TEEshift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs

Titouan Lazard
University of Rennes 1
ENS Rennes
France

Johannes Götzfried
Friedrich-Alexander University
Erlangen-Nuremberg
Germany

Tilo Müller
Friedrich-Alexander University
Erlangen-Nuremberg
Germany

Gianni Santinelli
Tages SAS Solidshield
France

Vincent Lefebvre
Tages SAS Solidshield
France

Abstract

We present TEEshift, a tool suite that protects the confidentiality and integrity of code by shifting selected functions into TEEs. Our approach works entirely on binary-level and does not require the adaption of source code projects or build environments, nor does it require compiler-level patches. Programmers provide a list of ELF symbols pointing to the functions that should be protected. After post-processing an ELF binary with TEEshift, the selected functions are not present in cleartext anymore. Only after attesting to a remote party that the loading enclave behaves with integrity, the functions are decrypted, but remain inside the enclave protected against reverse engineering. An online connection is only required when a program starts for the first time on a PC. Afterwards, sealing is used to securely store the decryption key and bind it to the PC. By allowing programmers to move selected function into TEEs, without patching their source code, we provide a convenient way to enable TEEs in existing projects while preserving the flexibility for a finegrained security and performance tradeoff. We evaluated our tool using a real world gaming application, confirming the practicability of our approach for existing projects. We overcome the limitation of the fragmented TEE landscape by building on top of Asylo, an open framework by Google for apps which aim to support different TEEs such as Intel SGX and AMD SEV using a unified API.

SysTEX '18, October 15, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5998-6/18/10…$15.00
https://doi.org/10.1145/3268935.3268938

1 Introduction

Traditional software protection solutions rely on obfuscation to increase the time an adversary needs to reverse engineer the functionality of a software product. All means of obfuscation, however, only raise the bar for attackers but will eventually be broken. To overcome the limitations of obfuscation, and to provide verifiable security under certain assumptions, emerging technologies such as Intel SGX [5] and AMD SEV [6] provide *Trusted Execution Environments* (TEEs) on commodity hardware [7]. Using TEEs, code can be decrypted inside or dynamically loaded into an enclave only after it has been proven to a remote server that the enclave behaves with integrity and thus, does not permit any means of reverse engineering.

Before SGX and SEV became widely available, the *Trusted Platform Module* (TPM) has long been the only hardware supported trust anchor on commodity hardware. The TPM's application for game copy and DRM protection, however, was limited and never gained acceptance in scenarios other than full disk encryption. The reason lies the TPM design which provides a rather *Static Root of Trust* (SRoT), enabling the verification of a PC's boot process but not of its processes or VM instances. SGX and SEV for the fist time provide support for *dynamically* placing software inside so-called *enclaves*, relying on a *Dynamic Root of Trust* (DRoT).

Intel and AMD proposed different approaches for their DRoTs in the recent past that cannot be used interchangeable. While SGX focuses on a per-process RoT, aiming at DRM, SEV focuses on per-VM RoT, aiming at cloud computing. In both cases, developers are posed with the challenge of how to integrate their legacy code base. Today, code which is to be run within a TEE needs to be prepared and rewritten for the targeted hardware architecture. A task which is complicated by restrictions regarding the instruction set that is allowed inside an enclave, restrictions on the enclave size, and also regarding the library support within enclaves.

1.1 Our Contribution

Providing a legacy app for both Intel SGX and AMD SEV is often associated with unreasonable expense as it exceeds the knowledge of most developers. Our solution was motivated by the question of how to protect an existing code base in a platform-independent manner using post-processing approaches to not require developers to adapt their source code or build environment. With TEEshift, we provide a solution that selectively shifts functions of ELF binaries into TEEs, relying on the Asylo framework to provide platform independence.

