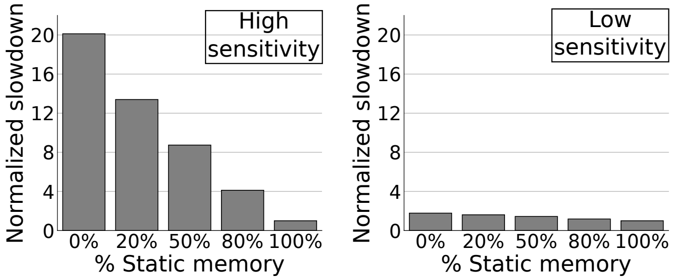
A *Tug-of-War* between Static and Dynamic Memory in Intel SGX

Sandeep Kumar[*#], Abhisek Panda[†#] , Advait Nerlikar[‡§] , and Smruti R. Sarangi[†]
[*]School of Information Technology, IIT Delhi, Delhi, India
[†]Computer Science and Engineering, IIT Delhi, Delhi, India
[‡]Electrical and Electronics Engineering, BITS Pilani, Goa, India
Email: sandeep.kumar@sit.iitd.ac.in, {abhisek.panda, srsarangi}@cse.iitd.ac.in, f20180282@goa.bits-pilani.ac.in

***Abstract*—Security of applications and data in a cloud setting has become a first-class design criterion. Hardware vendors have proposed *trusted execution environments* or TEEs where the hardware guarantees an application's data and code security both at rest and in-use, even from privileged entities such as operating systems and hypervisors. *Software Guard eXtension*, or SGX, is a popular, trusted execution environment or TEE solution from Intel. To ensure security guarantees, SGX provides secure sandbox environments called enclaves, which have encrypted physical memory. In the latest version of SGX, we start an enclave with a the specified amount of "static" memory specified by a developer. Subsequently, we add additional memory pages "dynamically" to an enclave depending on an application's memory usage. In this paper, we analyze the impact of the allocation and freeing of static and dynamic pages on an application's performance. We observe that inappropriately setting the static memory size may lead to a performance slowdown of up to $20\times$.**

We present *Harmony* – a profile-guided optimizer that measures the impact of dynamic memory management on an application's performance, and suggests a near-optimal distribution for static and dynamic memory pages. We show that Harmony improves the execution latency of an application by up to 68% and 29% when compared with the purely dynamic and purely static allocation schemes, respectively.

***Index Terms*—SGX, secure memory, performance**



(a) Application 1

(b) Application 2

Fig. 1: Different applications show different sensitivity to performance on the proportion of statically allocated pages. Application 1, which often allocates and deallocates memory, is significantly more sensitive to the number of static pages than Application 2, which allocates and deallocates memory only once. The performance slowdown is normalized w.r.t. their respective programs when executed with 100% statically allocated pages.

I. Introduction

Intel SGX [1] (Software Guard Extensions) is a popular hardware-based trusted execution environment (TEE) that enables the secure execution of applications on untrusted cloud servers across multiple domains such as file systems, key-value stores, AI/ML applications and blockchains [2, 3, 4, 5]. To ensure the secure execution of an application, SGX provides secure sandbox environments called *enclaves*, which have encrypted physical memory. The SGX hardware ensures the authenticity, confidentiality, integrity, and freshness properties of the code and data stored in the enclaves. Due to these protections, untrusted entities, even with root-level access, such as the operating system, hypervisor, and server administrators, cannot compromise the security of an application executing in an enclave [6].

To ensure security, SGX imposes the following restrictions on enclaves (at the software level): prohibiting system calls and privileged operations within an enclave, requiring predefined entry and exit points for an enclave , and prohibiting access to an isolated memory region that stores metadata – the enclave page cache map (EPCM). In the earlier version of SGX (SGXv1), an application had to specify the maximum amount of secure memory it needed before execution. Subsequently, SGXv1 allocated the specified amount of memory to an enclave at the time of its initialization. Any attempt by the application to allocate memory beyond the specified bound resulted in an *out of memory* (`OOM`) error. Estimating the memory requirements of a modern application is not a trivial task, and developers typically over-provision an enclave's memory to avoid such an `OOM` error.

To reduce the wastage in an enclave's memory, the latest version of SGX (SGXv2) introduces *enclave dynamic memory management* or EDMM. EDMM enables enclave initialization with minimum memory ($\geq 4KB$) and allows dynamic addition of memory pages to an enclave [7]. Data pages are now divided into two categories: *static* (allocated during the initialization phase) and *dynamic* (allocated *on-demand* while the application is executing).

