update the cache partitioning periodically after an epoch of 100 million instruction (chosen experimentally). We use a single LSB bit called as *Lock* bit of the tag address to distinguish between the locked and the unlocked region of the LLC. If the *Lock* bit is 1, then it means that the block is locked else unlocked.

We want to mention here that we cannot have all the secure blocks of the pages in the caches together at the same time because the footprint of the application is huge. We need an intelligent eviction policy for the caches to solve this issue. We ingeniously solve this by using the *PageCounter* to our advantage. Whenever a request for the block is sent to the main memory, we check the corresponding pageId of the block and obtain the corresponding *PageCounter* from the TLB. Till the block is fetched from the main memory, we check the range of the *PageCounter* to see whether it belongs to the *frequent* or the *non-frequent* page category. If the block belongs to the frequent page category, we choose the locked cache portion and check the blocks that have the lowest value of page counters. From the candidate blocks which we obtained, we use the Least Recently Used (LRU) policy to

evict the existing block from the cache. In addition, we also update the corresponding count in the TLB for this block along with the *PageCounter*. If the block belongs to the non-frequent page category, we use the same LRU eviction policy and additionally,we use a prefetching scheme for these pages so that we already have these pages copied from the untrusted memory to the secure memory, thus avoiding page faults. We discuss the prefetching scheme in the next section.

D. Prefetching for blocks of non-frequent pages

For the *non-frequent* pages of the application, we maintain a *PageQueue* of 16 entries in the processor that keeps track of the pages accessed in the recent past. We maintain the queue for only the non-frequent pages of the application. We update the queue after an epoch of 1 million instructions everytime so that we don't prefetch inaccurately.

Whenever a block is missed in the last level cache, we check the corresponding page of the block and check the queue of the pages that are likely to be accessed after this page. Therefore, we send two requests to the main memory, the first request will be to fetch the missed block while as the second request will be to prefetch the pages that are expected to be accessed next in the near future. These two requests will be different in a sense that only the first request will be sent back to the processor. For the second request, the pages that are to be prefetched will be checked in the EPCM and if they are already in the EPC, nothing needs to be done. However, if the pages are not in the EPC, a page fault will be triggered and the control will be taken by the OS. The OS will start transferring the pages from the untrusted to the secure memory. In our page level prefetcher, we send the request for P number of pages. The value of P is determined experimentally and we found that $P = 8$ as the optimal value. Since we maintain a finite queue, there can be a possibility that the page to which the block belongs does not exist in the queue. In such cases, we do not prefetch, however we update the queue by this page. Our technique will ensure we have all our demanding pages ready in the secure memory, thus reducing the overhead of page faults.

E. Hardware Modifications

Our efficient page fault handling design *SGXFault* requires few hardware modifications (refer to Figure 2) that need to be incorporated. They are ❶ Maintaining a saturating counter per block in a TLB along with the *PageCounter*, ❷ a *PageQueue* to keep track of the pages accessed in the recent past.

V. Evaluation

A. Experimental Setup

For our simulations, we use a cycle-accurate architectural simulator Tejas [21] which has been rigorously validated with the native hardware. We use a bag of 7 real world workloads that have a working set in the range 2-5 GB with a regular and an irregular memory access and are extensively used in server cloud environments. Tejas uses the Qemu [22] tool to capture the effects of both application and OS level traces. To obtain

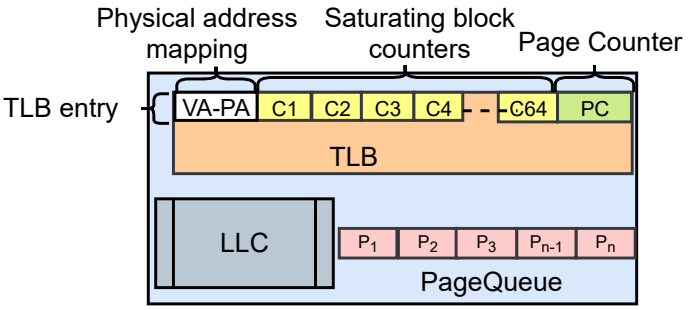

Fig. 2: Hardware Modifications for *SGXFault*

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
Shared L2 cache			
Latency	8 cycles	Block size	64
Associativity	8	Size	8MB
Main Memory Latency	200 cycles, 4 mem. controllers		

TABLE II: Simulation parameters

the latencies of different memory structures, we use the Cacti 6.0 [23] tool. The detailed architectural simulator parameters are mentioned in Table II. We simulate a 32-core chip with each core having a private data cache. We use a 8 MB shared L2 cache with directory based cache coherence. We have found empirically that an L3 cache did not improve our results given the fact that this necessitates cache coherence at the L2 level (similar observation made by Abellan et al. [24]).

