Faster enclave transitions for IO-intensive network applications

Svenningsson, Jakob; Paladi, Nicolae; Vahidi, Arash

Published in:
SPIN'21

2021

Link to publication

Citation for published version (APA):
Svenningsson, J., Paladi, N., & Vahidi, A. (in press). Faster enclave transitions for IO-intensive network applications. In *SPIN'21: Proceedings of the Workshop on Secure Programmable Network Infrastructure* Association for Computing Machinery (ACM).

Total number of authors:
3

Faster enclave transitions for IO-intensive network applications

Jakob Svenningsson
jaksve@kth.se
KTH Royal Institute of Technology
Sweden

Nicolae Paladi
nicolae.paladi@eit.lth.se
Lund University and RISE
Cybersecurity
Sweden

Arash Vahidi
arash.vahidi@ri.se
RISE Cybersecurity
Sweden

ABSTRACT

Process-based confidential computing enclaves such as Intel SGX have been proposed for protecting the confidentiality and integrity of network applications, without the overhead of virtualization. However, these solutions introduce other types of overhead, particularly the cost transitioning in and out of an enclave context. This makes the use of enclaves impractical for running IO-intensive applications, such as network packet processing. We build on earlier approaches to improve the IO performance of workloads in Intel SGX enclaves and propose the HotCall-Bundler library that helps reduce the cost of individual single enclave transitions and the total number of enclave transitions in trusted applications running in Intel SGX enclaves. We describe the implementation of the HotCall-Bundler library, evaluate its performance and demonstrate its practicality using the case study of Open vSwitch, a widely used software switch implementation.

CCS CONCEPTS

• **Security and privacy** → **Network security**; **Systems security**; **Security in hardware**; **Systems security**; • **Networks** → *Bridges and switches.*

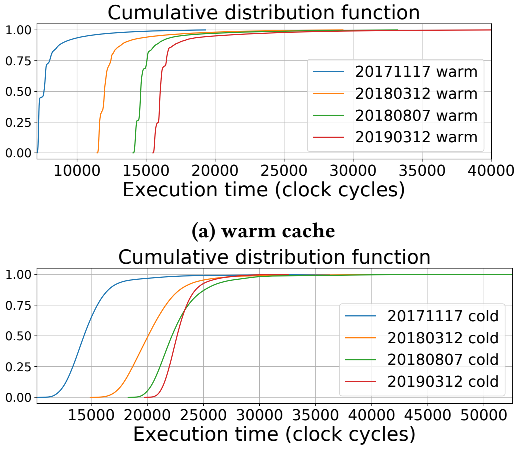
KEYWORDS

Open vSwitch, SGX, Hardware security, Performance optimization

ACM Reference Format:
Jakob Svenningsson, Nicolae Paladi, and Arash Vahidi. 2021. Faster enclave transitions for IO-intensive network applications. In *ACM SIGCOMM Workshop on Secure Programmable network INfrastructure (SPIN '21), August 23, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3472873.3472879

1 INTRODUCTION

Confidentiality and integrity are important topics when network computation moves from dedicated hardware to software deployed on shared commodity platforms. Addressing these topics should not offset the two core advantages of software network components: cost reduction and flexibility. *Confidential computing* is an increasingly popular approach to achieving this [34]. It relies on using a Trusted Execution Environment (TEE) backed by certified hardware, such that critical operations of Trusted Applications running inside the TEE cannot be manipulated by the platform operator or malicious entities (with the notable exception of the CPU manufacturer). For example, AMD SEV, Intel SGX, and IBM SVM provide mechanisms to achieve this in different ways [11, 20, 33]. The variety of vendor TEE implementations highlights the tradeoffs between security guarantees, portability of legacy applications, ease of deployment, and run-time performance. VM-based TEE implementations (e.g. AMD SEV, IBM SVM, and Intel TDX) support portability of legacy applications with a modest performance overhead [6], but have a larger attack surface and are vulnerable to several classes of attacks [17]. Process-based TEEs (e.g. Intel SGX and ARM TrustZone) on the other hand have a smaller attack surface and improved security and were proposed for protecting network applications [12, 13, 24, 30]. Unfortunately, the additional security checks together with memory access limitations negatively affect the performance of process-based TEEs [6]. Furthermore, these have shown to be particularly vulnerable to microarchitectural attacks [26] and platform vendors have repeatedly issued microcode patches to alleviate security problems [5]. As illustrated in Figure 1, recent microcode updates for Intel SGX have further degraded TEE performance.



