FastSGX: A Message-Passing Based Runtime for SGX

Subashiny Tanigassalame, Yohan Pipereau, Adam Chader, Jana Toljaga, Gaël Thomas

▶ To cite this version:

Subashiny Tanigassalame, Yohan Pipereau, Adam Chader, Jana Toljaga, Gaël Thomas. FastSGX: A Message-Passing Based Runtime for SGX. AINA 2024 - 38th International Conference on Advanced Information Networking and Applications, Apr 2024, Kitakyushu, Japan. pp.74-85, 10.1007/978-3-031-57916-5_7 . hal-04895325

HAL Id: hal-04895325

https://inria.hal.science/hal-04895325v1

Submitted on 17 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

FastSGX: a message-passing based runtime for SGX

Subashiny Tanigassalame[1], Yohan Pipereau[1], Adam Chader[1], Jana Toljaga[1], and Gaël Thomas[2]

Abstract Designing an efficient privacy-preserving application with Intel SGX is difficult. The problem comes from the prohibitive cost of switching the processor from the non-secure mode to the secure mode. To avoid this cost, we propose to design an SGX application as a distributed system with worker threads that communicate by exchanging messages. We implemented FastSGX, a runtime that exposes this programming model to the developer, and evaluated it with several data structures. Our evaluation with different workloads shows that the applications designed with FastSGX consistently outperform the equivalent applications designed with the software development kit provided by Intel to use SGX.

1 Introduction

Today, citizens deploy their sensitive data in cloud infrastructures. However, a cloud infrastructure is an untrusted system. A cloud infrastructure is shared among many users, which makes it an especially interesting target for an attacker. A cloud provider may also be honest but curious. Protecting user data when it is processed in a cloud infrastructure is thus today paramount to protect the privacy of the citizens.

In order to help users protect their personal data, Intel proposes a Trusted Execution Environment (TEE) named Intel SGX [6]. A TEE is a secure computation mode provided by a processor. A TEE is able to protect a memory zone, which is named an *enclave*, against an attacker, who fully controls the operating system, the hypervisor, and even the hardware. For that, a TEE relies on cryptography to enforce the confidentiality, integrity, and authenticity of an enclave.

Because of the cost of entering or leaving an enclave, designing an efficient application for Intel SGX is difficult. This cost is prohibitive: while a standard call costs only a few cycles, entering or leaving an enclave costs 7000 cycles [18, 27, 28].

[1]Telecom SudParis - IP Paris, France, e-mail: `first.last@telecom-sudparis.eu`
[2]Inria, France, e-mail: `first.last@inria.fr`

Several research works show that we can avoid this cost by using *switchless calls* [1, 24, 28, 33, 34]. A switchless call consists of leveraging *worker threads* in order to avoid switching the processor from the non-secure mode to the secure mode. In detail, each worker thread runs in a *security domain*: either the non-secure domain or in an enclave. In order to perform a call from one domain to another, a worker thread of one domain sends a message to a worker thread in the other. To send this message, the worker thread simply writes a value in a shared memory zone named an activation zone. Transferring the control from one domain to another costs a single cache miss: the cache miss that loads the activation zone from the core of the sender thread in the core of the receiver thread. Since transferring a cache line costs a few hundred cycles instead of several thousand, a switchless call significantly improves performance compared to switching the processor mode.

Whereas switchless call is a well-known technique, using this technique efficiently remains challenging for three reasons.

The first issue comes from the fact that the worker threads are hidden to the developer and configured statically. In detail, the SGX runtime provided by Intel can internally use worker threads to optimize the time to transfer the control from/to an enclave. These workers threads consume CPU resources when they idle because they actively spin on the activation zone. Unfortunately, the developer can only statically configure the number of worker threads, which is inadequate if the workload evolves over time. In such a case, the developer either over-provisions the number of worker threads, which wastes CPU resources when the workload is low, or under-provisions the number of worker threads, which leads to inefficiencies when the workload increases [33].

The second issue comes from the function semantic exposed by the SGX runtime provided by Intel. With a call semantic, the caller is suspended during a call. The caller uselessly wastes a CPU while actively waiting for the termination of the call. The developer can thus not use the wasted CPU resource to execute useful code in parallel.

The third issue comes from the design of the current SGX runtimes, which prevent a direct call from one enclave to another. Because of this design, while a worker thread in an enclave could easily transfer directly the control to a worker thread in another enclave, a worker thread has first to transfer the control back to a thread in non-secure mode in order to transfer the control to another enclave.

