H. Langweg, M. Meier, B.C. Witt, D. Reinhardt (Hrsg.): Sicherheit 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 209

Secure Remote Computation using Intel SGX

David Übler[1], Johannes Götzfried[1], Tilo Müller[1]

Abstract: In this paper, we leverage SGX to provide a secure remote computation framework to be used in a cloud scenario. Our framework consists of two parts, a local part running on the user's machine and a remote part which is executed within the provider's environment. Users can connect and authenticate themselves to the remote side, verify the integrity of a newly spawned loading enclave, and deploy confidential code to the provider's machine. While we are not the first using SGX in a cloud scenario, we provide a full implementation considering all practical pitfalls, e.g., we use Intel's Attestation Services to prove the integrity of the loading enclave to our users. We also take care of establishing a secure bidirectional channel between the target enclave and the client running on the user's machine to send code, commands, and data. The performance overhead of CPU-bound applications using our framework is below 10% compared to remote computation without using SGX.

Keywords: Intel SGX; Cloud Computing; Isolation

1 Introduction

Cloud services are a promising way for companies to lower their expenses for building and maintaining IT infrastructures. A recent survey shows that a majority of companies already shifted some of their workload to cloud providers [ri16] while the market share of cloud services is still growing rapidly. However, providers of these services often cannot be trusted and any promises on confidentiality made by them cannot be enforced by the user.

If a company wants to retain confidentiality and integrity of its code and data, the cloud provider must be prevented from accessing resources while still being able to execute them. This includes, for example, main memory and disk space dedicated to a user's task. In 2013, Intel introduced the Software Guard Extensions (SGX) [Ho13] which are a trusted computing architecture offering isolated memory regions and restricting access to authorized parts of an application. Software attestation can be used to verify those isolated regions by a remote user before secret code and data is transferred to the cloud.

The drawback of using SGX, however, is that a company must explicitly divide its application in trusted and untrusted parts, and set up an attestation scheme. This requires additional implementation effort and imposes a dependency on SGX, as the layout of the program must suit the constraints of SGX. Fulfilling these constraints and correctly designing an application to use SGX quickly becomes nontrivial and leaves ample room for mistakes, and thereby vulnerabilities [G7]. Thus, our approach is to provide an environment to users

[1] Department of Computer Science, FAU Erlangen-Nuremberg, Martensstr. 3, D-91058 Erlangen, david.uebler@posteo.de, {johannes.goetzfried,tilo.mueller}@cs.fau.de

doi:10.18420/sicherheit2018_16

that lets them deploy and execute arbitrary code which can conveniently be protected by the security mechanisms of SGX.

1.1 Our Contribution

Although proposals for SGX-based cloud solutions already exist [BPH14, Sc15], we provide the full implementation of a remote computation framework relying on SGX while considering all practical pitfalls including Intel's Attestation Services. In detail our contributions are:

- Our framework consists of two parts, a local part running on the user's machine and a remote part which is executed within the provider's environment. Users can connect and authenticate themselves to the remote side, verify the integrity of a newly spawned loading enclave, and deploy confidential code at the provider's machine.
- The loading enclave engages in our remote attestation protocol by generating a report containing the enclave's measurement and information needed to establish an encrypted connection to the user. The report is wrapped into a quote with the help of the quoting enclave provided by Intel. This quote is then verified by the user using Intel's Attestation Services.
- If the integrity of the loading enclave can be verified by the user, a bidirectional encrypted connection between the enclave and the user is established to transfer code, commands, and data.
- The performance overhead of CPU-bound applications using our framework is below 10% compared to remote computation without using SGX.

1.2 Related Work

Intel SGX has been mentioned in publications about cloud computing multiple times before. The trusted execution system Haven [BPH14] was designed to securely run unmodified legacy applications, while VC3 [Sc15] 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 [Ar16] introduced entirely isolated Linux systems by augmenting Docker containers with SGX. Software attestation, however, was also not implemented in Scone. Opaque [Zh17] offers confidentiality for database queries in particular by placing parts of a database within an SGX enclave. So Opaque exclusively aims to secure queries to a SQL database instead of arbitrary programs.