In detail, our contributions are:

- To provide code confidentiality of functions, we remove them from the binary in cleartext and decrypt them only inside an enclave after its integrity has been attested to a remote server owned by the software provider. A three-step loading process ensures that an online connection is only required on the first start of a program, afterwards the functions are bound to the CPU by means of sealing.
- We build our solution on top of Asylo, an open framework by Google for enclave applications which aims to support different TEEs such as Intel SGX, AMD SEV, or even TrustZone with a unique interface.
- We use ELF-rewriting to shift programmer-specified functions into TEEs and to redirect the control flow of those functions to their enclaved counterpart. We do not require source code, build environment or compiler patches.
- We evaluated our tool suite using a real world gaming app regarding practicability and performance, confirming that our tool can be used together with existing legacy code bases.

1.2 Related Work

TEEs, in particular Intel SGX, have been mentioned in publications about cloud computing and in the context of software protection multiple times. Haven [3] was designed to securely run unmodified legacy apps, while VC3 [9] offers distributed *Map-Reduce* computations that keep the data being processed hidden from cloud providers. Both solutions, however, did not use real hardware but an SGX emulator provided by Intel, and ignored the complexity of remote attestation in practice. Scone [1] introduced entirely isolated Linux systems by augmenting Docker containers with SGX and Graphene-SGX [11] provided a library operating system for unmodified applications on top of SGX. Compared to our solution, the TCB for Scone and Graphene-SGX, however, is typically larger since whole containers or a library operating system are protected. Opaque [13] offers confidentiality for database queries in particular by placing parts of a database within an SGX enclave. Opaque exclusively aims to secure queries to a SQL database instead of arbitrary programs. Most related to TEEshift is our industry solution Cloudshift [10] which shifts functions that should remain confidential into the cloud or a co-located processor. While Cloudshift is an effective approach compared to obfuscation, it requires a permanent online connection. Contrary to our new approach, Cloudshift uses a code interpretation logic. To the best of our knowledge, TEEshift is the first solution of selectively protecting binary functions without relying on obfuscation or requiring a permanent online connection while maintaining a small TCB.

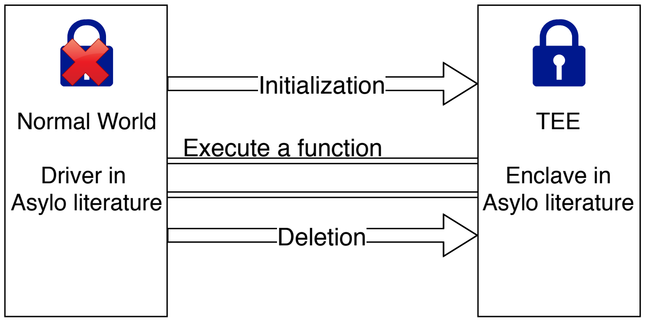

Figure 1. Asylo interface with the unprotected normal world (driver) and the protected TEE (enclave).

2 Background: Asylo

Asylo [8] is a recent framework by Google which aims to provide a unified API for different TEE hardware implementations. It is distributed under the Apache Licence 2.0. Asylo is in an early stage of development and does not support *real* TEEs at the moment. Among the currently supported backends, the simulation mode of the Intel SGX SDK can be found and the Asylo developers claim to support new backends such as AMD SEV in the near future. One of the core features of Asylo is that it abstracts from all the different TEE interfaces and management issues and provides a unique TEE API to its users.

An Asylo project can be divided into two parts. As shown in Figure 1, on the one hand there is the normal world, also called *driver*. The driver is unprotected, runs outside of the underlying TEE, and is responsible for managing, initializing, and deleting enclaves. On the other hand, there is the TEE, also called *enclave*, which is protected by the underlying trusted computing hardware mechanism. The design of Asylo only allows one function of the enclave to be exposed and called from the outside, that is from the driver. Furthermore, Asylo does not allow an enclave to directly call functionality provided by the outside world or the driver. If an enclave needs to call to the outside world, it has to use a special *Remote Procedure Call* (RPC) system provided by Asylo for both sides, the driver and the enclave.