(a) warm cache

(b) cold cache

Figure 1: Evolution of the SGX enclave transition time through Intel microcode updates.

Considering the performance effects described above, it is imperative to identify and implement new approaches that to maintain or improve TEE performance despite the latest countermeasures to microarchitectural attacks. This however should not significantly

SPIN '21, August 23, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8637-1/21/08…$15.00
https://doi.org/10.1145/3472873.3472879

increase the efforts required from network application developers. In this paper, we address a crucial limitation on the intersection between the portability of legacy network applications and the performance overhead introduced by the transition between the TEE and the Rich Execution Environment (REE) that comprises the platform operating system.

Our results show that while a tailor-made refactoring of legacy Trusted Applications for Intel SGX yields the best performance, it is work-intensive, application-specific, and often impractical. This insight led us to develop the *HotCalls-Bundler* software library as a generic approach for speeding up enclave transitions in legacy Trusted Applications, while maintaining the security benefits of SGX. This solution is particularly beneficial in IO-intensive applications such as network packet processing, as well as non-network applications such as remote sensing applications, biological sequence analysis, and long-running simulations [25]. We demonstrate the practical applicability and performance improvements of our approach using Open vSwitch, a popular software switch implementation.

The main **contributions** of our work are summarized as follows:

- We describe a generic approach to speed up transitions between the rich execution environment and SGX enclaves (section 3);
- We introduce *enclave execution graphs*, that allow executing arbitrary sequences of enclave functions using a single enclave transition (section 4);
- We implement a library to assist refactoring of legacy applications and introduce efficient transitions in and out of SGX enclaves;
- We demonstrate the applicability of our approach with the case study of Open vSwitch, a widely used IO-intensive network application;
- The implementation source code is openly available[1].

The rest of the paper is structured as follows. We introduce the required background and motivate the problem in Section 2, introduce the HotCall-Bundler library in Section 3 and describe the implementation of the library in Section 4. Next, we evaluate the performance of the HotCalls-Bundler library and its application in a case study in Section 5, present the related work in Section 6 followed by conclusion and future work in Section 7.

2 BACKGROUND

We next introduce several key concepts used in the paper.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) are CPU security extensions that allow execution of unprivileged trusted applications in the presence of possibly malicious privileged software such as a compromised OS or hypervisor [20]. An SGX-enabled CPU maintains an isolated memory region, the Enclave Page Cache (EPC), within which security enclaves can execute isolated from the rest of the system. SGX provides mechanisms to verify the integrity of an enclave (using local and remote attestation) and binding of information to specific configurations (sealing), which allows one to validate an enclave without direct access to its content.

Enclaves communicate with applications running in the Rich Execution Environment (REE) using the ECALL and OCALL (entry and out call) instructions. However, these instructions introduce a performance overhead that often makes SGX unsuitable for IO-intensive applications. Weisse proposed "HotCall", which utilizes a shared memory region outside the enclave for communication, resulting in significant performance improvements in real-world applications [32]. In response to published security vulnerabilities affecting Intel SGX [31], [18], [15], Intel issued a number of microcode updates. However, along with addressing software vulnerabilities this further degraded the performance of enclave transitions (see Figure 1). While the HotCalls approach [32] produces a tangible performance improvement, we note the importance of further efforts to offset the overhead introduced by subsequent microcode updates.

2.2 Memoization

Memoization is an optimization technique for reducing the execution time of computationally expensive functions [14]. Given a function with no side effects, memoization uses a cache to remember some input-output pairs. If an input used in a subsequent call is found in the cache, the recorded output value is returned, otherwise, the (expensive) function is called. Memoization is a simple way of trading execution time for space and is commonly used to optimize recursive algorithms. We use memoization to reduce enclave transitions between applications running in the TEE and the REE.

2.3 Open vSwitch

The motivating use case for this work is Open vSwitch (OvS), a software network switch for connecting physical and virtual network interfaces in a virtualized environment [16]. This is a critical component for providing network isolation in cloud infrastructure and other multi-tenant environment [23].

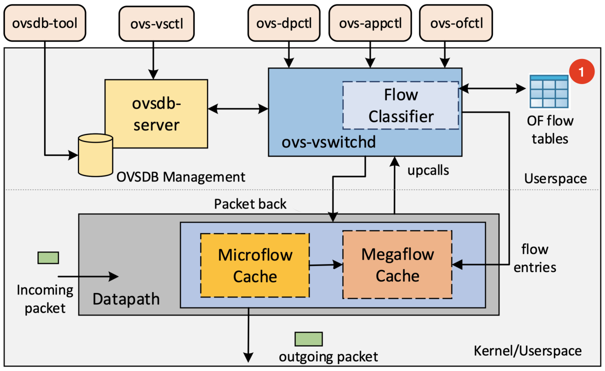


Figure 2: Overview of OvS main components.