2 Background: Graphene

Graphene [TPV17] is a *library operating system*, that is a library that provides all abstractions and the same environment usually provided by an OS. An application running in this environment can issue system calls and access a filesystem, but the resources exposed to it are not the host's resources. They are rather virtual resources provided by the library OS. As with a virtual machine, all abstractions provided by the library OS do not have to be identical to the abstractions of the host OS. This means that emulating another kernel version, or an entirely different operating system is possible.

Isolation through a library OS can be achieved by reimplementing all system calls and providing a virtual filesystem. System calls do not trap to the host OS, but are rooted in functions within the library OS. Those functions can handle system calls directly, or use the host OS's facilities to handle them. As most software does not issue system calls directly, but uses a standard library, redirecting system calls can usually be done by just modifying the standard library to call the library OS. A filesystem is reached via system calls as well, enabling the library OS to either redirect it to the host, or to use a virtual filesystem.

Graphene was initially designed to protect host systems against malicious guest applications by isolating the guest through virtual system calls. A similar scheme was later added to offer protection against the host to the guest, leveraging SGX.

To protect a guest application, it is entirely executed inside an enclave, including all libraries used and the library OS. This gives the guest's virtual address space the memory protection guarantees provided by SGX. However, a key feature of Graphene is offering the environment of an operating system to allow system calls, file access and dynamic loading. These actions require the guest to communicate with the outside world, receiving results of system calls and data from files controlled by the host. To reduce the risk posed by these outside influences, an additional protection layer is added. The purpose of this layer is to evaluate the host's responses given to system calls and either allow or reject them.

3 Design and Implementation

This section describes the architecture of our framework, the protocol spoken by the local and remote part, as well as selected implementation details.

3.1 Architecture

The overall architecture of our solution follows a client-server model. The client and server can (and should) be executed on different machines connected by a network. The client is responsible for sending code and data to the server, which is loaded into an enclave at runtime on the server side. Once loaded, the client can establish a connection with the deployed code directly.

The parties participating in the described scenario are the *user* and the *provider*. The user is a person, institution, or machine that aims to execute some of its application code using the provider's resources. The provider yields some of its computational resources, such as CPU time, memory and disk space to the user. To this end, the user's machine runs the client code while the provider's machine executes the server code. This distribution splits the whole application into two parts. One part, the user's machine and software, is called the *local side*. The other part, the provider's machine and software is called the *remote side*. Both parts are communicating over a network, which is not considered to be part of either side or in any way secured.

The user has full control over the client software and the machine it is running on. He or she can therefore trust the client side. The remote side, however, is under control of the provider and can not be trusted by the user. From the provider's point of view, the situation is reversed. It can rely on the remote side, being run on its hardware and operating system, but has no guarantee that the user is not malicious. Both parties consider the network being untrusted and make sure not to expose any sensitive information to it.

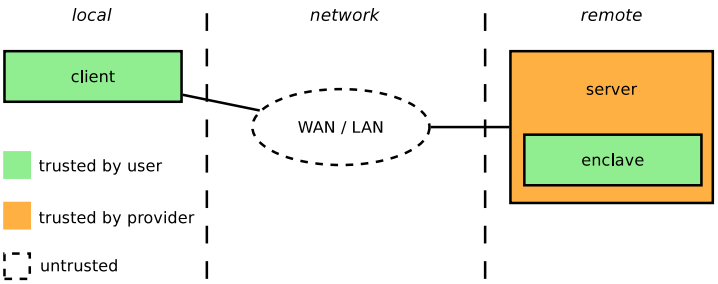

Fig. 1: Remote computation architecture.

While both parties cannot check each others behavior, they can determine the identity of the other side through *attestation*. For example, a guarantee is needed that the provider cannot access the user's code and data after it has been transferred to the remote side. To this end, an SGX *enclave* is running on the provider's machine, alongside the server software. It is at the same time under the user's control and inaccessible to the provider (see Figure 1).

3.2 Protocol

Our protocol has been designed to integrate well into existing networks by building on top of established base protocols. Furthermore, it offers a generic interface to the applications deployed by potential users.

