A Case for Unified ABI Shielding in Intel SGX Runtimes

Jo Van Bulck, Fritz Alder, Frank Piessens
imec-DistriNet, KU Leuven, Belgium

ABSTRACT

With hardware support for trusted execution, most notably Intel SGX, becoming widely available, recent years have seen the emergence of numerous *shielding runtimes* to transparently protect enclave applications in hostile environments. While, at the application level, a wide range of languages and development paradigms are supported by diverse runtimes, shielding responsibilities at the lowest level of the application binary interface (ABI) remain strikingly similar. Particularly, the ABI dictates that certain CPU registers need to be cleansed and initialized via a small, hand-written assembly stub upon every enclave context switch.

This paper and call for action analyzes the ABI sanitization layers of 8 open-source SGX shielding runtimes from industry and academia, categorizes historic vulnerabilities therein, and identifies cross-cutting tendencies and insights. We conclude that there is *no* technical reason for maintaining separate, often notoriously complex and vulnerable ABI code bases. Moving forward, we outline challenges and opportunities for a single, unified ABI sanitization layer that complies with best practices from software engineering and can be scrutinized and integrated across SGX runtimes.

ACM Reference Format:
Jo Van Bulck, Fritz Alder, Frank Piessens. 2022. A Case for Unified ABI Shielding in Intel SGX Runtimes. In *Proceedings of the 5th Workshop on System Software for Trusted Execution (SysTEX '22 Workshop)*. ACM, New York, NY, USA, 3 pages.

1 PROBLEM STATEMENT

Recent years have seen the rise of an emerging software ecosystem taking advantage of modern processor support for trusted execution, most notably Intel SGX. A key element in this ecosystem is the development of numerous open-source *shielding runtimes* that can transparently protect third-party enclave applications against their untrusted environments. Notable examples include Intel's SGX-SDK [11] and Microsoft's Open Enclave (OE) [13] for developing C/C++ enclaves, Fortanix's Enclave Development Platform (EDP) [7] integrated into the Rust compiler, the Enarx [5] shielding system for WebAssembly, and the library operating systems Gramine [10] (formerly Graphene [16]) and SGX-LKL [14].

The complex requirement of transparently shielding enclave applications from their untrusted environment has been previously broken down into two successive tiers of interface sanitization responsibilities [17]. First, shielding runtimes are responsible to cleanse low-level machine state so as to establish a trustworthy application binary interface (ABI) expected by the compiler. Relevant sanitizations at the ABI level commonly include initializing selected configuration registers and setting up a trusted stack on enclave entry, as well as scrubbing any secrets from CPU registers on enclave exit. This bootstrapping phase is typically implemented by means of a small, hand-written assembly stub that is transparently executed on every enclave context switch. Next, a secondary stage, written in a higher-level language, may sanitize application programming interface (API) state, such as pointer arguments. It is worth noting that low-level ABI shielding responsibilities are relatively contained and language-agnostic, whereas sanitizing program-visible API state is typically more complex and may be highly dependant on the specific runtime and supported programming model.

Table 1: Overview of the Intel SGX ABI vulnerability landscape. The top rows compare ABI sanitization layers in terms of total lines of code (as measured on January 20, 2022; using `cloc`) and lines changed since original release (as reported by `git`; following renamed/moved files). The third row distinguishes (aspired) production runtimes from research prototypes. The bottom rows list which runtimes were found to be vulnerable to (●), not vulnerable to (○), or not analyzed by (–) prior attack studies.

		SGX-SDK*	OE**	EDP	Gramine	Enarx	GoTEE	SGX-LKL	OpenSGX
Metrics	LoC ABI stub	301	277	248	427	169	239	103	49
	LoC changed	243	589	187	1,840	844	65	47	0
	Production?	✓	✓	✓	✓	✓	✗	✗	✗
Vulnerabilities	Entry flags [17]	●	●	●	●	–	–	●	–
	Entry stack [17]	○	○	○	●	–	–	●	–
	Exit registers [17]	○	○	○	○	–	–	●	–
	Entry FPU [1]	●	●	●	○	○	●	●	–
	Exception stack [3]	●	●	○	○	●	–	●	–

* Derived runtimes include Apache Teaclave [15, 18], VeraCruz [2], and Google Asylo [9].
** Derived runtimes include EdgelessRT [4], and recent versions of SGX-LKL "OE edition".