[#]Equal contribution
[§]Work done during an internship at IITD

Static pages, like in SGXv1, are added to an enclave using `EADD` instructions and are associated with the enclave for its entire lifetime. Dynamic memory management is similar to classical demand paging, albeit with significant overheads due to security constraints. A secure page addition requires one enclave exit causing a TLB flush due to a transition to unsecure mode, two context switches, and one `ECALL` [8]. During the page addition, an inverse page table lookup needs to be done to ensure that the newly allocated secure page is not mapped to any other process, and metadata entries need to be made in the EPCM. Subsequently, dynamic pages are added using the `EAUG` instruction [7].

As shown in Figure 1, based on the memory requirements, an application can suffer a performance overhead of up to 20× if the amount of static memory is not configured properly (see Figure 1a). However, certain applications show similar performance irrespective of the static and dynamic memory distribution (see Figure 1b). In this paper, we thoroughly analyze how SGX manages static and dynamic pages: the allocation/deallocation path, the specific allocation mechanisms and their associated latencies. We then study the effect of varying the static memory size on an application's performance and the enclave initialization latency. Essentially, we answer the following question, *"Why are certain applications highly sensitive to the amount of static memory, and why are others not affected by it?"*. Based on our observations, we introduce the *impact score* metric that determines the relationship between the memory access pattern and the application's performance (lower the better).

Finally, we propose *Harmony* – a profile-based optimizer that configures the system correctly with the right number of static pages to balance the initialization latency and overall performance impact. We show that Harmony improves the execution latency of an application by up to 68% and 29% when compared with an all-dynamic and all-static setting, respectively. To the best of our knowledge, this is the first work that solves such a problem, which has serious practical consequences. Our specific contributions are as follows:

1) We perform a deep-dive analysis of the allocation process of static and dynamic memory pages. We specifically study their methods of allocation, their lifespan and associated latencies in Intel SGX using microbenchmarks.
2) We show that an ill-configured system can incur a performance overhead of up to 20×. We present an analysis of what makes an application sensitive or immune to the extent of enclave dynamic memory allocation (EDMM).
3) Next, we propose a novel profile-based optimizer that provides the best allocation of static pages, such that the impact of a given dynamic memory allocation policy on the application's performance.

II. Background

A. A Primer on Intel SGX

Intel SGX guarantees the authenticity, confidentiality, integrity and freshness of the code and data stored within it. The application is executed within a sandbox. Due to its security constraints, an enclave cannot directly use OS services. To do so, it must make an *outside call* or `OCALL`. Similarly, a normal application needs to execute an *enclave call* or `ECALL` to access a function within an enclave.

TABLE I: Description of microbenchmarks. We perform sequential read and write operations on all memory regions in each `ECALL`. *Fixed memory:* Allocated at the beginning and freed at the end. *Scratchpad memory [9]:* Allocated and freed on every `ECALL`

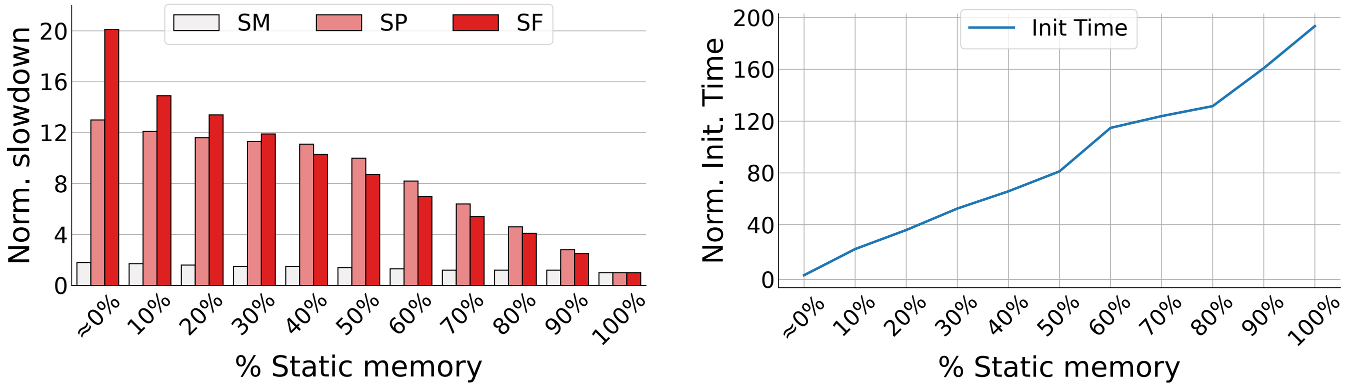
Benchmark	**Description**	**Fixed & Scratchpad**
SGXMain (SM)	Allocates a fixed memory region.	1 GB & 0 GB
SGXPartial (SP)	Allocates two regions: fixed and scratchpad – a hybrid memory organization.	500 MB & 500 MB
SGXFree (SF)	Allocates a scratchpad memory region.	0 GB & 1 GB