Among the OvS components (Figure 2), the *flow tables* (1) are of special interest to us as they contain the rules that define the routing behavior of a switch. While these tables are critical OvS assets,

[1] Source code repository: https://anonymous.4open.science/r/hotcall-bundler

they are often stored without sufficient confidentiality and integrity protection, leading to serious security vulnerabilities. For example, an attacker with access to flow tables could map the network structure [4], modify routing behavior to perform man-in-the-middle attacks, or bypass firewalls and intrusion-detection systems [3]. Furthermore, an attacker could inject malicious data into flow tables to propagate deeper into the network and compromise systems otherwise not reachable [8]. Proposed solutions to address flow table security issues include auditing flow table to detect discrepancies between the configured and current behavior [19], validating both executables and flow tables with a TPM [10], or moving critical components (the OpenFlow flow tables and forwarding logic) into Intel SGX enclaves [21]. The latter, while promising from a security point of view, is a very labor-intensive task and introduces additional overhead. We address both shortcomings in this work.

3 SPEEDING UP ENCLAVE TRANSITIONS

We next describe the *HotCall-Bundler* mechanism to address the performance penalty introduced by transitions into and out of SGX enclaves. To facilitate adoption and usability, we designed and implemented this mechanism as a software library.

3.1 Overview

The *HotCall-Bundler* library offers functionality to reduce the cost of individual enclave transition as well as the total number of enclave transitions for trusted applications (TAs) deployed in Intel SGX enclaves. This library extends work conducted in HotCalls [32] with novel ideas and is the core contribution of this paper. The library leverages three main features: switchless enclave function calls, execution graphs, and enclave function memoization.

Switchless enclave function calls are used to reduce the cost of a single enclave transition. Execution graphs and enclave function memoization are used to reduce the total number of enclave function calls in Intel SGX applications.

3.2 Functional Requirements

We consider the following functional requirements for the HotCall-Bundler library:

(1) **Switchless calls**: execute enclave functions without context-switching to enclave mode;
(2) **Merging**: execute an arbitrary number of enclave functions over a single enclave transition;
(3) **Batching**: apply an arbitrary number of enclave functions to each element of an input list over a single enclave transition;
(4) **Branching**: conditional execution of enclave functions over a single enclave transition;
(5) **Memoization**: cache enclave data in untrusted memory when only integrity is necessary. Caches allow untrusted applications to access data without transitioning into the enclave. Moreover, we implement a mechanism to verify the integrity of enclave data stored in untrusted memory.

The switchless enclave function call component presented in 4.1 fulfills requirement 1; the execution graph component in section 4.2 fulfills requirements 2-4, and the memoization component in section 4.4 fulfills requirement 5.

3.3 Architecture

In the case of SGX enclaves, implementing a shared memory switchless enclave communication library requires source code modifications in both the trusted application running in the TEE and the untrusted application running in the REE. Enclaves do not share source code (and libraries) with the untrusted application; therefore, the HotCall-Bundler library consists of two separate libraries. The first library is a *static C* library that needs to be linked with the untrusted application, and the second is a trusted enclave library which needs to be linked with the enclave binary [1].

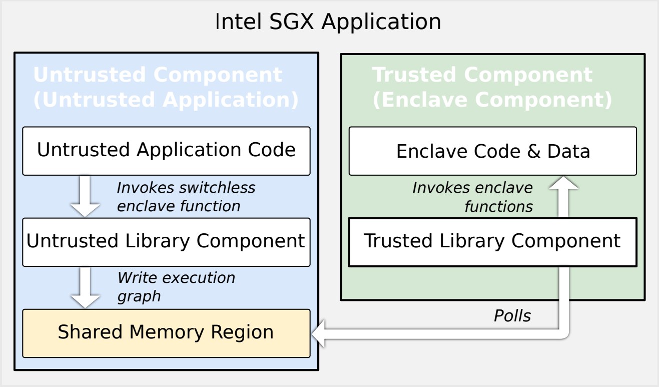

Intel SGX Application
Untrusted Component (Untrusted Application)
Untrusted Application Code
Invokes switchless enclave function
Untrusted Library Component
Write execution graph
Shared Memory Region
Trusted Component (Enclave Component)
Enclave Code & Data
Invokes enclave functions
Trusted Library Component
Polls

Figure 3: High-level overview of an Intel SGX application using the HotCall-Bundler library.