In this paper, we propose to use the switchless call technique more efficiently with a new programming model. In detail, we propose to explicitly design an SGX application as a distributed system with worker threads that communicate by exchanging messages. With our programming model, as with switchless calls, each worker thread runs in its security domain. However, the worker threads are made visible to the developer, who can create and destroy worker threads on the fly. The developer can thus adapt the number of worker threads to the workload on the fly. Moreover, instead of exposing a function-call abstraction, we propose to expose a message-passing abstraction. Thanks to this modification, a worker thread can continue its execution while another worker thread proceeds a message, which avoids wasting the CPU of the sender during a call. Finally, by exposing to the developer

an interface to send and receive messages, a developer can send a message from any worker threads to any other worker threads, which avoids the need to uselessly transfer the control to a worker thread in the non-secure domain in the case of an inter-enclave call.

We implemented our message-passing programming model in a new SGX runtime named FastSGX. We evaluated FastSGX with two classical data structures: a hashmap and a treemap. We evaluated versions of these data structures with one and two enclaves. Our evaluation with different access patterns shows that:

- The interface of FastSGX, with its 4 main functions, is simple enough to be usable in practice,
- FastSGX can be used to design efficient multi-enclave applications, while current switchless call runtimes become especially inefficient in this case,
- The data structures implemented with FastSGX consistently outperform the equivalent data structures implemented with the Intel development of Intel.

The remainder of the paper is organized as follows: §2 gives the background, §3 presents the design of FastSGX, §4 details our evaluation, §5 presents related works, and §6 concludes.

2 Background and threat model

In order to use Intel SGX, a developer defines enclaves. An enclave is a contiguous memory zone located inside the virtual address space of a process. Intel SGX protects an enclave by leveraging two processor execution modes: a non-secure mode and a secure mode. When the processor runs in non-secure mode, it prevents any access to the memory of the enclaves. When the processor enters secure mode, it gains access to a single enclave. In secure mode, the processor can access the memory of that enclave, the memory located outside any enclave, but not the memory of the other enclaves.

In order to protect the enclaves, the processor first prevents read and write access to the pages that belong to the enclave. Only preventing read and write access from the processor is not enough if we suppose an attacker that controls the operating system, the hypervisor, or the hardware. Such an attacker can bypass the protection mechanisms of the processor by using direct memory access from the devices. In order to prevent such attacks, the processor encrypts the cache lines before sending them to the main memory, which enforces confidentiality. With Intel SGX v1, the processor additionally enforces authenticity by maintaining a tree of hashes used to detect unintended writes [23].

In the remainder of the paper, we suppose an attacker that fully controls a machine (operating system and hypervisor included). We suppose that the attacker cannot read or write the memory of the enclaves protected by Intel SGX. For that, we suppose that the processor, the FastSGX runtime, and the software development

```
1 // initialize the runtime with nenclaves enclaves
2 int initialize(size_t nenclaves, ...);

3 // create a worker thread in the enclave eid
4 int new_worker(pthread_t* tid, size_t eid);

5 // send the value to the enclave eid with the message id mid
6 void send(size_t eid, size_t mid, union value value);

7 // receive a message with the message id mid from eid
8 void recv(size_t eid, size_t mid, union value* value);
```

Fig. 1 Main functions of the FastSGX interface.

kit provided by Intel to use SGX are correct and do not contain bugs. We do not consider side-channel attacks since Intel SGX does not address this attack vector.

3 FastSGX design

FastSGX is a message-oriented runtime for Intel SGX. It manages a set of enclaves. For each enclave, FastSGX creates a communication channel. Internally, FastSGX implements a communication channel as a lock-free FIFO queue stored in unsafe memory [9]. FastSGX also implements a lock-free memory allocator in order to allocate and free the messages. For that, FastSGX uses a simple lock free stack [9].

Additionally to the enclaves and communication channels, FastSGX manages a set of worker threads. Each worker thread is associated to a single enclave. It receives messages through the communication channel of its enclave. If several worker threads are associated to the same enclave, they share the communication channel. In this case, a message is not broadcasted to the worker threads: each message is only received by a single worker thread.