Memory Management : At boot time, SGX reserves a portion of the physical address space to be used as secure memory. This reserved region is called the processor-reserved memory (PRM), and the region used by applications is referred to as the *Enclave Page Cache* or EPC [6]. SGX allows applications with a size larger than the EPC size by transparently evicting pages from the EPC [6]. In SGXv1, the developer of a secure application is required to estimate the total memory requirements of the application upfront. The SGX hardware statically "commits" this memory to the enclave using the `EADD` instruction [7] during initialization.

SGXv2 introduced the concept of *static* and *dynamic* memory regions in an enclave's address space using *enclave dynamic memory management* (EDMM). A static region represents an area that has physical EPC pages backing it. The dynamic region indicates that no physical EPC pages have been assigned to it during initialization. They will be assigned later on demand. The application developer can specify the initial *static* enclave memory size that needs to be provided to the enclave at initialization time (`HeapInitSize`) along with a parameter specifying the maximum memory the enclave can use (`HeapMaxSize`).

Dynamic Memory Allocation: SGXv2 dynamically allocates memory to an enclave on a *fault* when the enclave attempts to access a page that is yet to be allocated (for example, right after a `malloc()` call). The SGX driver allocates a page using the `EAUG` instruction and later adds it to the enclave using the `EACCEPT` instruction [7, 10]. In SGXv2, an enclave can be configured to execute with just a single 4 KB page for the heap (SGX may add a few additional pages for its metadata). It is important to note that the OS still manages the page table; however, all updates to the TLB need to be vetted by the SGX subsystem, which uses an inverted page table (stored in the enclave page cache map or EPCM) to verify that there are no security violations [6].

EPC Evictions: If an application's working set crosses the EPC limit, SGX transparently handles the eviction and loading



(a) Performance slowdown (normalized to 100% static memory.)

(b) Enclave initialization time (normalized to 0% static memory.)

Fig. 2: (a) Slowdown of microbenchmarks (also see Table I) and (b) enclave initialization time as a function of ratio of the static memory size to the maximum memory requirement.

TABLE II: Table showing number of EADD, EAUG, and EREMOVE operations for microbenchmarks (see Table I).

Bench	Inst.	≈0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
SM	EADD	0K	26K	51K	77K	103K	128K	154K	179K	205K	231K	256K
SM	EAUG	256K	230K	205K	179K	154K	128K	102K	77K	51K	26K	0K
SM	EREMOVE	512K	487K	461K	436K	410K	384K	359K	333K	308K	282K	256K
SP	EADD	0K	26K	51K	77K	103K	128K	154K	179K	205K	231K	256K
SP	EAUG	384K	358K	333K	307K	282K	256K	205K	154K	102K	51K	0K
SP	EREMOVE	768K	743K	717K	692K	666K	640K	564K	487K	410K	333K	256K
SF	EADD	0K	26K	51K	77K	103K	128K	154K	179K	205K	231K	256K
SF	EAUG	512K	461K	410K	358K	307K	256K	205K	154K	102K	51K	0K
SF	EREMOVE	1M	947K	871K	794K	717K	640K	564K	487K	410K	333K	256K

of pages from/to the EPC while ensuring the confidentiality and integrity of the page. The EWB instruction is used to evict a page from the EPC. The instruction takes the EPC page as the input and generates an encrypted version of the page (along with an 8-byte nonce). It also generates the corresponding MAC (message authentication code). The encrypted page and the MAC are written to unsecure memory, and the nonce is stored in a VA slot (page within the EPCM). Encryption ensures confidentiality, the MAC ensures integrity, and the nonce ensures the freshness of the evicted page. Note that cold boot and side-channel attacks are not in the scope of SGX [7].