Figure 3 illustrates the untrusted and trusted part of the HotCall-Bundler library when integrated into an arbitrary Intel SGX application and the interactions between the different parts. The untrusted application invokes switchless enclave functions through an API exposed by the untrusted library. Next, the untrusted library writes the job to a shared memory region in the form of an execution graph (execution graphs are discussed later in section 4.2). Finally, the job is processed by an enclave worker thread which calls the associated enclave function and writes back potential return values to the shared memory region.

4 HOTCALL-BUNDLER IMPLEMENTATION

We next describe the HotCall-Bundler implementation.

4.1 Switchless Enclave Function Calls

HotCall-Bundler protocol for switchless enclave function calls builds on HotCalls [32] and is presented in Figure 4. This component fulfills functional requirement (1) listed above in 3.2. The shared memory region contains a *spinlock* primitive that must be acquired by either the untrusted application or the TA before accessing the shared memory region to avoid data races. While Intel SGX SDK supports condition variables, this synchronization primitive is implemented with OCALLS, which is a context switch operation and conflicts with our goal of a switchless communication protocol. Spinlock is the only synchronization primitive that can be used by the enclave worker threads without leaving the enclave.

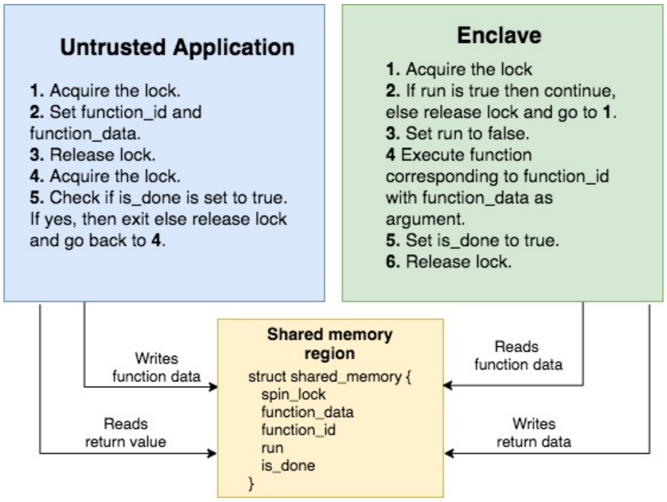


Figure 4: Switchless enclave function call protocol

The untrusted application invokes switchless enclave functions by acquiring the lock and writing the enclave function call, represented by a (*function_id*, *function_data*) tuple, to shared memory. An enclave worker thread initiated through an API exposed by the trusted part of the library is continuously polling the shared memory region for scheduled jobs to execute. The enclave worker thread uses a busy-waiting scheme where it repeatedly checks for pending jobs inside of an infinite loop. We use Intel's *pause* instruction inside of the spinlock loop to improve the efficiency of the busy-waiting scheme. The pause instruction provides a hint to the processor that it is executing inside a spinlock loop, enabling the processor to perform memory optimizations [9]. In Section 4.2) we replace this tuple with a data structure representing an *execution graph* to create a more efficient enclave communication scheme able to execute multiple enclave functions using a single enclave transition.

4.1.1 Translation Functions. Input and output parameters are treated as generic elements, which simplifies the implementation but must be translated to correct data types before an enclave worker thread can be invoked. This is done by defining a translation function for each function exposed to the untrusted application, see Listing 1 for an example. Note that translation functions are constructed to accept an array of parameters, which will enable the use of batching (see section 4.2.1).

Listing 1: A translation function for an enclave summation.

```
void translation_ecall_plus(
  unsigned int itrs, unsigned int params, void *args[][]) {
  for(int i = 0; i < iters; ++i) {
    *(int *) args[2][i] = hotcall_plus(
        *(int *) args[0][i], *(int *) args[1][i]);
  }
}
```

4.2 Execution Graphs

A limitation of the *HotCall* switchless enclave communication implementation [32] is that it only allows executing a single enclave function call per enclave transition. Each enclave transition introduces an overhead, estimated to be around ~600 to ~1400 clock cycles for warm and cold caches respectively [32].