Protocol Stack and State Transitions Our protocol stack consists of different layers. HTTP is used as the base protocol, because it is a broadly accepted protocol and already

used in most existing networks. Using a ubiquitous protocol such as HTTP eliminates the need to reconfigure the network such as opening or forwarding additional ports at routers and firewalls. HTTP messages consist of a header and body, where the body carries the actual payload while the header offers a place to store metadata such as the message's destination and payload length. The body together with the length can be thought of as a *frame*. The layout and contents of those frames is defined by the management layer.

The transitions of the protocol are shown in Figure 2; only client and server take part in the communication. After the connection between client and server has been established, the distributed application is in the *connected* state. In this state, the identities of server and client have not yet been validated and the enclave has not yet been created.

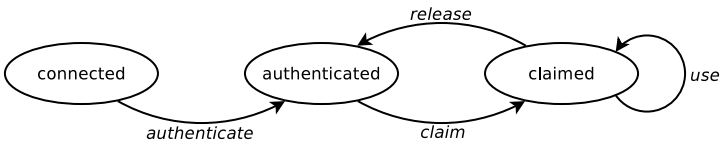

Fig. 2: Protocol states and transitions.

For transition to the next state, the client must authenticate itself to the server. For this transition, a challenge-response protocol based on cryptographic signatures is used. After authentication, the system is in the *authenticated* state. The server now trusts the identity of the client and an encrypted channel is opened between them. Any message arriving on this channel is now guaranteed to have been sent by the client.

When authenticated, the client can request the server to create a new enclave and bind it to the client. This newly created enclave is, however, not usable until it is *claimed*. Claiming consists of the enclave proving its integrity to the client and establishing a second encrypted channel. This second channel is used to deploy the user's code on the enclave. Only after claiming the enclave and sending code, the enclave is within the *claimed* state.

Claiming and Remote Attestation When the server receives a request for a new enclave, it spawns a loading enclave. The loading enclave is not yet bound to a specific client and has an identical layout at the beginning of each claiming process. The initial state has been measured beforehand and is known to both, the client and the server.

Figure 3 shows how SGX remote attestation was adapted to authenticate the loading enclave. In step 1 the client sends a request to the server to launch a new loading enclave. This enclave is spawned in step 2. At this stage the enclave is not yet bound to the client, but to start the binding, the user's public key must be passed to the enclave in step 3.

The public key is part of the secure channel to be established between enclave and client. For this channel to be complete, a keypair and an initial nonce are required as well. The nonce and the keypair is generated in step 4 by the enclave, making the keypair valid only for a

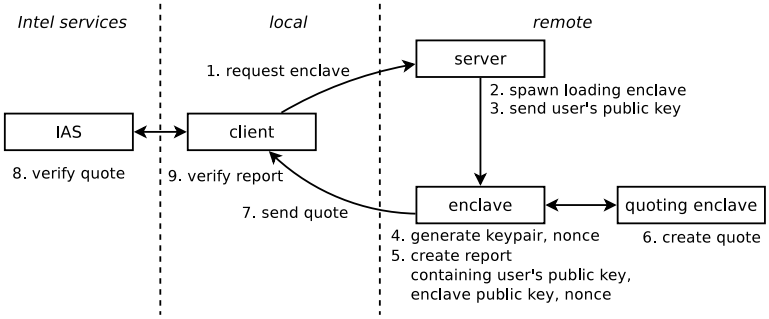

Fig. 3: Remote attestation during enclave claiming.

single session. A permanent keypair is not needed as the enclave's identity is defined by the measurement. Its public and private key have the sole purpose of forming an encrypted channel providing confidentiality, integrity, freshness, and authentication.

The keypair and nonce have to be delivered to the client while at the same time providing proof that the enclave was not tampered with. This proof is given via remote attestation. To this end, the keypair and nonce are embedded in an SGX report, created in step 5. This can be done by copying them to the userdata field of the report. This report is only meaningful on the same platform it was generated. To allow the client to verify the report, a quote is created from it in step 6. This is done by the help of the quoting enclave provided with Intel's SGX SDK.

In step 7, the quote is delivered to the client, which passes it to Intel's Attestation Service, shown in step 8. The attestation service responds to this request with two possible results. If the quote is invalid, the attestation is cancelled and the remote computation session is terminated. Otherwise the client can proceed with step 9, checking the report embedded in the quote. This step can again be done locally without using Intel's services. The report given to the client can now be assumed to be valid. It must, however, be checked if the parameters given in the report are as expected. This is done by comparing the measurement of the enclave to the measurement provided beforehand. Additionally, the user's public key embedded in the report is compared to the one stored locally. By checking the public key, it can be guaranteed that the enclave is bound to the client.