III. Motivation

A. Static and Dynamic Memory

As discussed before, data pages can be added to an enclave either at the time of initialization (static) or on a page fault (dynamic). At first glance, starting an enclave with the minimum amount of memory and letting the SGX memory system handle the page allocations seems like the best possible choice. This is akin to the behavior in operating systems like the latest version of Linux. However, as seen in our experiments (see Figure 1 and Figure 2a), the performance of certain applications is extremely sensitive to the number of pages allocated statically (static:dynamic or SD ratio).

Performance analysis: To analyze the effect of the initially committed memory on the performance, we execute a set of micro-benchmarks (see Table I) with different amounts of static memory. As shown in Figure 2, different applications show different behaviors when the statically allocated memory size increases.

SGXMain, which allocates a fixed amount of memory upfront and then performs read and write operations on it, is the baseline. As a matter of fact, if there are no free calls, the sum of the number of EADD and EAUG instructions will always remain the same ($256K \times 4KB = 1GB$). In the figure, the performance of SGXMain shows a slight variation due to the additional cost associated with allocating a dynamic page. Note that the cost of adding a single 4 KB page dynamically is higher that if it is added statically due to additional SGX operations associated with the former: an asynchronous enclave exit, two context switches between user and kernel mode, and one ECALL [8].

For SGXPartial, we allocate half of the total memory as the fixed memory region, while the remaining half serves as the *scratchpad* memory region (refer Table I). We observe that as the enclave's static memory allocation increases, the application's performance slowdown decreases. This is because a higher static memory allocation reduces the frequency of EAUG and EREMOVE instructions, which are required for

TABLE III: Latency of key SGX operations

Instruction	Description	Latency
EADD (add_page)	Adds an EPC page to a pre-initialized enclave	5.52 μs
EAUG (encl_augment)	Adds an EPC page to a running enclave	5.70 μs
EREMOVE (free_page)	Frees an EPC page	1.5 μs

dynamic memory operations. This improvement is associated with a rise in the enclave initialization latency (see Figure 2b). Notably, the performance improvement in SGXPartial is significantly more evident as static allocation increases from 50% to 60%. There is an 18% decrease in the slowdown.

We observe a similar trend with SGXFree, where we allocate the entire memory in the scratchpad memory region.

Observation: The amount of statically allocated memory should be decided based on an application's memory usage characteristics.

B. Allocation Latencies

Here, we measure the latency of the key operations in the SGX memory management subsystem. We use the standard *ftrace* [11] utility to measure the latencies of functions in the SGX driver responsible for allocation and freeing of secure memory. The latencies reported are the geometric mean of 10 K readings. As shown in Table III, the functions executing `EADD` and `EAUG` have roughly the same latencies.

Observation: A *free_page()* (`EREMOVE`) operation in SGX is a much faster operation as compared to adding a page.

IV. Design and Implementation

In this section, we discuss the design and implementation of Harmony. Harmony aims to optimize the initial enclave size of an application such that we find a balance between the initialization time and the overall performance of an enclave.

A. Overview

Figure 3 shows a high-level design of Harmony, which primarily consists of three components: ❶ a profiler, ❷ an evaluator, and ❸ an optimizer. The profiler collects execution-related statistics, which are then fed to the evaluator to determine the impact of memory allocations on the execution of the workload. The optimizer takes this information and computes the ratio of the static and maximum memory requirement for the application.

B. Profiler

In SGXv2, the memory allocation and deallocation pattern of a secure application impacts its performance (discussed in Section III). The profiler is responsible for collecting this pattern for every application.

The profiler executes an application by setting the initial enclave size (static memory) to a single 4 KB page. It then records the `alloc()` and `free()` calls along with the respective values of the "size" argument using the `strace` [12] utility. Additionally, it uses an instrumented version of the SGX driver that captures calls made to `sgx_encl_add_page()`, `sgx_encl_augment()`, and `sgx_encl_free_page()` functions. After the completion of the execution, it finds the number `EADD`, `EAUG`, and `EREMOVE` instructions by processing the logs.