This paper introduces *execution graphs* in the context of enclave transitions. An enclave *execution graph* is an arbitrary sequence of dependent or independent enclave functions, control statements, and iterators that can be executed with a single enclave transition. This provides a significant improvement over the original HotCall implementation and is to best of our knowledge a novel concept that has not been explored in previous studies. Figure 5 illustrates how two enclave function calls can be executed using only a single enclave transition using execution graphs. Execution graphs are represented as an array of items where each entry is either an enclave function, a control statement, or an iterator. In its simplest form, an execution graph is only a list of enclave functions that are executed sequentially. This enables arbitrary merging of enclave function calls and fulfills functional requirement (2) listed in section 3.2. Figure 6 illustrates a simple execution graph with three enclave function calls. Each function call is represented by a tuple (*function_id*, *function_data*), where *function_data* is a list of pointers. By convention, the last element in the function_data list is the address where potential return values are written. We next describe other node types that can be used to construct execution graphs. These enable batching and branching and fulfill the functional requirements 3 and 4 of section 3.2.

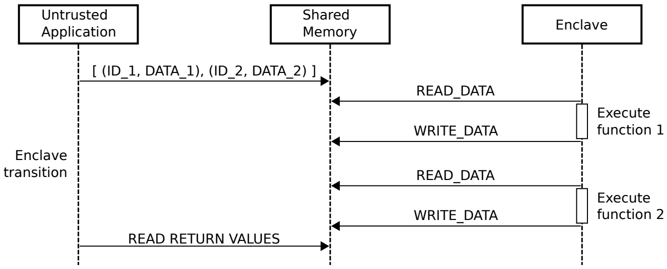


Figure 5: Sequence diagram illustrating the enclave and untrusted application interaction when invoking two enclave function calls using execution graphs.

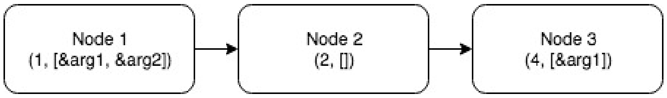


Figure 6: A simple execution graph consisting of three enclave function calls.

4.2.1 Iterator. Iterator-nodes allow multiple invocations of an enclave function with different parameters in a single enclave transition. Iterators can be used for functional style operators such as *map* and *for-each.* Figure 7 illustrates the overview of iterator implementation. Note that input parameters are now represented as a matrix which will be processed by a translation function before each row is used for one invocation of the target function.

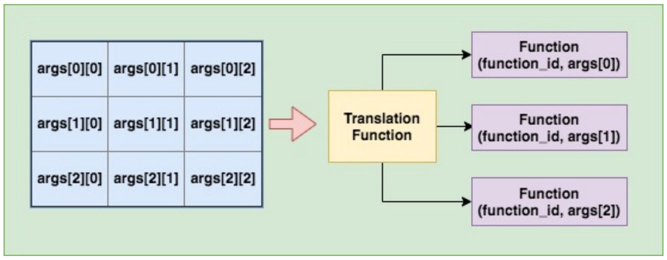


Figure 7: Iterator implementation.

4.2.2 If. If-nodes choose between two possible execution paths, possibly depending on the result of a previous enclave operations. An example is shown in Figure 8 where the choice between two enclave functions depends on value of `arg2`, which may have been modified in the previous enclave function call. The implementation allows use of complex conditions in postfix notation [7] and automatically handles required type conversions.

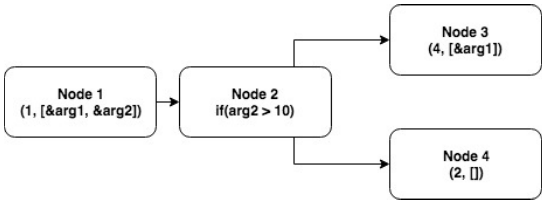


Figure 8: Execution graph containing an if-node.

4.2.3 For. A *for*-node allows to execute a subset of the execution graph repeatedly for n iterations. This is an alternative to iterators presented in section 4.2.1. However, for-loops can execute an arbitrary subset of the execution graph in each iteration while an iterator can only execute a single enclave function. Figure 9 illustrates a execution graph containing a for-node. Note that a for-loop requires two additional nodes to be inserted in the execution graph. One node in the front and one in the end. All parameters of enclave functions in the for-loop body which are marked as list parameters are automatically incremented in each loop iteration.

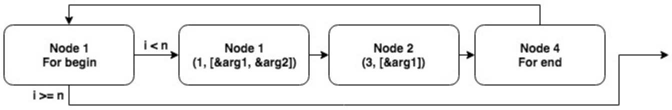


Figure 9: Execution graph containing a for-node.