3.1 Interface

Figure 1 presents the main functions provided by FastSGX. To initialize the runtime, a developer calls the initialize function. This function takes the number of enclaves that have to be created as a parameter and, for each enclave, a path to the binary that has to be loaded in the enclave. The initialize function gives sequentially an enclave identifier for each enclave by starting with the eid 1. After initialization, FastSGX considers that the pseudo-enclave with the eid 0 represents the code and data located in unsafe memory. It also considers that the main thread of a process, i.e., the thread that called initialize, is a worker thread associated to enclave 0.

In order to start executing code in the enclaves, the developer has to call the `new_worker` function. This function creates a new worker thread. The new worker thread enters the enclave eid given as a parameter by executing the function named `start_routine`, which is located in the binary loaded in enclave eid.

As soon as worker threads execute in the enclaves, they can communicate by exchanging messages. For that, FastSGX provides the `send` and `recv` functions. The `send` function takes three parameters: the destination (eid), a message identifier (mid), and a content (value). The message identifier is used to address a message to a specific `recv` function, which is useful to avoid confusing two messages with two different meanings received in parallel. The value exchanged in a message is a union that has the size of a machine word (64 bits on an Intel).

The `recv` function takes the same three parameters. If eid is equal to -1, the function receives messages from any enclave, otherwise, it only receives messages from eid. If mid is equal to -1, the function receives messages with any mid, otherwise, it only receives messages with the mid given as a parameter. The `recv` function blocks until a message that matches eid/mid is received. For that, as with switchless call, the `recv` function actively spins on the message queue while waiting for a new message. When such a message arrives in the communication channel of an enclave, the `recv` function removes the message from the channel, fills the value with the content of the message, and returns.

3.2 Hazard pointers

Since FastSGX implements the communication channels with lock-free queues, it is subject to the ABA problem [17]. This problem appears with lock-free data structures implemented with compare and swaps.

To illustrate, we first give an overview of the algorithm used to implement a lock-free queue. We suppose a queue with two messages A and X. To dequeue A, a receiver thread t_1 loads a pointer to A from the head of the queue and then loads a pointer to X from the `next` field of A. Finally, the receiver thread replaces the head by the pointer to X. Since another receiver thread could dequeue A between the read of the head and its update in t_1, t_1 only updates the head from A to X if the head is still equal to A. For that, t_1 uses an atomic compare and swap instruction, which atomically compares head to A, and, if they are equal, replaces head by X. This simple algorithm is correct, but only if a message is never freed. In FastSGX, a receiver has to free a message when it is consumed in order to avoid a memory leak, which leads to the ABA problem described below.

To illustrate the ABA problem, we also suppose a queue with two messages A and X. The receiver thread t_1 loads A and X, and then, another receiver thread t_2 is scheduled. t_2 dequeues A and X, and frees the messages. Finally, a sender thread t_3 inserts a new message B. At this step, the queue contains a single message: B. When t_1 is re-scheduled, the compare and swap is supposed to fail because B is not the A message. However, this is not necessarily the case: since A is free when t_3 allocates

a message, B may be allocated at the location of A. In such a case, the compare and swap of t_1 succeeds since t_1 thinks that A is still the head of the list. t_1 thus install a pointer to X as the head of the queue, which is incorrect because X was freed by t_2.

FastSGX avoids the ABA problem by using hazard pointers [17]. Technically, when a receiver loads the head pointer, it signals to the other threads that freeing the pointed message is unsafe. For that, the receiver thread records the head pointer in an array named the hazard pointer array. Then, to free a message, FastSGX leverages two lists: a purgatory list and a free list. When the application frees a message, FastSGX adds the message in the purgatory list. When an application allocates a message, FastSGX uses the free list. If the free list is empty, FastSGX tries to move a batch of N messages from the purgatory list to the free list. It inspects the purgatory list by starting from the least recently added message, and, if the message is still referenced by the hazard array, FastSGX ignores the message since a receiver may still use the message. Otherwise, FastSGX moves the message to the free list.