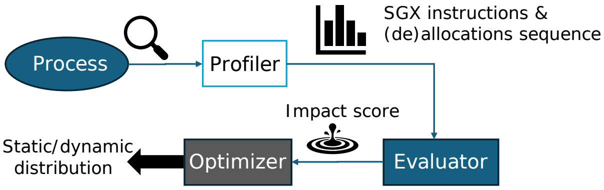


Fig. 3: A high-level design of Harmony.

C. Evaluator

The evaluator takes the profile information from the profiler, and determines the relationship between the memory access pattern and the slowdown of an application. To determine this relationship, we formulate a novel score function that analyzes a given memory access pattern and provides an Impact Score. The Impact Score $\in [0, 1)$, where 0 indicates no impact and a higher value indicates greater impact.

Intuition: As seen in Section III, application performance is negatively impacted due to `EAUG` calls only if it makes an allocation request after a free request. Assume that an application allocation and freeing sequence is as follows: `alloc(100MB)`, `free(50MB)`, `alloc(100MB)`, `free(150 MB)`. Here, the second allocation request is doing an additional 50 MB of allocation due to the `free(50MB)` request made before. The last `free(150 MB)` has no impact on the performance as there are no `alloc()` requests after that. Hence, we calculate two values:

- $alloc_{tot}$: Total allocation requests made by the application. (200 MB in the above example)
- $free_{tot}$: Total free requests made by the application followed by at least one allocation request. (50 MB in the above example). Note that the allocation at the end is mandatory, otherwise $free_{tot} = 0$.

Impact Score: We then define the Impact Score (IS) as follows:

$$IS = 1 - \left(\frac{alloc_{tot} - free_{tot}}{alloc_{tot}} \right) \qquad (1)$$

For the example above, Impact Score will be $1 - (\frac{150}{200}) = 0.25$. Intuitively, Impact Score indicates that 25% of 200 MB = 50 MB should be allocated as static memory. Based on the impact score of an application, the developer decides whether to reconfigure the static memory size of an application using the optimizer. *Note that $free_{tot} < alloc_{tot}$ because of a mandatory allocation after $free_{tot}$ is computed.*

D. Optimizer

Static Page Distribution: Let mem_{max} represent the maximum memory requirement of an application, and let β represent the ratio of the size of static memory to mem_{max}; $\beta = 1$ implies that all the pages are static, whereas $\beta \approx 0$ denotes that the number of static pages is set to one 4KB page. Note that in addition to the memory pages allocated for an application, SGX allocates around 600 KB of heap pages for its internal metadata, irrespective of the value of β.

Overhead Analysis: In Section III, we observe that an application suffers from a performance slowdown for low values of β and an increase in the enclave's initialization latency for high values of β. The overheads associated with the enclave's initialization ($init\_ovh$) and dynamic memory management ($exec\_ovh$) contribute to the total overhead of an application. Thus, we can formulate the total overhead (ovh) of an application as follows:

$$ovh = init\_ovh + exec\_ovh \tag{2}$$

All of them are a function of β.

Initialization Overhead: An application experiences minimal initialization time when the enclave is initiated with a single 4 KB static page ($\beta \approx 0$); conversely, the initialization time gets prolonged when all of the required pages are marked as static ($\beta = 1$). This occurs because a higher value of β leads to an increase in the number of `EADD` invocations, resulting in a higher initialization latency for the application's enclave. Therefore, we can measure the initialization overhead ($init\_ovh$) due to static page allocation as follows:

$$init\_ovh = \#EADD(\beta) \tag{3}$$

Note that the number of `EADD` instructions is a function of β.

Execution Overhead: An application suffers from a performance slowdown due to the additional `EAUG` and `EREMOVE` instructions. If we increase β, we observe a decrease in the frequency of `EAUG` and `EREMOVE` calls (refer to Table II). For instance, if we increase the static memory by 50% across all microbenchmarks, the number of `EAUG` and `EREMOVE` calls reduces by 20-50%. Therefore, we can formulate the execution overhead ($exec\_ovh$) using the following equation:

$$exec\_ovh = \#EAUG(\beta) + 0.3 \times \#EREMOVE(\beta) \tag{4}$$

The factor of 0.3 is derived from the latency of `EAUG` and `EREMOVE` (see Table III). Finally, the total overhead of an application is as follows:

$$ovh = \#EADD(\beta)) + \#EAUG(\beta) + 0.3 \times \#EREMOVE(\beta) \tag{5}$$

The optimizer collects the profiling logs of an application from the profiler, which consists of the number of `EADD`, `EAUG`, and `EREMOVE` calls for different values of β. Subsequently, it infers the relationship between the SGX statistics and β based on the memory access patterns and characteristics of the application. Finally, it calculates a suitable value of β that minimizes the value of ovh using Equation 5.