4.2.4 While. A while-node allows to execute a subset of the execution graph repeatedly conditioned on a boolean expression. A while-node is implemented in the same way as the for-node presented in section 4.2.3. The difference between for and while-nodes is the loop condition. While-nodes use a boolean loop condition, which is implemented in the same way as boolean conditions in if-nodes, discussed in section 4.2.2.

4.3 Construction of Execution Graphs

Each node of the execution graph requires 5 – 10 lines of boilerplate code, which can make construction of graphs a tedious task and result in a less readable code. To simplify this process the untrusted part of the library exposes a user-friendly API based on C preprocessor macros for building execution graphs using both an imperative and functional-style syntax.

4.4 Enclave Function Memoization

Memoization enables caching of enclave data in untrusted memory, which is accessible by the untrusted application without any enclave interaction. This approach is widely applicable for many IO-intensive application use cases (see Section 1). Integrity of memoization caches in untrusted memory is guaranteed by storing a hash of each memoization cache in the enclave. The enclave worker thread, responsible for populating memoization caches, updates the corresponding memoization hash each time a cache entry is inserted or deleted. The enclave worker thread periodically verifies the caches by recalculating the hashes of the memoization caches in untrusted memory and compares them with the hashes stored in enclave memory. In our implementation, we clear the cache whenever an unauthorized modification is detected; however, other actions may be appropriate depending on the application. For example, appropriate actions in a production system may be clearing the cache and notifying the system administrator.

System time is not accessible inside of enclaves and hence cannot be used to implement periodic recalculation of memoization cache hashes. Instead, the periodic triggering of the recalculation mechanism is implemented using the spinlock loop used in the switchless enclave function call protocol described in section 4.1. The enclave worker thread decrements an integer value in each spinlock loop iteration, and the recalculation of memoization caches is triggered when the integer reaches zero. Later, the counter is reset to its original value and a new countdown begins.

5 PRELIMINARY RESULTS

We examined the performance impact of the proposed solutions in a complex network application (Open vSwitch commit `53cc4b0`).

We studied four Open vSwitch flow table operations: add flow rule, delete flow rule, modify flow rule, and evict flow rule. The performance of each operation has been compared across five different versions: *baseline* is the original version, *SGX vanilla* is the OFTinSGX from [21], *Switchless* uses hotcalls instead of ECALLs as described in [32] while *Bundler* uses all optimization described in this paper. Finally *SGX refactored* is the authors heavily modified version tailored specifically for SGX and will be used to compare the trade-offs between performance and development effort. Evaluation scripts are openly available[2].

Add Flow Rule. Figure 10 illustrates CDFs of add flow execution times for all evaluated versions. We observe a relatively similar performance for *SGX refactored*, *Bundler* and *Switchless* while *SGX vanillia* is significantly slower.

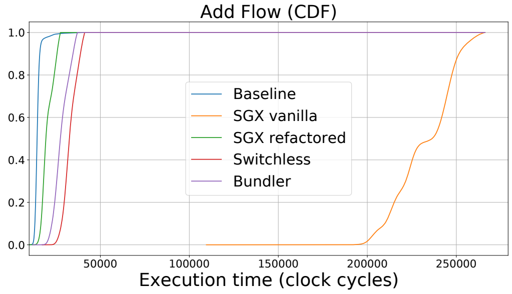


Figure 10: Add flow execution times.

Delete & Modify Flow Rule. The execution time for the delete and modify flow operations is presented in Figures 11 and 12. Here we see a similar pattern: *SGX refactored* has the best performance after baseline, followed by *Bundler* and *Switchless.* Note that unlike add flow, delete and modify often target multiple table entries. We intend to examine the effect of batching in these operations in a follow-up paper.

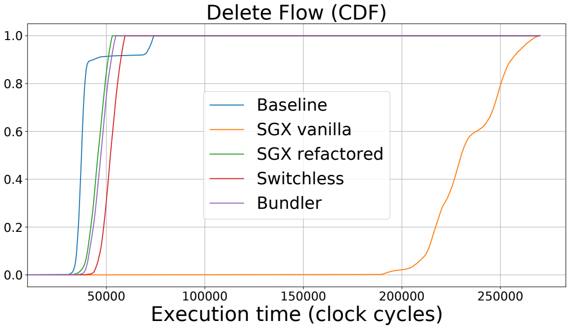


Figure 11: Delete flow execution times.