If both values match, the remote attestation and key exchange is complete. A secure channel can be established between both sides using the clients and enclave's keys as well as the nonce. It is important to note that this channel is not accessible to the server. The secret key was computed within the enclave and is thus isolated from the rest of the provider's environment.

Once claimed, the enclave can finally be used. The client is able to send messages to the enclave that are passed to and handled by the user's code. Doing so will cause the system to stay in the *claimed* state. When a user's tasks are complete, and the enclave should no

longer be used, the resources occupied by it will be freed. This is done by issuing a release request to the server. The server will destroy the enclave and the system will fall back to the *authenticated* state. Now, either a new enclave can be requested or the session can be terminated.

3.3 Implementation

We provide an I/O subsystem with basic abstractions to handle all input and output streams used to exchange data between client, server and enclave. These streams are TCP/IP sockets, and on top of a streaming interface, messages are defined as fixed size sequences of bytes, called frames. Our protocol module allows the creation, marshalling and unmarshalling of strongly typed messages to these frames, allowing them to be exchanged using the I/O component.

Encryption is provided on a per-message-parameter basis. Because the server needs partial access to some messages, e.g., the intended recipient of the message, encryption must only be applied to parameters containing sensitive information. To achieve this, the public key authenticated encryption functions provided by the NaCl [Be17] library are used.

An API to be used by client applications and the remote code is provided in C and C++. This allows using the remote computation system with any language that can call C functions, while providing a more comfortable class based interface when using C++.

Graphene as an SGX Environment The enclave is executed in an environment that is isolated from its host in two ways. It has only limited access to the host's resources, by allowing only safe system calls and by having only partial access to the filesystem. To implement these requirements, a sandbox is used to protect the host's system and SGX is used to protect the user's data. With Graphene, a framework is available that provides both. Graphene implements its own virtual filesystem and limits the hosted application to it. It also installs a system call filter that can be configured to allow only certain system calls and with recent versions of Graphene, SGX support is available.

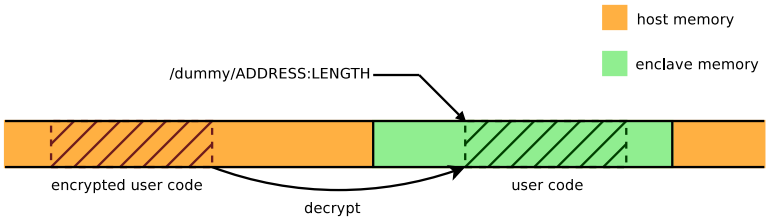

Fig. 4: Pointer filesystem to refer to enclave memory.

Setting up an SGX enclave and performing system calls is handled by Graphene without using SGX functions. Nonetheless, care has to be taken when library calls and system calls

are executed, because data may be copied between enclave and host memory. One such situation arises when code is loaded at runtime. Loading code can be done by using the `dlopen()` function family. However, these functions expect the code to be stored in a file and therefore require the user to pass a filename. There is no way to provide a preallocated buffer directly. Saving the user's code to a file on the provider's disk is unacceptable as well, as it has to be decrypted first to be loaded by `dlopen()`.

To work around this limitation, we use Graphene's virtual filesystem. A new virtual filesystem has been written to map filenames to previously allocated memory regions. A valid filename in this system consists of a memory address, encoded as a decimal number and the length of the memory region in bytes as a decimal number. Using this scheme, the user's code is placed encrypted in the host memory and is decrypted to the enclave's memory. Once it resides in enclave memory, a filename is generated to identify the memory region containing the code. This filename can then be given to `dlopen()`. Figure 4 shows how ciphertext and plaintext reside in host and enclave memory.

4 Performance Evaluation

To evaluate the performance of our secure remote computation framework, it first must be considered which aspects contribute to potential performance losses. The application deployed by the user is inherently distributed, communicating over a network. However, this is not unique to our work when compared to other cloud scenarios and thus, latency and performance loss due to communication, including the marshalling of data, are excluded. Instead we evaluate the overhead of the encryption mechanisms used to protect the user's secrets as well as the sandboxing scheme, including the use of SGX.