Generally, the RPC system consists of two parts, a server and a client. To communicate, the server first needs to register a function to be callable and then waits for requests to arrive on a particular channel. Afterwards, the client is able to send requests to the server together with well defined arguments, waits for the server to complete the request, and then gets back the result. To successfully establish a connection, client and server need to agree on a channel as well as the formats of requests and responses. A channel can be a network socket, for example. RPCs in Asylo are implemented using *gRPC*, which relies on *Protobuf* to format the request and the arguments. Both projects *gRPC* and *Protobuf* are Google projects, as well.

3 Design and Implementation

In this section, we present the design and implementation of our tool suite. We first give a general overview over the TEEshift architecture (Sect. 3.1) and then point out important implementation details (Sect. 3.3).

3.1 TEEshift Architecture

The basic idea of TEEshift is to automatically strip functions from an existing binary and to run them within a given TEE. In the following, to the functions we want to protect by running them within a TEE, we will refer to as *target functions*. The binary, that is about to be protected is called the *target binary*.

To be able to run arbitrary programmer-specified functions within a TEE, we use a multi-step process. First, the code of the target functions needs to be analyzed such that the corresponding Asylo driver and enclave can be built. The pair consisting of driver and enclave is specific to every target binary. Second, the code of the target functions must be modified and added to the enclave code. Finally, the Asylo driver is compiled as a shared library and the target binary is patched to replace every call to the target function with a call to the dispatcher within this library.

Figure 2 shows how the multithreaded architecture looks like after functions have been shifted to Asylo enclaves. Calls to protected functions are replaced by a call to the dispatcher within our shared library. The library then redirects the call through gRPC to the enclave where the code is executed. Afterwards the results are passed back to the patched binary again using gRPC. In case a protected function needs to execute functionality from the original binary, we also support callbacks to the patched binary.

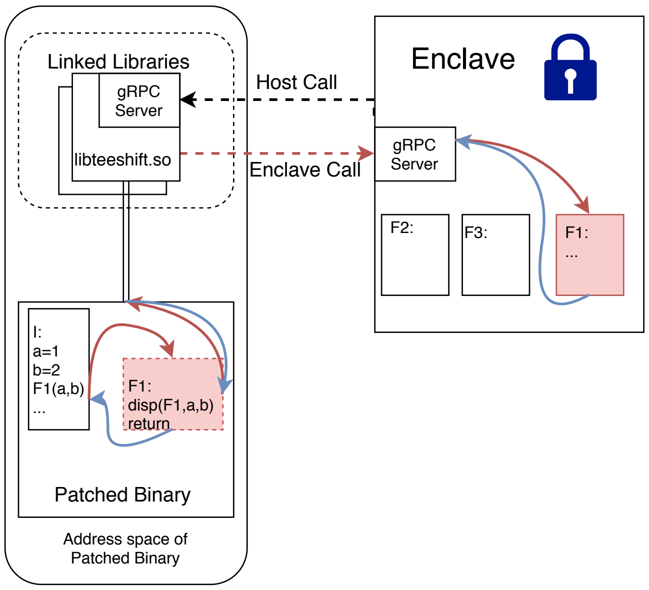

Figure 2. Overview of the TEEshift architecture. Calls to protected functions are redirected to the dispatcher within our shared library containing Asylo. They are then forwarded to the enclave thread using gRPC.

3.2 Confidential Code Loading

Shifting functions into TEEs effectively protects the integrity of code by means of hardware support. To additionally ensure code confidentiality, the original unencrypted code must be stripped from the binary and the transformed enclave code is stored only encrypted. The code is decrypted during runtime if and only if the loading enclave has attested to a remote server that it behaves with integrity.

On the first start of a protected app, the decryption key is fetched from a trusted remote server, e.g., a licensing server. The server will only reveal the key if the loading enclave has successfully attested to the licence server that it is in the correct state. Once the enclave received the key it will use Asylo's sealing capabilities to seal the key and store it on the hard disk. The key is sealed in such a way that it can only be unsealed on the very same machine, i.e., it is bound to the platform.