B. Hardware Overheads

We obtain the area and the time overheads of our hardware modifications using the Cacti6.0 [23] tool as shown in Table III. The designs were modeled with a 14nm technology. We found the area overheads of *PageCounter* (along with saturating counter per block) and *PageQueue* to be around 0.07% and 0.00005% respectively which is minimal. Furthermore, to access these structures, the typical delay is nearly $\sim$1 cycle which is negligible.

Hardware Structure	Total Area Overhead
PageCounter	0.07%
PageQueue	0.00005%

TABLE III: Area overheads of different structures

C. Prefetching different number of pages

We now look at the performance of *SGXFault* by varying the number of pages P to be prefetched from the untrusted to the secure memory. We vary the value of P from 2 to 16 and

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3268169

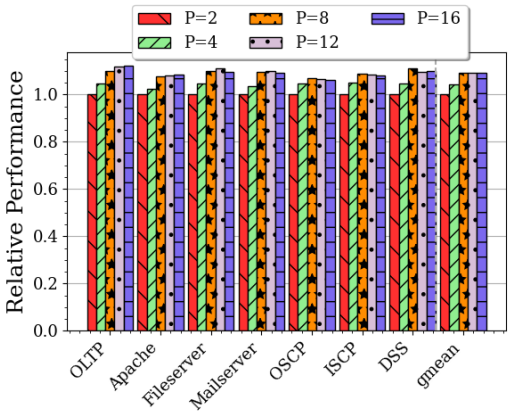

Fig. 3: Relative Performance (Ratio with respect to P=1 as baseline) of *SGXFault* when varying the number of pages to be prefetched from 2 to 16. Prefetching pages equal to 8 is an optimal value for our design.

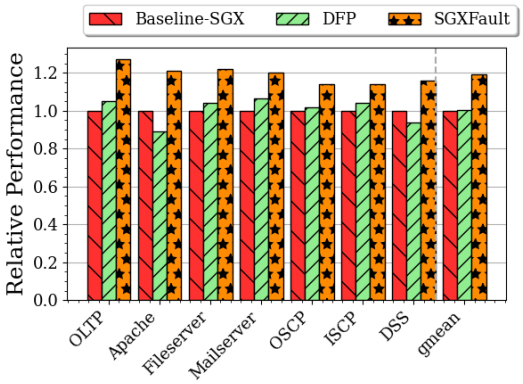

Fig. 4: Performance (Ratio with respect to *Baseline-SGX*) comparison of the different schemes. *SGXFault* performs 18.6% and 17.8% better than *Baseline-SGX* and *DFP* respectively. Lesser page faults in *SGXFault* lead to better performance.

we observe that on increasing the number of pages from 2 to 8, the performance of *SGXFault* increases by around 10.1%. This is because we are likely prefetching the correct pages that are to be accessed in the near future. However, on increasing the value of P from 8 to 16, we don't see much improvement in the performance. This is because increase in the number of pages leads to prefetching of the pages that are not likely to be accessed and thus reduces the accuracy. Thus, we choose a value of P=8 as the optimal value for our design.

D. Performance Comparison using SGXFault

In this section, we compare our *SGXFault* with the *Baseline-SGX* (which implements Intel SGX) and DFP (which implements Dynamic Fault History-Based Preloading) done by [7]. We compare the relative performance among the different schemes and we define *performance as the inverse of the execution time*. We simulate all our workloads for a warmup period of 500 million instructions and then with a simulation of 5 billion instructions.

We observe that *SGXFault* performs around 18.6% better than *Baseline-SGX*(refer to Figure 4). We explain this behaviour by looking at the Figure 5 that shows the relative number of page faults among the different schemes. We observe that the page faults in *Baseline-SGX* are 39.5% more compared to our proposed scheme. This is because we ensure that most of our frequent pages (or blocks) are in the cache only (using Cache lockdown) and additionally, we use a prefetcher that ensures that we have the secure pages ready in the EPC memory. Since the page faults are less in our scheme, the performance of our scheme is better compared to *Baseline-SGX*. Similarly, *SGXFault* performs better than *DFP* by around 17.8%. This is because for *SGXFault*, we take the memory access pattern of the whole application compared to taking only the page fault access pattern as done in *DFP*. Taking the entire memory access pattern leads to a better accuracy, thus less page faults (see Figure 5). In addition to this, we intelligently categorize the pages into frequent and non-frequent pages and keep the frequent pages in the caches for most of the execution leading to less main memory requests and subsequently lesser page faults. Note that we do not simulate the overheads of context switching in *Baseline-SGX* to evaluate the efficacy of our design.