4 Evaluation

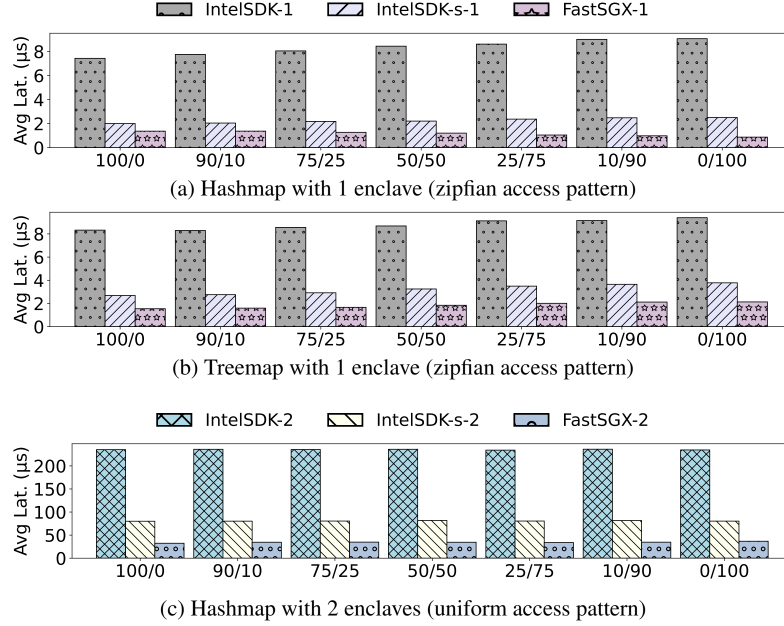
We evaluate the performance of FastSGX on an Intel i5-9500 CPU 3 GHz with 16 GiB memory. This 6-core CPU ships SGX version 1 with a maximal memory size usable by the enclaves of 93 MiB. The machine runs Linux 5.15.0, glibc 2.31, clang 10.0.0, and Intel SGX SDK 2.19.100.

4.1 Data structures

We first evaluate two maps that associate keys to values: a hashmap and a treemap. The hashmap uses a separate chaining algorithm: it is designed as an array of linked-lists, in which each linked-list contains the keys that collide. The treemap is implemented as a red-black tree, which ensures that the tree remains balanced.

We evaluate three versions with a single enclave: IntelSDK-1, IntelSDK-s-1 and FastSGX-1. These versions store the whole map in a single enclave. The two IntelSDK versions expose the `put`/`get` functions to the non-secure domain. IntelSDK-1 switches the processor mode during a call, while IntelSDK-s-1 uses switchless calls. FastSGX-1 is implemented with FastSGX. For a `get`, the worker thread in the non-secure domain sends a message to execute the get in the enclave, and waits for the result. For a `put`, since the result of a `put` is not used, the worker thread in the non-secure domain sends a message to execute `put`, but continues its execution in parallel.

We also evaluate three versions with two enclaves. These versions store the keys in one enclave and the values in another, which makes the communication pattern between the enclaves more complex. As with a single enclave, we evaluate



(a) Hashmap with 1 enclave (zipfian access pattern)

(b) Treemap with 1 enclave (zipfian access pattern)

(c) Hashmap with 2 enclaves (uniform access pattern)

Fig. 2 Latency of the data structures. X/Y: get/put ratio.

(i) IntelSDK-2, which switches the processor mode, (ii) IntelSDK-s-2, which uses switchless calls, and (iii) FastSGX-2, which uses message passing.

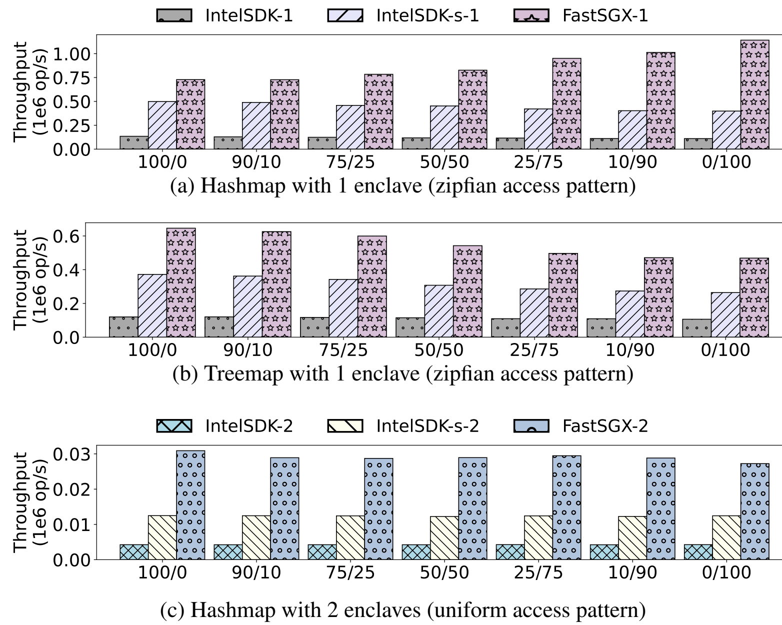
Figure 2 reports the latency and Figure 3 the throughput with different `put`/`get` ratios. With one color, we execute 100 000 operations in a map with 100 000 keys, and with two colors, 20 000 operations in a map with 20 000 keys. A key is 8-bytes long and a value 1024-byte long.