ABI vulnerability landscape. Several recent attack studies [1, 3, 17] have exploited subtle ABI-level sanitization oversights in SGX shielding runtimes. Table 1 comprehensively overviews this ABI vulnerability landscape by analyzing the sanitization layers of 8 open-source SGX shielding runtimes from industry and academia.[1] For every runtime, we report both its proneness to previous ABI vulnerabilities, as far as it was analyzed by the corresponding attack studies [1, 3, 17], as well as metrics obtained from the open-source repositories in terms of the current total lines of code of the relevant assembly stub, and lines changed since the original release. The former metric provides a rough indication of the relative complexity of the ABI layer, whereas the latter allows to assess how stable the sanitization code has been.

A first important insight from Table 1 is that the complexity of ABI sanitization is evidently non-trivial, ranging in the order of hundreds of lines of carefully developed, hand-written assembly logic. While larger ABI layers do not necessarily equate to more or less vulnerabilities, undersized research prototypes [8, 12, 14] may offer inadequate protection. Consider, for instance, SGX-LKL's original, especially vulnerable proof-of-concept ABI layer, which has recently been retired in favor of adopting OE. Second, while the effort to develop a custom ABI sanitization layer has been duplicated across all projects, there appears not to be a single runtime that has

[1] Analysis data available at https://github.com/jovanbulck/sgx-abi-data.

Table 2: Overview of ABI patch timelines in SGX production runtimes. The top row provides initial commit dates as a reference. The next rows list the dates of the initial patch (and the last revision, if any) for the ABI sanitization responsibilities in the left column, where 🗎 and ★ indicate vulnerabilities disclosed by the referenced academic study and this work, respectively.

	SGX-SDK	OE	EDP	Gramine	Enarx
Initial commit	° 24/06/16	° 29/08/17	° 07/12/18	° 20/06/16	° 20/02/20
DF [17]	🗎 17/10/19	🗎 09/10/19	07/12/18	01/05/19	20/03/20
AC [17]	🗎 12/11/19	🗎 09/10/19	🗎 21/10/19 10/02/20	🗎 19/11/19	★ 17/02/22
FPU [1]	🗎 16/01/20	09/10/19 🗎 14/07/20	🗎 10/02/20 🗎 19/06/20	17/10/19	29/05/20
EXC [3]	🗎 13/07/21	🗎 13/07/21	N/A	01/04/19 31/01/20	🗎 22/10/21

DF = direction flag sanitization; AC = alignment-check flag sanitization; FPU = extended-state sanitization; EXC = exception handler stack pointer initialization.

been immune to all attacks (excluding runtimes that have not been studied before). Third, relative to their size, these ABI software stubs have been heavily modified over recent years. There is, furthermore, a clear discrepancy between research prototypes and aspired production-quality runtimes in terms of code line changes, strongly indicating that maintaining a secure ABI sanitization layer is an *ongoing* and *living* effort. This point becomes especially evident when comparing recent runtimes to OpenSGX [12], a discontinued research prototype before SGX hardware and production runtimes became widespread available. Compared to OpenSGX's extremely rudimentary (and incomplete) ABI sanitization layer, recent runtimes have clearly matured and ABI sanitization responsibilities are now much better understood.

ABI patch timelines. Perhaps more concerning upon closer inspection of historic ABI vulnerabilities is that insufficient coordination between developers of different runtimes often allows the same oversights to be repeated across different code bases. Moreover, a lack of deeper understanding and scrutinization has sometimes led to only partial patches being applied. For instance, OE developers appear to have been first aware of FPU sanitization oversights and tried to initially patch them, without informing other potentially affected runtimes. A systematic attack study [1] subsequently revealed that the initial OE patch, which was later also adopted by EDP, lacked full understanding of the issue and left some subtle attack surface, ultimately leading OE and EDP to have to adopt a second patch. Similar observations hold for other ABI-level sanitization pitfalls [3, 17] that were at some point partially or fully understood by certain runtime developers, but not communicated towards other potentially affected projects (cf. Table 2). In fact, our systematic analysis revealed a subtle alignment-check flag sanitization oversight in Enarx, which has now also been patched.

2 TOWARDS ABI UNIFICATION

From our analysis of the ABI vulnerability landscape, we conclude that SGX runtimes have substantially matured in recent years. We welcome diversity at the application and programming language levels, but we argue that the time is now ripe for making a first step towards unification of shared insights at the ABI level. That is, compliant with best practices from software engineering, code duplication should be avoided wherever possible, and developers efforts are instead better focused on one common project. We, hence, envision a *unified* ABI shielding layer that can be jointly developed and easily shared across SGX runtimes.

ABI shielding responsibilities. In principle, standardized calling conventions adopted by mainstream compilers [6] would allow a single ABI layer to transparently shield binaries of different runtimes supporting diverse programming languages. That is, the main responsibility of the ABI layer is to bridge well-documented compiler expectations regarding the low-level machine state. Specifically, upon enclave entry, the ABI layer should *(i)* initialize CPU and FPU configuration registers to expected sane values; *(ii)* point the stack to trusted, in-enclave memory; *(iii)* jump to the runtime entry point written in higher-level code; and, upon return, *(iv)* cleanse all CPU registers not holding return values before enclave exit.