Evict Flow Rule. The eviction operation requires a very high number of enclave transitions, which as seen in Figure 13 favors solutions optimized to minimize transition cost. Given these results, while *Bundler* introduces a measurable performance overhead it does not drastically increase execution time even in corner cases.

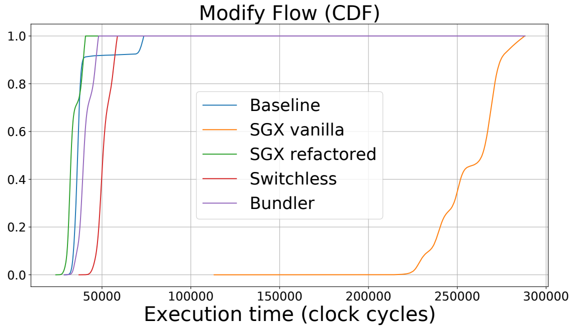


Figure 12: Modify flow execution times.

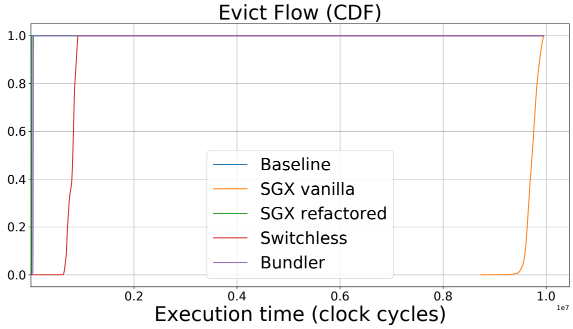


Figure 13: Evict flow execution times.

6 RELATED WORK

Software-Defined Networking (SDN) and in particular the SDN control plane has been extensively scrutinized by the security research community [28],[2]. Some researchers considered the use of Trusted Computing and trusted execution to address security issues. For example, Jacquin *et al.* proposed using TPM to ensure a trusted boot and use of attestation to monitor the integrity of flow tables [10], while Paladi *et al.* suggested using Intel SGX to ensure a secure boot and to provide secure communication channels [22]. Similarly, Shih *et al.* proposed executing parts of a virtual network function inside an Intel SGX enclave [27].

Medina *et al.* proposed *OFTinSGX*, an OvS implementation where OpenFlow flow tables are placed inside an SGX enclave [21]. While this provided confidentiality and integrity guarantees to flow tables, it introduced a significant performance degradation to OvS.

Performance issues in SGX applications can sometimes be attributed to the high cost of entering and exiting enclaves. Weisse *et al.* introduced "*HotCalls*" for communicating with enclaves using shared untrusted memory [32]. This approach can be orders of magnitude faster than ECALLs, although the use of untrusted memory increases the enclave attack surface. The switchless enclave function call component of the *HotCall-Bundler* library developed in this paper is heavily inspired by this work.

The HotCalls protocol requires an enclave worker thread that communicates with the main thread through a shared memory region. This thread will occupy one CPU core, which is economical only when the SGX enclave is under some load. Tian *et al.* suggested using an adaptive approach where ECALLs are used when the device is mostly idle and switchless calls are used when it is under some load [29]. This scheme has been included in recent versions of the Intel SGX SDK as a mainline feature. We chose to not use

[2] Source code repository: https://anonymous.4open.science/r/ovs-sgx/

this scheme in our paper as it lacked the flexibility and control granularity of a custom solution.

7 CONCLUSIONS

In this paper, we described HotCall-Bundler, a mechanism to help improve the performance of IO-intensive applications in Intel SGX enclaves. HotCall-Bundler combines switchless SGX communication and novel optimization using execution graphs and function memoization. We extended earlier work and developed two prototypes using switchless communication both with and without execution graphs and memoization. We evaluated the performance improvements introduced by the HotCall-Bundler library in Open vSwitch, a widely used network switch implementation.

The HotCall-Bundler library can be used for other IO-intensive applications that can benefit from the security guarantees of isolated execution in SGX, such as in-network packet processing, biological sequence analysis or long-running simulations. Considering the many parameters that each evaluation entails, we will explore in future work the performance effects of the HotCall-Bundler library in other IO-intensive applications, as well as evaluate the required programming efforts across several case studies.

8 ACKNOWLEDGEMENTS

This work was financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0035.

