Secure Database Outsourcing to the Cloud: Side-Channels, Counter-Measures and Trusted Execution

Matthias Gabel, Jeremias Mechler

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Emails: {matthias.gabel,jeremias.mechler}@kit.edu

***Abstract*—Outsourcing data processing and storage to the cloud is a persistent trend in the last years. Cloud computing offers many advantages like flexibility in resource allocation, cost reduction and high availability. However, when sensitive information is handed to a third party, security questions are raised since the cloud provider and his employees are not fully trusted. Standard security mechanisms like transport encryption and regular audits alone cannot solve the issue of insider attacks. Additional cryptographic techniques are required. In this paper, we build upon an existing proxy for secure database outsourcing. We address potential side-channels and weaknesses, which are later analyzed and mitigated. Furthermore, we take a look at trusted execution environments (TEEs) like Intel Software Guard Extensions (SGX) and show how they can be applied to allow for secure execution in the secure database outsourcing case.**

***Keywords*-Databases, Secure Outsourcing, Side-Channels, Trusted Execution, SGX**

I. Introduction

The usage of cloud services for processing and storing large amounts of data has reshaped the IT industry in the last years. Adaptive use of resources, cost-efficiency and high scalability are among the major benefits of cloud computing. Especially for small and medium enterprises, this has high potential since such companies usually are not in a position to build and maintain their own computing centers.

Despite the aforementioned advantages of cloud computing, it must be clear that security concerns are a huge drawback. Because some employees – mainly system administrators – have high privileges, there is a high risk of a so-called insider attack. As a consequence, in particular personal data must be protected from curious cloud providers. Depending on the scenario, it is even impossible to outsource sensitive data to the cloud at all while still complying with the law. Fortunately, these concerns can be addressed using cryptographic methods.

Therefore, there is a need for an entity that secures the outsourced data and handles the cryptographic that methods transparently to the application. This work addresses an existing database outsourcing scheme and points out security issues as well as mitigation techniques. The entity considered, called the database proxy (DB proxy), provides a practical, reasonable trade-off between performance, SQL expressiveness and security.

Furthermore, we consider possible applications of trusted execution environments (TEEs) at the example of Intel's implementation SGX. TEEs allow an untrusted remote machine to execute instructions on sensitive data in a secure way so that data privacy is guaranteed and the code is not altered. Due to the huge amounts of data that have to be handled and implementation limitations, it is currently not possible to move the whole DBMS (database management system) into a trusted execution environment.

Our contribution is two-fold: We analyze side-channel attacks and show how they can be countered with both conceptual adaptations and TEEs.

This work is structured as follows: Section I-A gives an overview of existing database outsourcing schemes and trusted execution environments. The DB proxy where this work mainly builds upon is described in detail in Section II. Side-channel attacks and countermeasures against the DB proxy are discussed in Section III. Sections IV, V and VI introduce TEEs and SGX technology and show they can be applied to increase the DB proxy's security. Finally, Section VII concludes and gives an outlook on future research directions.

A. Related Work

With respect to secure database outsourcing, there are currently three Horizon 2020 programs called Secure Enclaves for Reactive Cloud Applications (SERECA)[1], SecureCloud[2] and PaaSword[3] (see Section II). While SERECA intends to "build a platform able to protect the confidentiality and integrity of applications and services executed in the cloud", especially considering reactive applications, SecureCloud "aims to remove technical impediments to dependable cloud computing, i.e., SecureCloud will ensure the confidentiality, integrity, availability and security of applications and their data" with the focus on secure and efficient data processing.

Both projects make use of trusted hardware, namely SGX. Of particular interest is SGX-MySQL [1], which is a partitioned version of MySQL that executes parts of the

[1]https://www.serecaproject.eu
[2]https://www.securecloudproject.eu/
[3]https://www.paasword.eu

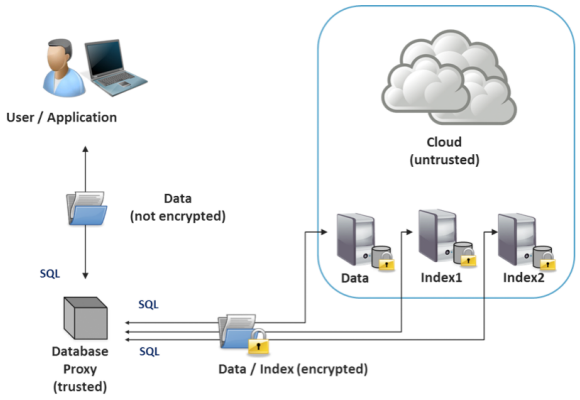

Figure 1. Deployment of the DB proxy between user/app and cloud.