TABLE IV: Description of the workloads (adapted from SGXGauge [14]) and their respective settings.

Workloads	**Description**	**Input/Setting**
BFS	Traverse graphs generated by web crawlers. Use breadth-first search.	Nodes 150 K Edges 1.9 M
B-Tree	Create a B-Tree and perform lookup operations on it.	Elements: 16 M
HashJoin	Probe a hash-table (used to implement equijoin in DBs)	Look ups: 20 M
OpenSSL	Encryption-decryption library.	Size: 1.1 GB
PageRank	Assign ranks to pages based on popularity (used by search engines).	Nodes 5000 Edges 12.5 M
SVM	Popular ML algorithm (application: text and hypertext categorization)	Rows 10000 Features 128

TABLE V: The execution latency of workloads under the following settings: all-static (A-STAT), all-dynamic (A-DYN) and Harmony (HMY).

	Setting	Static Mem (MB)	β	Init Time (ms)	Total Time (s)	vs A-DYN	vs A-STAT
BFS	A-DYN	0	–	28	17	0.00%	18.24%
	A-STAT	2,144	–	3,712	20	-22.30%	0.00%
	HYM	1,073	0.5	1,948	15	9.76%	26.21%
BTree	A-DYN	0	–	26	37	0.00%	-6.60%
	A-STAT	1,077	–	1,755	35	6.19%	0.00%
	HYM	862	0.8	1,450	35	4.96%	-1.31%
HashJoin	A-DYN	0	–	26	82	0.00%	-1.82%
	A-STAT	1,227	–	2,463	80	1.79%	0.00%
	HYM	982	0.8	160	78	4.23%	2.48%
OpenSSL	A-DYN	0	–	46	61	0.00%	-122%
	A-STAT	2,155	–	2,882	28	54.97%	0.00%
	HYM	435	0.2	3,784	20	67.97%	28.88%
SVM	A-DYN	0	–	26	26	0.00%	-6.05%
	A-STAT	688	–	1,116	25	5.70%	0.00%
	HYM	140	0.2	256	24	10.42%	5.00%

V. Evaluation

A. Experimental Setup

We use a 2-socket system with two Intel Xeon Gold 6534 CPUs (72 cores), 1 TB of memory, 512 GB of SSD storage, 1 TB HDD, running Linux kernel 5.15. The systems are connected via a 1 Gbps network. The system support scalable SGX with an EPC up to 64 GB. We use Intel SGX SDX v2.13 for our SGX setup. The workloads used for the experiments are shown in Table IV. We use *ftrace* [11] to measure the frequency and time taken by the functions in the Intel SGX driver. To measure the performance-related hardware counters, we use the popular *perf* tool [13].

B. Results

Harmony improves the performance of an enclave by 19.47% and 12.25% compared to all-dynamic (set the static

TABLE VI: SGX statistics for the following settings: all-static (A-STAT), all-dynamic (A-DYN) and Harmony (HMY).

	Setting	EADD	EAUG	EREMOVE
BFS	A-DYN	0.7K	547.9K	1.1M
	A-STAT	548.8K	0.0K	549.9K
	HMY	274.7K	274.0K	823.2K
BTree	A-DYN	0.7K	262.7K	526.0K
	A-STAT	275.7K	0.0K	276.2K
	HMY	220.6K	42.8K	306.6K
Hashjoin	A-DYN	0.7K	313.1K	626.9K
	A-STAT	314.1K	0.0K	314.7K
	HMY	251.3K	62.5K	376.8K
OpenSSL	A-DYN	1.4K	1.6M	3.3M
	A-STAT	551.8K	0.0K	552.9K
	HMY	111.4K	1.3M	2.7M
SVM	A-DYN	0.7K	174.0K	348.8K
	A-STAT	176.1K	0.0K	176.4K
	HMY	35.8K	139.0K	313.8K

memory size to a 4 KB page) and all-static (set the static memory size to the maximum memory requirement) configurations, respectively. Table V shows the distribution of pages (β) for different workloads. The selected distribution achieves a balance between the initialization time and overall execution speed of the workload. Table VI shows the corresponding instructions executed by SGX.