For each subsequent start of the protected app, the key is unsealed and used to decrypt the transformed code. Hence, an online connection is only required during install time and not permanently. For multiple function invocations during the same execution, no unsealing is required, because the code resides decrypted to an enclave in memory and is ready to be executed.

In addition, all client server communication is encrypted to prevent untrusted parties such as other apps or the OS from retrieving the decryption key or decrypted code.

3.3 Implementation

In this section, we describe the implementation of TEEshift and discuss selected problems that were particularly challenging to solve.

Shared library loading A dynamically linked library, containing Asylo and all shifted functions, must be loaded into the address space of the target binary at start time. Modifying an ELF binary to load a new library is non-trivial, as newly introduced code falsifies following addresses and offsets. Thus, instead of editing the binary, we use the options provided by the dynamic loader for Linux. Specifically, we use the environment variable `LD_PRELOAD` to force the loader to load our shared library before any other library. Note that more information about the library loading process, and the external function address resolving can be found in a paper by Chamberlain and Taylor [4].

Control flow redirection After loading the shared library, we need to redirect every call to a target function to the enclave. We achieve this with a technique called *PLT hooking*.

To call functions of an external library, an ELF file creates one entry per imported function in the *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT), respectively. The idea of PLT Hooking is to declare a function within the library loaded with `LD_PRELOAD` which is then imported by the target binary. When the dynamic loader resolves the address of the imported function at runtime, it resolves the one of our shared library instead of the original one. Every call to this imported function will then automatically result in a call to our function.

We replace the body of all target functions with jumps to imported functions from our shared library, which then jump to a dispatcher routine. Each time before jumping into our shared library, we set the `r11` register to an identifier of the target function. Based on the value of `r11`, the dispatcher routine either redirects the control flow to the enclave or executes the original function. To redirect the control flow to the enclave, the library makes requests to the gRPC server running within the enclave. The gRPC request is received by the server which then calls the requested function present in its address space. The return value is computed and returned to the gRPC server, which sends it back to the gRPC client waiting for the return value in the shared library.

Another challenge when working on binary level is to determine how many arguments are passed to the target functions. To avoid static binary analysis, we decided to refer to the standard calling convention on x86_64 Linux systems and to always copy the first 5 arguments (registers `RDI`, `RSI`, `RDX`, `RCX`, and `R8`) to the enclave, also if less than 5 arguments are used. Note that functions with stack parameters, i.e., with more than 5 arguments, are currently not supported.

Call from the enclave to the binary Only a very limited number of functions, so called leaf functions, do computations without calling other functions themselves. Thus, a standard case is that the programmer needs to shift a function which calls functions that should not be shifted to the TEE. Thus, to be able to shift arbitrary functions, we need to find a way to handle calls from enclaved functions to non-enclaved functions of the original binary. When rewriting the code of a target function, we replace every call to a non-enclaved function of the binary by a call to the gRPC client running inside the TEE, which is responsible for transferring the request to a server running in the shared library and wait for its result. This server is not running inside the TEE and hence, can directly call non-enclaved functions of the binary.

Pointer dereferencing Oftentimes functions have to dereference pointers that are passed as arguments and point to arbitrary memory locations. Due to our multithreaded setup, it is not possible to dereference a pointer from inside the TEE which points to the outside world. Thus, we replace every pointer dereferencing to the outside world by a function in the driver that reads or writes to the target address. Those calls to the driver are done using gRPC.

Handling PIE binaries For security reasons, today binaries are often compiled as *Position Independent Executable* (PIE) to randomize the base address of a binary at load time. In PIE binaries, references to global variables and function calls are computed relatively to the instruction pointer `IP`. When shifting a function into the enclave, relative references based on the `IP` are changed to the value they have if the functions were executed within the binary.

Accessing the TEE memory The TEE and the original binary are not running in the same address space. Thus, it is not possible to dereference a pointer inside the TEE which points into the original address space, and vice versa. Unfortunately, it is not possible to rewrite the body of every function in the binary which could at any point in time dereference a pointer pointing to the TEE address space. Also, it is not possible to offer functions the ability to arbitrarily read and write into the TEE address space without compromising the confidentiality of the code.