REFERENCES

[1] [n.d.]. *Intel Software Guard Extensions Programming Reference.* Technical Report. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf

[2] A. Abdou, P. C. van Oorschot, and T. Wan. 2018. Comparative Analysis of Control Plane Security of SDN and Conventional Networks. *IEEE Communications Surveys&Tutorials* 20, 4 (2018), 3542–3559. https://doi.org/10.1109/COMST.2018.2839348

[3] Markku Antikainen, Tuomas Aura, and Mikko Särelä. 2014. Spook in your network: Attacking an SDN with a compromised openflow switch. In *Nordic Conference on Secure IT Systems*. Springer, 229–244.

[4] Roberto Bifulco, Heng Cui, Ghassan O Karame, and Felix Klaedtke. 2015. Fingerprinting software-defined networks. In *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*. IEEE, 453–459.

[5] Daniel Genkin and Yuval Yarom. 2021. Whack-a-Meltdown: Microarchitectural Security Games [Systems Attacks and Defenses]. *IEEE Security & Privacy* 19, 1 (2021), 95–98.

[6] C. Göttel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin, and V. Schiavoni. 2018. Security, Performance and Energy Trade-Offs of Hardware-Assisted Memory Protection Mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 133–142. https://doi.org/10.1109/SRDS.2018.00024

[7] C. L. Hamblin. 1962. Translation to and from Polish Notation. *Comput. J.* 5, 3 (11 1962), 210–213. https://doi.org/10.1093/comjnl/5.3.210 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/5/3/210/1172943/5-3-210.pdf

[8] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society. https://doi.org/10.14722/ndss.2015.23283

[9] Intel. 2015. *Benefitting Power and Performance Sleep Loops*. https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops

[10] Ludovic Jacquin, Adrian Shaw, and Chris Dalton. 2015. Towards trusted software-defined networks using a hardware-based Integrity Measurement Architecture. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. https://doi.org/10.1109/NETSOFT.2015.7116186

[11] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD memory encryption.* White paper. Advanced Micro Devices, Inc.

[12] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2017. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 145–161. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kim-seongmin

[13] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. 2015. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (Philadelphia, PA, USA) *(HotNets-XIV)*. Association for Computing Machinery, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/2834050.2834100

[14] Jon Kleinberg and Eva Tardos. 2005. *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[15] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[16] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 203–216. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen

[17] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1257–1272. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[19] Taous Madi, Suryadipta Majumdar, Yushun Wang, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. 2016. Auditing Security Compliance of the Virtualized Infrastructure in the Cloud: Application to OpenStack. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 195–206. https://doi.org/10.1145/2857705.2857721

[20] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. ACM, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/2487726.2488368

[21] J. Medina, N. Paladi, and P. Arlos. 2019. Protecting OpenFlow using Intel SGX. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 1–6. https://doi.org/10.1109/NFV-SDN47374.2019.9039980

[22] Nicolae Paladi and Christian Gehrmann. 2017. Bootstrapping trust in software defined networks. *ICST Transactions on Security and Safety* 4 (12 2017), 153397. https://doi.org/10.4108/eai.7-12-2017.153397

[23] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. 2009. Extending networking into the virtualization layer.. In *Hotnets*.

[24] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. 2016. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference* (Trento, Italy) *(Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/2988336.2988346

[25] Xiao Qin, Hong Jiang, Adam Manzanares, Xiaojun Ruan, and Shu Yin. 2009. Dynamic load balancing for I/O-intensive applications on clusters. *ACM Transactions on Storage (TOS)* 5, 3 (2009), 1–38.

[26] M. Schwarz and D. Gruss. 2020. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security Privacy* 18, 5 (2020), 18–27. https://doi.org/10.1109/MSEC.2020.2993896

[27] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization* (New Orleans, Louisiana, USA) *(SDN-NFV Security '16)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/2876019.2876032

[28] Zhaogang Shu, Jiafu Wan, Di Li, Jiaxiang Lin, Athanasios V. Vasilakos, and Muhammad Imran. 2016. Security in Software-Defined Networking: Threats and Countermeasures. *Mobile Networks and Applications* 21, 5 (01 Oct 2016), 764–776. https://doi.org/10.1007/s11036-016-0676-x

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

[30] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/3185467.3185469

[31] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[32] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 81–93. https://doi.org/10.1145/3140659.3080208

[33] Jiewen Yao and Vincent Zimmer. 2020. *Virtual Firmware*. Apress, Berkeley, CA, 459–491. https://doi.org/10.1007/978-1-4842-6106-4_13

[34] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng. 2020. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1450–1465. https://doi.org/10.1109/SP40000.2020.00054