DBMS in a secure enclave, providing data confidentiality and result integrity while minimizing the trusted computing base (TCB), i.e. the amount of software inside the enclave.

Secure Linux Containers with Intel SGX (SCONE) [2] is a mechanism to deploy Docker containers within SGX enclaves by providing a small-footprint C standard library supporting transparent encryption and decryption of input and output, keeping the overhead and the TCB small.

VC3 [3] is a framework for securely running MapReduce algorithms with SGX. By only running the core algorithms inside the enclave and other parts like the operation system and the Hadoop framework outside, the performance overhead is minimized.

II. Architecture and DB Proxy Overview

As illustrated in Figure 1, we consider the case of database outsourcing with a trusted entity (the database proxy) that is located between the user and the untrusted cloud provider and transparently handles encryption and decryption. With respect to data encryption, there are basically two alternatives: While fully homomorphic encryption (FHE) allows performing arbitrary queries on the encrypted data in the cloud provider's realm and decrypting the result locally, its overhead in the order of 10^7 [4] makes it unsuitable for practical use. Another approach could be to download all data, decrypt it, perform the operations and encrypt it before finally uploading it again. However, this is also not applicable because it requires all data to be transferred for every query, reducing the concept of outsourcing *ad absurdum*.

The chosen approach for the DB proxy is somewhat similar to the last one, with the goal to transfer as little data as possible. Therefore, we use additional data structures to determine which data segments seem relevant to the query, relying on the relational model of structured data. A simple version of the DB proxy presented here was created within the research project MimoSecco [5], which had the goal to enable the secure use of cloud services. In the PaaSword project, the DB proxy [6] was redesigned and adapted to be able to handle more complex, realistic SQL queries while providing higher performance and better security.

Row	Name	Surname	Condition
1	Claire	Jones	Diabetes
2	Emilia	Jones	Hypertonia
3	Phil	Connor	Hypertonia

Table I
Original data before the transformation.

Row	Encrypted Data
1	Enc(Claire‖Jones‖Diabetes)
2	Enc(Emilia‖Jones‖Hypertonia)
3	Enc(Phil‖Connor‖Hypertonia)

Table II
Encrypted data table after the transformation.

Consider a simple SQL select query, e.g. *SELECT * FROM customers WHERE customer_id = 12345*. In a nutshell, the DB proxy executes the following steps for this query:

1) **Query analysis** Determine which parts of the tables are affected and how the query is to be processed.
2) **Retrieval of relevant data** Download and decryption of only the relevant data fraction from the cloud.
3) **Query execution** Data is temporarily inserted into an empty local database and queried there to return the result to the user application.

If a query is highly selective like in the example above (... *customer_id = 12345*) this is a huge improvement compared to the naïve approach where all data is downloaded. In order to find out which parts of the cloud database are needed to process the query we use index tables that rely on a reverse index for the attributes.

The aim is that the application is left untouched and always handles the database as if it were not encrypted or distributed. All encryption, decryption, distribution and composition is done by the DB proxy. This makes the security measures transparent to the application.

To understand how data is manipulated before upload we give an example of the PaaSword transformation [6]. Consider a simple table as depicted in Table II that needs to be protected because it contains personal identifiable information. This table represents the application's view on the data and obviously cannot be outsourced that way. We outsource two different classes of tables instead: encrypted data tables, which hold all information in encrypted form and index tables for faster look-up of relevant data.

The encrypted data table as depicted in Table II is identical to the original table but uses probabilistic row-based encryption. Only the ID column is in plain text. The attributes in each row are concatenated (indicated by the symbol ‖) before encryption.

The index tables as depicted in Table II represent a reverse index for each attribute. The first column contains each attribute once while the second column is a probabilistically encrypted list of all the rows where the specific attribute is found for a given column. Each entry in this list is the ID

Keyword	Rowlist
Phil	Enc(3)
Claire	Enc(1)
Emilia	Enc(2)

Keyword	Rowlist
Jones	Enc(1‖2)
Connor	Enc(3)

Table III
INDEX TABLES AFTER THE TRANSFORMATION.

where the corresponding row is found in the encrypted data table.

The described encryption and distribution mechanisms make the outsourced data reveal less information to the honest-but-curious cloud provider.