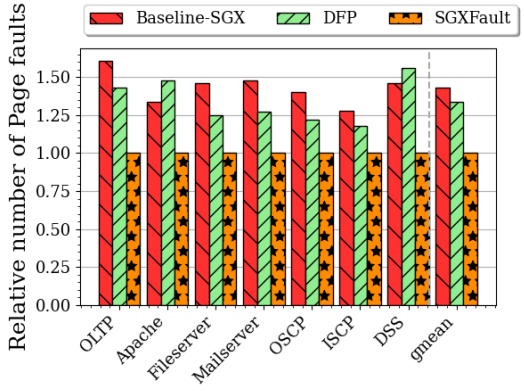

Fig. 5: Number of page faults (Ratio with respect to *SGXFault*) for all the schemes. Better caching and prefetching ensures page faults are the least in *SGXFault*.

E. Sensitivity Analysis

1) Performance for Varying Lower Level Cache sizes: We look at the performance of *SGXFault* by varying the size of the lower level cache. We vary the cache size from 2MB to 16MB and we observe that as we increase the LLC cache size, the performance improves significantly (refer to Figure 6). This is because more frequent pages can be locked in the larger space cache leading to less page faults in *SGXFault*, hence the increase in the relative performance. However, there is an additional power and area overheads if we keep the LLC size

to be higher. Therefore in our design, we keep the LLC size as 8MB.

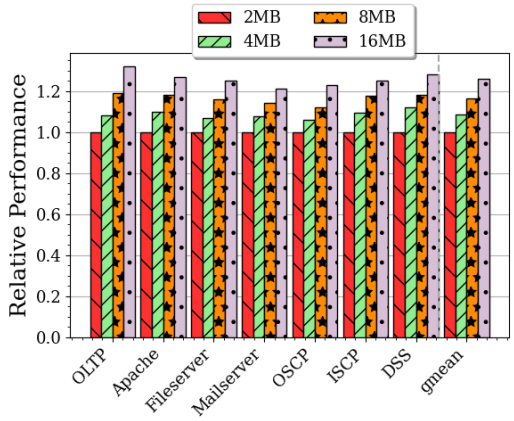

Fig. 6: Performance comparison (Ratio with respect to cache size as 2MB) for varying cache sizes. We choose LLC size as 8MB as it balances performance and area/power.

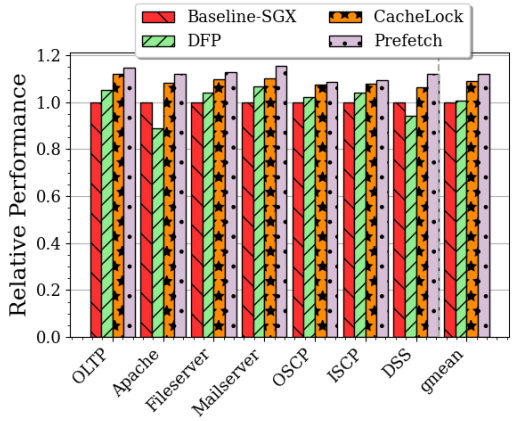

Fig. 7: Performance Comparison (Ratio with respect to *Baseline-SGX*) using Cache Lockdown and Prefetching individually. Both these techniques separately also outperform the state of the art schemes.

2) Relative Performance using Cache Lockdown and Prefetching individually: We now look at the performance of Cache lockdown and Prefetching individually compared to the baseline schemes. We observe that even separately, we outperform the baselines (refer to Figure 7). Cache lockdown performs 8.2% and 7.76% better than *Baseline-SGX* and *DFP* respectively. This is because we ensure that the blocks of the frequent pages remain in the LLC by locking the blocks in the cache. Similarly, Prefetching is able to perform better than *Baseline-SGX* and *DFP* by 10.8% and 10.46% respectively. In this case also, we are ensuring that the pages are fetched from the untrusted memory to the secure memory before they are actually accessed in the near future.