Thus, we currently do not support direct access from unprotected code to protected functions inside TEEs. Cloudshift [10], for example, has the same technical limitation, i.e., code on the local side can not directly access code on the remote server, and also here, those issues are prevented by a preliminary thorough static code analysis.

4 Evaluation

In this section, TEEshift is evaluated regarding its performance and practicability. To this end, we used TEEshift to protect a real world gaming app called Teeworlds [2] (which has nothing to do with TEEs).

4.1 Performance Considerations

There always is a tradeoff between security and performance. Since instead of regular function calls, we have transitions between a TEE and the original binary, including communication via RPCs, performance is certainly affected. However, the performance drawback for practical apps depends on the selection of functions that are protected.

In general, a call to a protected function is replaced by multiple calls, a client/server communication and possibly context switching. In our lab setup, we found out that each call to a function inside an enclave adds around 0.5ms. This overhead is constant per call, however, and it does not depend on the function size, function code, or call frequency. Furthermore, modifications applied to the code of target functions can imply performance overhead, too, but is hard to evaluate since it depends on the function code. Finally, we have to add the performance differences between code running inside and outside of the underlying TEE itself. This performance overhead fully depends on the TEE hardware and cannot be influenced by us. For example, a code running inside an SGX enclave suffers from approximately 10% overhead compared to native code [12].

While the performance overhead can increase notably depending on which functions are getting protected, we show that for practical use cases, like the Teeworlds gaming app, the overhead is not even noticeable when selecting a reasonable set of functions.

4.2 Case Study: Teeworlds

We chose Teeworlds as an evaluation example for several reasons. First, games have a reasonably large code base to check that our approach works on legacy projects. Second, performance is typically important for games and thus, we can show how performance is impacted *practically* by our tool. Finally, games are the number one example where code confidentiality is an issue in the form of *Digital Rights Management* (DRM). Shifting functions into TEEs to replace or support existing DRM schemes seems like a promising research field.

We chose to protect *Teeworlds* [2], a 2D shooter multiplayer game written in C++. Besides the nice pun of TEEshift protecting Teeworlds, Teeworlds is a middle size open source project with dependencies to SDL and other real libraries.

How to choose the target functions to shift? To get the best tradeoff between security and performance, we tried to find a set of functions which meets the following criteria:

- The functions should not be called too frequently to not affect the interaction with the game logic very much. In terms of frequency, ideal functions are called too often to be able to execute the program without them, but not too rarely to notably impact the performance of the gameplay.
- The functions are not allowed to pass pointers pointing inside TEE memory to functions outside of the TEE, due to the limitations of TEEshift mentioned above. Note, however, that this technical limitation is not a semantic limitation in most cases. From a security perspective, pointers to enclave memory should also not be returned to non-enclaved code, in order not to leak protected data to the outside world.

Functions that initialize an intern state of the game seem to be adequate targets. In Teeworlds, multiple functions meet this criteria and were selected for shifting. As an exemplary target function, we have a look to the member function `Add()` of the `ServerBrowser` object. The `ServerBrowser` object is responsible for managing different server entries. A player of Teeworlds has to use the server browser to choose and manage which game server to join. In detail, a `ServerEntry` object is added to the server browser list of server entries for each game server the player can join. By placing the code of the `Add()` method of the server browser, we can totally hide its code. To be able to run Teeworlds without a valid licence, an attacker would have to reconstruct the code necessary to launch the game including the `Add()` method or avoid making calls to this method altogether. Then, however, the attacker would not be able to join remote servers rendering the game pretty much useless.

During our evaluation, launching Teeworlds and using the server browser to choose a server to join resulted in 812 consecutive calls to the enclaved `Add()` method. In the same run, we noted 4345 calls from the enclave back to the binary. Note that those numbers may vary per run, as the exact number of calls is dependent on the servers that are reachable. All these calls, however, were completely transparent to the end user and did not have noticeable performance impacts, nor impacted the functionality of the game.