III. SIDE-CHANNEL ATTACKS AND COUNTER-MEASURES

The security notion that is achieved by this technique is called Ind-ICP (indistinguishability under independent column permutation) [7]. It guarantees that that only single attributes are leaked but not the correlation among them. There is, however, data that is sensitive even without association to other attributes. Credit card numbers are such an example and need to be protected by other means as described in the next section. The intuition is that after the transformation is applied, the cloud provider cannot distinguish between the original and a permuted database. The permutation is applied to each column independently, thus destroying the association between attributes in every row of the original table. For more information and a security proof that the proposed transformation achieves this security notion, see [7]. Ind-ICP, however, is not sufficient to provide real-world security because of attacks outside the security model. Such issues emerge due to real-world deployments and are considered as side-channels that can be exploited to gain additional information. Side-channel attacks against the DB proxy haven been addressed before [8]. The fact that our proposed scheme and implementation are vulnerable to those kinds of attacks is not a specific issue but one that applies to secure outsourcing schemes in general.

A. Side-Channel Attacks

The following observations can be used by the cloud provider to infer non-trivial information about the dataset.

1) Background Knowledge: An adversary with background knowledge can attack any scheme that stores some meta data in the plain. Consider a database that only contains people that suffer from a rare disease and live in a small town. If the adversary observes that someone with a rare first or last name occurs in the dataset it can be inferred that this person probably has this disease. As mentioned before this attack is not an inherent problem of our scheme. Our scheme is vulnerable to this attack but it can be prevented by counter-measures like deterministic index encryption or distributing the index tables to different (non-communicating) servers. Notice that the security notion Ind-ICP requires that the adversary has no background knowledge.

Keyword	Rowlist
Phil	Enc(3)
Claire	Enc(1)
Emilia	Enc(2)

Keyword	Rowlist
Jones	Enc(1‖2)
Connor	Enc(3)

Table IV
INDEX TABLES BEFORE OPERATION.

Keyword	Rowlist
Claire	Enc(1)
Emilia	Enc(2)

Keyword	Rowlist
Jones	Enc(1‖2)

Table V
INDEX TABLES AFTER OPERATION.

2) Observing Database Operations: If metadata (like index entries in our case) is not encrypted, an adversary can gain additional information by looking at differences in the tables.

Consider the following set of index tables depicted in Table IV and those in Table V. Assume the adversary observes the state represented in Table IV and notices that a DELETE query was performed. After comparing with the new state represented in Table V it is clear that the attributes *Phil* and *Connor* are associated. Notice that this is also possible if a new row is inserted that has unique attributes. Thus, INSERT and UPDATE queries can leak information as well. To mitigate this vulnerability our recommendation is – as in the above mentioned case – index encryption or distribution.

3) Access Statistics and Order on Physical Storage Devices: Sometimes a database is not constantly queried but query groups can be separated by looking at the time. Consider once again a state as depicted in Table IV and two queries that are issues with only little delay:

- SELECT * FROM index1 WHERE keyword = 'Phil'
- SELECT * FROM index2 WHERE keyword = 'Connor'

The adversary can at least assume that there could be a correlation. He can be certain if he can register access to the encrypted data table and an (encrypted) result is returned meaning that the query has a non-empty response. Of course this attack only works if the query is highly selective.

Similarly an adversary can correlate index table rows even if not two states but a single one is observed. It is safe to assume that real-word database systems store new records at the end of the existing data structures. This does not necessarily hold for tree-based data types but often at some level unique ascending IDs are inserted. This is the case for our implementation that uses PostgreSQL.

Because these attacks are similar to the one mentioned before, the same mitigation techniques apply as well.

B. Countermeasures

This section will introduce techniques that can be used to mitigate the aforementioned side-channel attacks.

1) Index Encryption: Notice that an index table has two columns. The left column holds all the keywords that can be looked up. The second column is a probabilistically encrypted list of rows. Since there is little to gain from observing a

ciphertext except the plaintext's length, the main issue is the first column. As pointed out in the previous section, there is a need to enable for index encryption which means to apply deterministic encryption to the first column. This is done separately for every cell.

Instead of searching for a specific keyword *SELECT * FROM index1 WHERE keyword = 'Connor'* we compute the ciphertext Enc('Connor') first. Then we query *SELECT * FROM index2 WHERE keyword = Enc('Connor')*. Because of the deterministic encryption, all exact-match statements can be easily transformed that way.

There are, however, some issues with this approach. First, wildcard searches such as SQL's *LIKE* query are not supported. Second, performance will be heavily degraded if a binary search on the keywords is required.