VI. Related Work

Apart from challenges in porting an application to Intel SGX [15], the adoption of Intel SGX is hampered by the performance overheads associated with executing an application. As shown by prior work and also seen in our experiments, application execution in SGX can incur a slowdown of up to 20×. This is due to the high cost of enclave transitions to make system calls and costly EPC faults. Researchers have proposed many different solutions to mitigate the performance overheads associated with Intel SGX. However, they have not studied the impact of the ratio of static to dynamic memory allocation on an application's performance degradation.

Liu et al. [16] propose a page pre-fetching mechanism for bringing a page from the unsecure memory to the secure memory based on an application's memory access pattern. Dinh Ngoc et al. [17] argues for a dynamic page table switching mechanisms from nested paging [18] to shadow paging [19] in a virtual setting based on the application behavior. Weisse et al. [20] propose a better mechanism to interact with the operating system by leveraging an idle core in a multi-core system.

Weisse et al. [20] propose a mechanism to mitigate the cost of frequent transitions from the secure to unsecure worlds to make a system call. An enclave that wants to make a system call writes the system call details including its arguments in the untrusted memory. A proxy thread, running on a separate core, reads the arguments and performs the system call on behalf of the enclave. Results are written back to the untrusted memory from where it can be read by the enclave.

VII. Conclusion

This paper demonstrates the critical role of an enclave's static memory size on an application's performance. The performance slowdown arises from the frequent invocation of EAUG and EREMOVE instructions. Increasing static memory reduces these slowdowns but raises the enclave initialization latency. To address this trade-off, we propose Harmony, a profile-based optimizer that determines the optimal static memory allocation to balance the enclave initialization latency and performance impact.

References

[1] Intel, "Intel software guard extensions — intel software," https://software.intel.com/en-us/sgx, 2019, (Accessed on 12/14/2019).

[2] G. Ayoade, V. Karande, L. Khan, and K. Hamlen, "Decentralized IoT Data Management Using BlockChain and Trusted Execution Environment," in *IRI'18*.

[3] L. Chen, J. Li, R. Ma, H. Guan, and H.-A. Jacobsen, "EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX," in *Middleware '19*.

[4] Y. Wang, L. Liu, C. Su, J. Ma, L. Wang, Y. Yang, Y. Shen, G. Li, T. Zhang, and X. Dong, "CryptSQLite: Protecting Data Confidentiality of SQLite with Intel SGX," *NaNA'17*.

[5] S. Kumar and S. R. Sarangi, "SecureFS: A Secure File System for Intel SGX," in *RAID '21*.

[6] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[7] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Intel® software guard extensions (intel® sgx) software support for dynamic memory allocation inside an enclave," in *HASP*, 2016.

[8] V. Dhanraj, "Fosdem 2023 - gramine library os," https://archive.fosdem.org/2023/schedule/event/cc_online_gramine/, 2 2023, (Accessed on 10/29/2024).

[9] L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *PACT'05*.

[10] 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 *HASP*, 2016.

[11] M. Gebai and M. R. Dagenais, "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead," *ACM Comput. Surv.*, 2018.

[12] Wikipedia contributors, "Strace — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Strace&oldid=922720825, 2019, [Online; accessed 17-November-2019].

[13] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves," in *Middleware*, 2018.

[14] S. Kumar, A. Panda, and S. R. Sarangi, "Sgxgauge: A comprehensive benchmark suite for intel sgx," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 135–137.

[15] A. Hasan, R. Riley, and D. Ponomarev, "Port or Shim? Stress Testing Application Performance on Intel sgx," in *IISWC 2020*.

[16] X. Liu, W. Wang, L. Wang, X. Gong, Z. Zhao, and P.-C. Yew, "Regaining Lost Seconds: Efficient Page Preloading for SGX Enclaves," in *Middleware '20*.

[17] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything You Should Know About Intel SGX Performance on Virtualized Systems," *Proc. ACM Meas. Anal. Comput. Syst.*, 2019.

[18] Oracle, "3.7. nested paging and vpids," https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/nestedpaging.html, (Accessed on 09/11/2021).

[19] K. Nahrstedt, "Cs 423 – operating systems design lecture 4 – processes and threads," https://courses.engr.illinois.edu/cs423/fa2011/lectures/lect35-virt2.pdf, 2011, (Accessed on 09/11/2021).

[20] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *ISCA '17*.