While there are some runtime-specific differences, especially regarding step *(ii)* for optional, in-enclave exception handlers [3], we argue that these differences are overcomable. Particularly, all runtimes essentially follow the same high-level ABI shielding flow that can be abstracted via a unified ABI layer supporting (nested) enclave entry calls "ecalls", out calls "ocalls", and exception handlers.

Design principles. We envision a single, language-agnostic ABI code base that can be easily integrated into varying SGX runtimes. Specifically, the ABI library would be preferably standardized and developed as a joint project in a standalone repository maintained by relevant practitioners from industry and academia, e.g., via the newfound Confidential Computing Consortium.

Integration of the ABI layer into various runtimes would proceed through the linker. Particularly, all API-level shielding responsibilities should be deferred to runtime-specific, higher-level code, and runtimes should declare a standardized entry point name where control is transferred to from the assembly code. Certain challenges may exist in terms of adopting a uniform convention for the enclave's binary layout. That is, the unified ABI layer will have to make assumptions on the locations and layout of trusted in-enclave stacks for ecalls and exception handlers. Furthermore, in order to obtain a relocatable enclave image, the uniform ABI layer would preferably avoid any absolute symbols, e.g., by assuming all thread-specific data structures are located relative to SGX's hardware-provided TCS pointer. We expect that any potentially remaining runtime-specific ABI logic, e.g., support for 32-bit mode, could be relatively easily encapsulated by means of precompiler configuration options.

3 CONCLUSIONS AND OUTLOOK

The logical next step in the emerging landscape of SGX shielding runtimes is to strive for more unification. While limiting code diversity may come at the risk of introducing "superbugs" that would be exploitable across *all* runtimes, we believe that continued attack evidence supports the need to concentrate developer efforts on a single ABI code base. We expect that such a unification would, furthermore, inspire efforts to limit the risks of any remaining vulnerabilities through established methods, such as testing, fuzzing, symbolic execution, and ultimately even formal verification.

ACKNOWLEDGEMENTS

This research is partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by a gift from Intel Corporation. Jo Van Bulck and Fritz Alder are supported by a grant of the Research Foundation – Flanders (FWO).

REFERENCES

[1] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens. 2020. Faulty Point Unit: ABI Poisoning Attacks on Intel SGX. In *36th Annual Computer Security Applications Conference (ACSAC)*. 415–427.
[2] ARM Research. 2022. Veracruz: Privacy-Preserving Collaborative Compute. https://veracruz-project.com/.
[3] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *28th ACM Conference on Computer and Communications Security (CCS)*. 779–793.
[4] Edgeless Systems. 2022. Edgeless RT. https://github.com/edgelesssys/edgelessrt.
[5] Enarx Project. 2022. Enarx: WebAssembly + Confidential Computing. https://enarx.dev/.
[6] A. Fog. 2021. Calling Conventions for Different C++ Compilers and Operating Systems.
[7] Fortanix. 2022. Fortanix Enclave Development Platform – Rust EDP. https://edp.fortanix.com/
[8] A. Ghosn, J. R. Larus, and E. Bugnion. 2019. Secured routines: Language-based construction of trusted execution environments. In *USENIX Annual Technical Conference (ATC)*. 571–586.
[9] Google. 2022. Asylo: An Open and Flexible Framework for Enclave Applications. https://asylo.dev/
[10] Gramine Project. 2022. Gramine – A Library OS for Unmodified Applications. https://grapheneproject.io.
[11] Intel. 2022. Intel Software Guard Extensions – Get Started with the SDK. https://software.intel.com/en-us/sgx/sdk
[12] P. Jain, S. J. Desai, M.-W. Shih, T. Kim, S. M. Kim, J.-H. Lee, C. Choi, Y. Shin, B. B. Kang, and D. Han. 2016. OpenSGX: An Open Platform for SGX Research.. In *23th Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 21–24.
[13] Microsoft. 2022. Open Enclave SDK. https://openenclave.io/
[14] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143* (2019).
[15] The Apache Software Foundation. 2022. Apache Teaclave (Incubating). https://teaclave.incubator.apache.org/.
[16] C.-C. Tsai, D. E. Porter, and M. Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*.
[17] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *26th ACM Conference on Computer and Communications Security (CCS)*. 1741–1758.
[18] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *26th ACM Conference on Computer and Communications Security (CCS)*. 2333–2350.