With one enclave, we observe that, as expected, IntelSDK-s-1 consistently performs better than IntelSDK-1. We also observe that FastSGX-1 is consistently better than IntelSDK-1 and IntelSDK-s-1. The better performance of FastSGX-1 comes from two complementary phenomenons.

First, with a high number of puts, a put is executed in parallel with the load injector in FastSGX-1. This is not the case with IntelSDK-s-1, which suspends the caller during a call. The better parallelism of FastSGX-1 explains why FastSGX-1 is better than IntelSDK-s-1 with a high number of puts (right of the curves).

Second, FastSGX is designed with lock-free data structures while Intel SDK uses a blocking scheme. In detail, with IntelSDK-s-1, a thread takes a spin-lock when it accesses an activation zone. FastSGX-1 does not take a lock since it uses lock-free data structures. Thanks to the use of lock-free data structures, FastSGX-1 is also better than IntelSDK-s-1 with a high number of get operations (left of the curve).

With two enclaves, we observe similar results. With two enclaves, the relative difference between FastSGX and the Intel SDK versions is higher because FastSGX allows the two enclaves to communicate directly. This is not the case with the Intel



(a) Hashmap with 1 enclave (zipfian access pattern)

(b) Treemap with 1 enclave (zipfian access pattern)

(c) Hashmap with 2 enclaves (uniform access pattern)

Fig. 3 Throughput of the data structures. X/Y: get/put ratio.

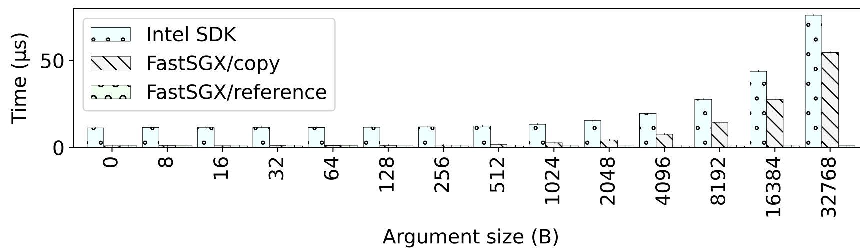


Fig. 4 Ping-pong time with Intel SDK and with FastSGX.

SDK versions, which pay an additional transfer of control to the non-secure domain for each operation. This result confirms that using an explicit message-passing scheme is important to optimize a multi-enclave application.

4.2 Ping-pong

In this experiment, we compare the cost of switching the processor with the cost of exchanging messages. For that, we evaluate a ping pong application in three configurations: FastSGX/copy, FastSGX/reference and Intel SDK.

With FastSGX/copy and FastSGX/reference, we run two worker threads: one worker thread in the non-secure domain and one worker thread in an enclave. The non-secure worker thread sends a ping to the enclave worker thread, which answers with a pong message. The ping message contains a pointer to an array of bytes. In FastSGX/copy, the application copies the pointed array in a buffer before replying with a pong. In FastSGX/reference, the application ignores the argument.

With Intel SDK, the non-secure thread of the application calls a ping function provided by the enclave. The non-secure thread of the application is blocked during the execution of the ping function, which is equivalent to waiting for a pong. With Intel SDK, the buffer is copied from the non-secure domain into the enclave.

Figure 4 reports the results of the experiment. We first observe that, when the size of the argument is equal to 0, using message passing instead of mode switching divides the function execution time by 12 (11.2 µs for Intel SDK versus 0.9 µs for FastSGX/copy or FastSGX/nocopy). This result highlights the benefit of using messages instead of switching the processor mode.

We also observe that, when the argument size increases, the cost of copying the buffer becomes larger than the cost of transferring the control. With a buffer of 32 KiB, we observe, however, that using message passing still saves 28% of the time (76.1 µs for Intel SDK versus 54.6 µs for FastSGX/copy). This result shows that, even for an application that protects large user data sets by copying them in an enclave, using message passing remains interesting.