3) Performance for a bag of SGX and non-SGX applications: We now look at the performance of the overall system when both SGX and non-SGX applications are executed together. To evaluate this, we create different bags of applications where each bag consists of SGX and non-SGX applications combined in a 1:1 ratio with a load factor of 1X for a 32 core system (32 threads on 32 core system). Figure 8 shows the performance of *SGXFault* for SGX and non-SGX combined compared to *DFP* and a Baseline *SGX-NonSGX* system. We see that *SGXFault* outperforms both *SGX-NonSGX* and *DFP* system by around 12.8% and 10.6% respectively. We could observe that even in the presence of non-SGX applications, *SGXFault* outperforms the baseline schemes. This is because *SGXFault* uses two mechanisms to improve the performance. One is the Prefetching based scheme that SGX applications take full advantage of. The second is the Cache lockdown where the blocks of the frequent pages are locked. Both the SGX and the non-SGX applications take advantage of this locking as Cache lockdown will lock the blocks of the frequent pages, which is independent of what type (SGX or non-SGX) of application is running. We also show the performance of Non-SGX applications running with and without *SGXFault*. We observe that NonSGX applications with *SGXFault* (we call it as *NonSGXFault* in the Figure 8) has some decrease in the performance by around 4-5% because SGX applications (using *SGXFault*) use Cache Lockdown which affects the performance of other running applications. The standalone NonSGX application (red bars of the Figure 8) will have the highest performance since no secure hardwares are used for executing these applications.

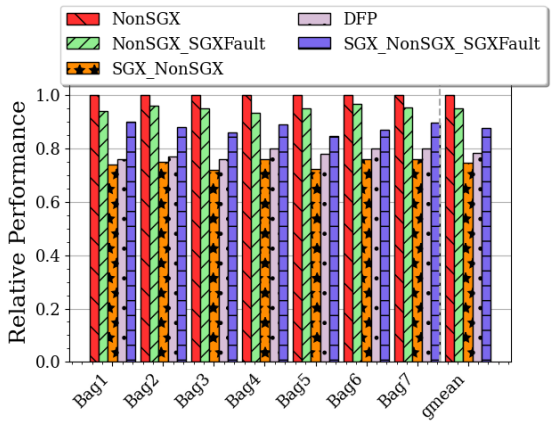

Fig. 8: Overall System Performance when running a bag of SGX and non-SGX applications together. The performance is relative with respect to Non-SGX implementation. Cache Lockdown along with prefetching technique makes *SGXFault* the best design among all.

VI. Conclusion

Intel SGX has significant overheads because of page faults for large memory footprint applications. In this work, we propose *SGXFault* that categorises the pages into frequent and non-frequent pages. For the blocks of the frequent pages, we use a Cache lockdown mechanism and for the rest, we use a page level prediction based prefetching. Using both the approaches, we are able to outperform the Intel SGX and recent competing scheme by around 18.6% and 17.8% respectively without any compromise in the security.

REFERENCES

[1] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[2] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[3] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pp. 21–30, 2008.

[4] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 1–13, 2018.

[5] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pp. 1–9, 2016.

[6] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 81–93, ACM, 2017.

[7] X. Liu, W. Wang, L. Wang, X. Gong, Z. Zhao, and P.-C. Yew, "Regaining lost seconds: Efficient page preloading for sgx enclaves," in *Proceedings of the 21st International Middleware Conference*, pp. 326–340, 2020.

[8] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 238–253, ACM, 2017.

[9] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, no. 6, pp. 54–62, 2016.

[10] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 416–427, IEEE, 2018.

[11] M. Taassori, A. Shafiee, and R. Balasubramonian, "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*, pp. 665–678, ACM, 2018.

[12] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pp. 123–134, IEEE, 2004.

[13] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, pp. 357–368, ACM, 2014.

[14] "Page fault cost." https://stackoverflow.com/questions/10223690/cost-of-a-page-fault-trap.

[15] O. Shafi and J. Bashir, "Secsched: Flexible scheduling in secure processors," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 229–240, 2020.

[16] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 45–54, IEEE, 2013.

[17] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 143–148, 2007.

[18] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *2017 26th international conference on parallel architectures and compilation techniques (pact)*, pp. 194–205, IEEE, 2017.

[19] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, 2018.

[20] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 211–222, IEEE, 2017.

[21] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, pp. 47–54, IEEE, 2015.

[22] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.

[24] J. L. Abellán, C. Chen, and A. Joshi, "Electro-photonic noc designs for kilocore systems," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 2, p. 24, 2017.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3268169

Omais Shafi Pandith received the B.Tech degree in Information Technology from the National Institute of Technology, Srinagar, Jammu and Kashmir, India. He is currently working as a Research Scholar in the Department of Computer Science and Engineering at Indian Institute of Technology, Delhi, India. His research interests span the areas of Computer Architecture, Hardware Security and Embedded Systems. He has published four papers in reputed International Conferences and journals. He was awarded the travel grant by ACM India IARCS for his paper in the DAC 2021. He is a student member of the ACM.