Besides `Add()`, we experimented with shifting other target functions to an enclave. Whenever carefully choosing the target function, we achieved similar performance results as for `Add()`. Of course, without restrictions it is possible to shift multiple functions into an enclave at once. The more code is run in an enclave the more code an attacker has to reconstruct.

5 Conclusion and Future Work

To conclude, we give an outlook over future research directions that may continue our work.

5.1 Future Work

While TEEshift is an early prototype implementation, it proves the practicability of our approach. As such it can be used as a base for further research directions.

Currently, we guarantee the integrity and confidentiality of target functions. To this end, programmers select a number of functions and TEEshift lets them transparently run inside a TEE. In future, we want to extend this concept to data and not focus on code only. TEEshift then could, for example, ensure that a set of programmer-selected variables is stored and processed securely by a TEE. Static code analysis would have to be used to make sure that no data passed from the TEE to the outside world leaks content or parts of the content of secured variables.

One of the key design choices behind TEEshift is to provide simplicity for developers, particularly when porting parts of a legacy code base to TEEs. To make TEEshift even more user-friendly, a list of possible target functions worth to protect could be generated automatically based on heuristics that take the tradeoff between security and performance into account. Code complexity metrics and static code analysis would have to be used to determine functions being both worth to protect and called rarely enough to not impact the gameplay much.

Last but not least, we are aiming to integrate TEEshift, currently having a Linux/ELF focus as a research project, with our industry solution Cloudshift for Windows/PE binaries. We want to continue using Cloudshift's UI that enables the user with an interface to select the portion of code to be executed in a trusted and safe environment. But instead of moving those functions to a remote machine, we are planning to wrap TEEshifts's Asylo-based driver and move them into a TEE.

5.2 Conclusion

We presented TEEshift, a tool suite that protects the integrity and confidentiality of code by shifting selected functions into TEEs. While there is room for improvements, e.g., regarding assistance in the choice of the target functions to shift, TEEshift shows it is possible to give programmers a secure alternative to current obfuscation-based packer approaches. While obfuscation-based binary packing can only prolong the analysis time, with TEEshift the set of target functions remain a blackbox to reverse engineers — under the assumption that Intel SGX and AMD SEV cannot be broken. As a consequence, attackers have to analyze enclaved functions based on their I/O behavior and to reconstruct them from scratch and emulate their behavior. We argue that this is typically much harder than breaking obfuscation when the target functions were chosen wisely.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and by the CELTIC+ SENDATE project "TAilored Network for Data cEnters in the Metro" (TANDEM).

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI.*

[2] Magnus Auvinen. 2018. Teeworlds: A retro multiplayer shooter. (2018). https://www.teeworlds.com

[3] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI.*

[4] Steve Chamberlain and Ian Taylor. 2009. The GNU linker ld. (2009). https://www.eecs.umich.edu/courses/eecs373/readings/Linker.pdf

[5] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016). http://eprint.iacr.org/2016/086

[6] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD Memory Encryption.* Technical Report. AMD. http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[7] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. 2017. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. In *IEEE Transactions on Computers 67*. 361–374.

[8] Nelly Porter, Senior Product Manager Google Cloud. 2018. Asylo: an open-source framework for confidential computing. (2018). https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html

[9] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA*. IEEE Computer Society, 38–54. https://doi.org/10.1109/SP.2015.10

[10] Tages SAS Solidshield. 2018. Cloudshift: state-of-the-art code remotization technology. (2018). https://www.solidshield.com/software-protection-and-licensing/cloudshift/

[11] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference.*

[12] David Übler, Johannes Götzfried, and Tilo Müller. 2017. Secure Remote Computation using Intel SGX. In *Sicherheit, Schutz und Zuverlässigkeit (SICHERHEIT 2018), Bonn.* Gesellschaft für Informatik (GI).

[13] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI.*