5 Related works

Many applications directly rely on the Intel SDK to use SGX [3,5,7,11,22,30,35]. Since using the Intel SDK can be complex for legacy applications, several frameworks propose to run a complete application with its dependencies in an enclave [1, 2, 16, 19, 25]. These frameworks are not satisfactory because they lead to a large trusted computing base. Other tools propose to automatically partition an application by starting from variables or functions annotated as sensitive [4, 8, 10, 12–15, 20, 26, 29, 31, 32]. These tools ease the development while minimizing the trusted computing base. These tools are complementary to FastSGX: they could rely on FastSGX to optimize the time to transfer the control from/to an enclave.

As presented in the introduction, several runtimes rely on switchless calls to avoid the cost of switching the processor from/to secure mode [1, 24, 28, 33, 34]. These runtimes hide the worker threads to the developer, which makes the dynamic optimization of the number of worker threads difficult, prevents the execution of code in parallel in the worker threads, and is sub-optimal for a multi-enclave application. With FastSGX, by exposing the worker threads to the developer, and by exposing a message-passing interface, we avoid these three limitations.

EActors [21] proposes to design a SGX application as a set of actors. As in FastSGX, EActors runs worker threads in the enclaves. The worker threads execute

eactors, which communicate by exchanging messages. Using EActors requires a whole redesign of an application in order to implement the application with actors. Using FastSGX is more straightforward since only adding calls to send and receive messages is required. Moreover, with FastSGX, the developer explicitly creates on the fly the worker threads, which allows the developer to dynamically adjust the number of worker threads to the workload. This is not the case with EActors. With EActors, the number of worker threads is configured statically, which is inadequate for an application with a dynamic workload because worker threads may become useless in case of a low workload, or saturated in case of a high workload.

6 Conclusion

This paper presents FastSGX, a message-based runtime for SGX. FastSGX relies on the switchless call principle, but avoids the limitations of the current implementations. In detail, FastSGX (i) allows the developer to adjust the number of worker thread to the actual workload on the fly, (ii) allows the developer to execute code in parallel in the sender while a receiver proceeds a message, and (iii) allows the developer to directly transfer the control from an enclave to another. Thanks to these properties, our evaluation with different data structures and workloads shows that FastSGX consistently outperforms the SGX development kit of Intel, and with one or two enclaves. Our evaluation also shows that FastSGX, with its 4 main functions, is simple and usable in practice.

References

1. Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.
2. Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.
3. Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.
4. David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 5, USA, 2004. USENIX Association.
5. Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

6. Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.
7. Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, New York, NY, USA, 2022. Association for Computing Machinery.
8. Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 571–586, Renton, WA, July 2019. USENIX Association.
9. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
10. Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient SGX programming and its applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS 2020)*, pages 826–840, Taipei, Taiwan, 2020.
11. Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
12. Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association.
13. Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.
14. Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.
15. Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 437–456.
16. Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 205–216. IEEE, 2021.
17. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, jun 2004.
18. Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 238–253, 2017.
19. Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.
20. Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 923–934, New York, NY, USA, 2016. Association for Computing Machinery.
21. Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using sgx. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 187–200, New York, NY, USA, 2018. Association for Computing Machinery.

22. Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy (SSP 15)*, pages 38–54, San Jose, CA, USA, 2015. IEEE.
23. Meysam Taassori, Ali Shafiee, and Rajeev 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*, ASPLOS '18, page 665–678, New York, NY, USA, 2018. Association for Computing Machinery.
24. Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel sgx. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, page 22–27, New York, NY, USA, 2018. Association for Computing Machinery.
25. Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.
26. Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient Java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, Online, August 2020. USENIX Association.
27. Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. Sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 201–213, New York, NY, USA, 2018. Association for Computing Machinery.
28. Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 81–93, New York, NY, USA, 2017. Association for Computing Machinery.
29. Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, page 323–333. IEEE Press, 2013.
30. P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana. Plinius: Secure and persistent machine learning model training. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–62, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.
31. Peterson Yuhala, Hugo Guiroux, Jean-Pierre Lozi, Pascal Felber, Valerio Schiavoni, Alain Tchana, and Gaël Thomas. Secv: Secure code partitioning via multi-language secure values. In *Proceedings of the 24nd International Middleware Conference*, Middleware '23. Association for Computing Machinery, 2023.
32. Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel sgx shielding for graalvm native images. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 352–364, New York, NY, USA, 2021. Association for Computing Machinery.
33. Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. SGX switchless calls made configless. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 229–238. IEEE, 2023.
34. Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. SGX Switchless Calls Made Configless (PER). In *Proceedings of the international conference on Dependable Systems and Networks, DSN'23*. IEEE Computer Society, 2023.
35. Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, USA, 2017.