CPU and Memory Bound Performance To asses performance of mostly CPU- and memory-bound applications, an example application for edge detection has been developed. The *edgedetect* application reads a PNG file on the local side and sends it to the remote side. The remote side keeps the received PNG in RAM and performs an edge detection algorithm on it. It then sends back the resulting image showing the edges of the original. The resulting image is either displayed or saved on the local side.

Measurements are taken on the local side by starting a timer just before the image is sent over the network. Timing differences includes the transmission, encryption and decryption as well as the algorithm itself. However, it excludes the initial loading of the image from disk, as this is not considered to be part of the remote computation. This measurement cycle was repeated 30 times for four modes, each mode enabling or disabling the sandbox and encryption. In each mode the arithmetic mean, the median and standard deviation was calculated and compared to each other. The ratio was calculated by dividing the median of the mode in question by the median of the fastest mode, the baseline.

Tab. 1: Results for the *edgedetect* testcase.

SGX	encryption	median	s.dev.	ratio to fastest
no	no	84 ms	2 ms	1.00
no	yes	86 ms	3 ms	1.02
yes	no	87 ms	2 ms	1.03
yes	yes	90 ms	4 ms	1.06

Table 1 shows the result of our measurements. Modes running outside the sandbox show shorter time frames. Running the testcase with encryption enabled slows it down by about 2%. Enabling SGX, the testcase takes even longer, but stays within the same order of magnitude with a maximum slowdown of about 6%. During these tests, both memory access and CPU instructions were measured. The encryption only applies to the messages being sent between client and enclave.

System Calls and Disk Usage Our second testcase makes heavy use of I/O operations by writing and reading to files. The *ioslide* testcase sends a configurable number of bytes of arbitrary data from the local to the remote side. The remote side receives the data and enters a loop that is repeated a configurable number of times. Inside the loop, data is simply incremented byte-wise, encrypted and written to a file on the hosts disk as well as to standard output. The file is finally read in again, decrypted and compared to the original data.

This setup gives us two variables to measure: The number of bytes, i.e. the chunk size, determines how many bytes are handled by a system call and the number of iterations that controls how many system calls are executed. In analogy to the *edgedetect* application, the performance of the remote side is measured. The timer is started before sending the byte sequence and stopped after receiving a completion acknowledgement. The time difference includes the transmission, encryption, I/O operations, and decryption of the exchanged data.

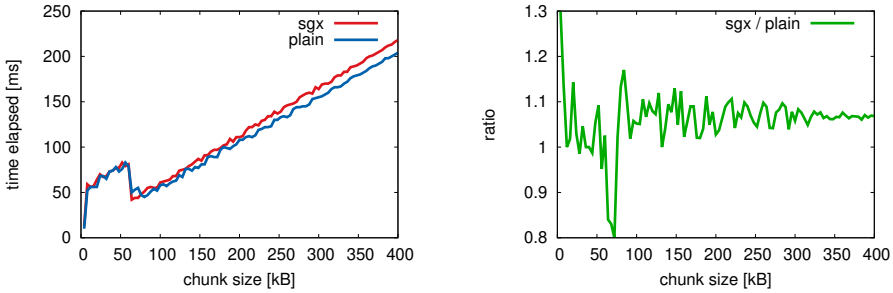

Fig. 5: Results of the *ioslide* testcase with fixed number of iterations but varying chunk size.

Figure 5 shows the result of varying the chunk size but keeping the number of read and write cycles fixed. The time was taken over 100 rounds, each round increasing the chunk size.

The chunk size starts at 4096 bytes and is multiplied by the round number, up to 409,600 bytes or about 400kB. The number of iterations, or cycles, is kept fixed at two. This leads to 8192 bytes being processed in the first round and 819,200 bytes in the last round.

Looking at the left diagram of Figure 5, it can be seen that both the SGX and the plain version remain in close proximity to a common line. Increasing the chunk size increases the time taken to process the data linearly in both versions. This is supported by the right diagram of Figure 5, which shows the ratio between the SGX and plain version. After initial oscillations, it settles on a constant ratio value of about 1.07.

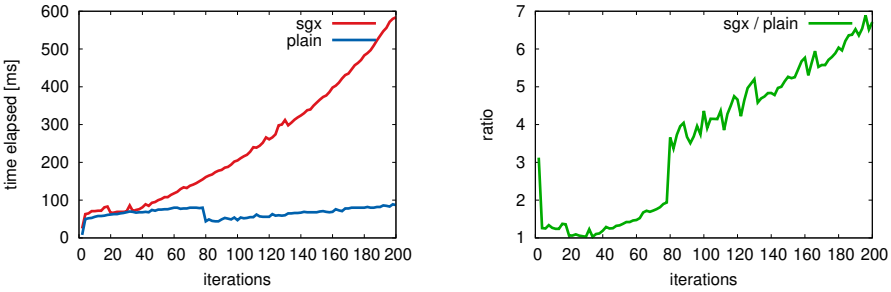

Fig. 6: Running *ioslide* with fixed chunk size and varying number of read/write cycles.

Figure 6 shows the result of keeping the chunk size fixed, but increasing the number of iterations in each round. The number of bytes in each chunk is kept at 4096. The number of cycles starts at two and is multiplied by the round number. Using this configuration, the total number of bytes processed in each round is the same as above, starting at 8192 bytes and leading up to 819,200 bytes.

As seen in the left diagram of Figure 6, the runtime measured of the SGX and plain versions diverge quickly. The time taken by the SGX enabled version increases at a steeper angle than the plain version. This becomes clearer in the right diagram of Figure 6, which shows the time taken by the SGX version divided by the time measured when running the plain version. The ratio settles on a line with a positive slope. It can therefore be concluded that the performance difference between the SGX and plain version increases with each round.

5 Conclusion

We introduced a framework providing remote computation using Intel SGX while considering practical pitfalls such as using Intel's Attestation Services to prove the integrity of the loading enclave to our users. Our framework consists of two parts, a local part running on the user's machine and a remote part which is executed within the provider's environment. Users can connect and authenticate themselves to the remote side, verify the integrity of a newly spawned loading enclave, and deploy confidential code at the provider's machine. The performance overhead is below 10% compared to remote computation without using SGX.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89).

References

[Ar16] Arnautov, Sergei; Trach, Bohdan; Gregor, Franz; Knauth, Thomas; Martin, Andre; Priebe, Christian; Lind, Joshua; Muthukumaran, Divya; O’Keeffe, Dan; Stillwell, Mark; Goltzsche, David; Eyers, David M.; Kapitza, Rüdiger; Pietzuch, Peter R.; Fetzer, Christof: SCONE: Secure Linux Containers with Intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI. 2016.

[Be17] Bernstein, Daniel J.: , NaCl: Networking and Cryptography library, July 2017.

[BPH14] Baumann, Andrew; Peinado, Marcus; Hunt, Galen C.: Shielding Applications from an Untrusted Cloud with Haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI. 2014.

[G7] Götzfried, Johannes; Eckert, Moritz; Schinzel, Sebastian; Müller, Tilo: Cache Attacks on Intel SGX. In: Proceedings of the 10th European Workshop on Systems Security. EuroSec’17, 2017.

[Ho13] Hoekstra, Matthew; Lal, Reshma; Pappachan, Pradeep; Phegade, Vinay; del Cuvillo, Juan: Using innovative instructions to create trustworthy software solutions. In: Workshop on Hardware and Architectural Support for Security and Privacy. 2013.

[ri16] rightscale: , Cloud Computing Trends: 2016 State of the Cloud Survey, February 2016.

[Sc15] Schuster, Felix; Costa, Manuel; Fournet, Cédric; Gkantsidis, Christos; Peinado, Marcus; Mainar-Ruiz, Gloria; Russinovich, Mark: VC3: Trustworthy Data Analytics in the Cloud Using SGX. In: IEEE Symposium on Security and Privacy. 2015.

[TPV17] Tsai, Chia-che; Porter, Donald E.; Vij, Mona: Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: USENIX Annual Technical Conference. 2017.

[Zh17] Zheng, Wenting; Dave, Ankur; Beekman, Jethro G.; Popa, Raluca Ada; Gonzalez, Joseph E.; Stoica, Ion: Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI. 2017.