2) Data Distribution Using Privacy Constraints: Some of the aforementioned side-channel attacks rely on associating rows from different index tables. This is not possible if the index tables are located on different non-communicating servers. This could be achieved by using different cloud providers. Further details can be found in [9].

3) Shuffling Modifications: The goal of this technique is to limit the adversary's ability to associate database rows, especially among index entries. For this purpose write operations are put into a queue. If the dependencies on other queries allow, the queue items are shuffled and thus executed in a random order. Furthermore, it is possible to execute them with random delays or all at once if the queue size reaches a desired number.

IV. Practical Limitations of the Current Approach

As outlined above, there are many things to consider to securely deploy the DB proxy and the index servers. In particular, the DB proxy is entrusted with the database keys and sees the decrypted query results, precluding the proxy from being deployed in a public could or being part of a PaaS package. As a consequence, this may lead to higher costs due to the local deployment and higher access times due to the distance between proxy, index servers and the encrypted database.

As data (such as phone numbers) might be considered sensitive on its own even when not linked to other information, it is highly desirable to encrypt indexes containing such sensitive data. While deterministic encryption allows for efficient exact-match queries even on encrypted indexes, it is often necessary to decrypt the whole index for other query types such as comparisons, leading to performance degradation.

Under the assumption that different cloud providers do not co-operate, distributing index sets with non-sensitive data over multiple providers provides security against linking attacks. However, it also incurs possibly higher costs and administrative overhead because providers must be chosen in a way that the privacy constraints are enforced.

In the following, we consider how these limitations of the current PaaSword DB proxy can be alleviated by using TEEs such as SGX.

V. Trusted Execution Environments and SGX

After giving a short introduction to trusted execution environments and Intel's implementation SGX, roughly following [10] and [11], we show how these tools can be used to efficiently solve the aforementioned problems.

A. Trusted Execution Environments

In short, a trusted execution environment implemented in hardware allows the execution of software *isolated* from the remaining system like the operating system, BIOS, EFI or management engine (ME). Using keys bound to the hardware which are usually certified by a public-key infrastructure, the trusted execution environment can *attest* the running software by creating signatures of the loaded program before, during and after execution, allowing a third party to verify the execution's integrity, e.g. before deploying sensitive encryption keys.

B. SGX: An Implementation

Starting with the Skylake microarchitecture, many Intel processors support an instruction set extension called Software Guard Extensions (SGX), which aim to provide a trusted execution environment on the processor. Conceptually, the trusted execution environment is implemented by a dedicated memory region called processor reserved memory (PRM), which is protected from outside access. Part of the PRM is the so-called enclave page cache (EPC) which holds the enclave-internal memory. Currently, the EPC size is limited to 128 MB, which can be extended by paging (which is performed by the host operating system) at the cost of performance loss as pages evicted from the EPC have to be encrypted and authenticated by the memory controller before they are stored inside the normal system memory. While SGX protects from outside memory access, it does not protect from timing attacks with respect to the processor caches (which are shared between enclave and non-enclave memory) and memory access. Using services provided by Intel, remote attestation and subsequent deployment of sensitive information to the enclave is possible. We consider SGX in this paper due to its general availability, its features (in contrast to e.g. ARM TrustZone which does not feature attestation or memory encryption) and its popularity in cryptography-related research. However, our results are generally applicable to other TEEs with comparable features as well.

VI. Using Trusted Execution Environments for Secure Database Outsourcing

At first glance, the reasonable solution to database outsourcing would be to run the whole DBMS inside a secure

enclave. However, current implementations such as Intel SGX suffer from practical problems such as limited enclave page cache [2], which severely impact performance in realistic deployment scenarios. Thus, we focus on the problems mentioned in Section IV in order to improve the PaaSword secure database outsourcing scheme by running only parts inside attested trusted execution environments. In particular, we are interested in the following scenarios:

1) Running the DB proxy inside an enclave
2) Storing the index in the clear inside the enclave in contrast to encrypting each row individually

A. Adversarial Model

In order to discuss possible advantages and drawbacks when using trusted execution environment, the adversarial model has to be properly defined. In the following, we consider three disjoint classes of adversaries:

1) The party controlling the root of trust, usually the hardware manufacturer who controls the public key infrastructure, master keys and possibly hardware-specific keys. A compromised root of trust can result in false attestations, convincing other parties that the execution environments runs some program which it actually does not. Compromised keys can, for example, break the confidentiality of stored or communicated data or enable a malicious enclave to fake a different identity. Furthermore, if the implementation of the trusted execution environment were broken, data could be leaked to outside programs. While trusting the manufacturer seems like a very strong assumption to make, we implicitly trust the manufacturer when running software outside of trusted enclaves, too (for example to not involuntarily send sensitive data over the network).
2) Apart from the manufacturer, the cloud provider (and other tenants that run software on the same hardware), could be considered adversarial. We assume that the cloud provider is honest but curious, meaning that he adheres to the protocol but uses available information he gets by observing the communication and machine state. With respect to other tenants, timing cache and memory access could leak information about memory access of programs running in an enclave as the processor cache is shared. If programs running in the enclave are not data-oblivious, memory access patterns could possibly leak information about sensitive data. While such attacks have been successfully demonstrated for SGX [12], we consider them out of scope for this paper, as they are not inherent to TEEs in general.
3) In addition, we have to consider (possibly active) adversaries that are separate from the aforementioned two, which, for example, try to tamper with the communication. Assuming a correct implementation of the trusted execution environment and a small trusted computing base, the isolation property of the trusted execution environment can protect the enclave from untrusted and possibly malicious code running on the host system. In case of SGX, the enclave is also protected from BIOS, EFI and ME. Using counters, nonces and CCA-secure encryption, attacks on the communication can be prevented.

B. Running the DB proxy Inside the Enclave

In the original model as outlined above, the DB proxy locally stores sensitive data and encryption keys in the clear, which requires the proxy to be set up on a trusted system, precluding it from being outsourced.

However, when using a trusted execution environment, trusting the underlying system is no longer necessary. The consequences are two-fold: A DB proxy instance running inside an enclave is isolated from the host system, which greatly enhances security as certain classes of attacks (such as cold boot attacks to extract encryption keys or vulnerabilities at the operating system level) are mitigated. Furthermore, the proxy application can be set up at a possibly untrusted third party such as a public cloud provider, which can cryptographically attest to the tenant that the enclave runs the desired program before he deploys his encryption keys.

In contrast to running the whole DBMS inside an enclave, which would be impractical in SGX due to the small enclave page cache and the associated performance overhead of random memory access which is the typical access pattern in databases, the DB proxy is well-suited for the execution inside an enclave: As only the results of the current query are needed, memory consumption is a function of the result size and not the whole database.

Being able to run the DB proxy in proximity of the index and data servers has the potential to greatly benefit performance due to the reduced communication latency. Furthermore, a PaaS provider is then able to run multiple DB proxy instances on the same physical server while still guaranteeing tenant isolation.

The feasibility of running a DBMS (SQLite [2] and a partitioned version of MySQL [1]) inside SGX has been successfully demonstrated.

Even when not trusting the manufacturer (and thus the attestability, integrity and confidentiality of the trusted execution environment), running the DB proxy at least partially within an enclave can still be beneficial when considering adversaries which are distinct from the party controlling the trust anchor as explained above.

C. Storing Index Tables Unencryptedly Inside the Enclave

In order to protect sensitive data, our DB proxy architecture allows to employ encrypted indexes, which is a sensible measure when the index data is sensitive as-is (cf. Section III). In the current implementation, the keywords are encrypted as single ciphertexts, leading to both a high performance and

space overhead due to the necessary padding and encoding. While using *deterministic* encryption schemes allows for efficient execution of exact-match queries, other query types such as comparisons cannot be efficiently performed and require the decryption of the whole index table at the proxy, incurring computation and communication overhead.

Fortunately, modern DBMS allow encryption to be performed at their persistency layer. When running the DBMS inside an enclave, we can make use of this feature by storing the data outside the enclave encrypted with a key that is never exposed to the untrusted host system and only deployed after the enclave content has been attested. While we still potentially suffer from the limited enclave page cache and the resulting penalty for random memory access, the index-related queries can be performed very efficiently due to the use of b-tree data structures used in database systems: To access an element in a sorted set of n elements, only $O(\log(n))$ memory queries are necessary, which allows us to establish an upper limit of the possible page faults. For many query types, the correct database row(s) can be found using the memory-efficient index data structures of the DBMS. Furthermore, the query can always be fully evaluated on the index server instead of having to transfer the whole index table in the worst case.

Depending on the nature of the data, this setting might even be interesting if the hardware manufacturer is not completely trusted. For example, phone numbers might be interesting to criminals but not to Intel. In contrast, genomic data might be considered so sensitive that it should not even be possibly exposed to the manufacturer or any co-operating party.

VII. Conclusion and Outlook

We have shown how to address major limitations, namely the necessity to host the DB proxy at a trusted location and the performance degradation for some query types associated with encrypted indexes. In principle, they can be alleviated using attestable trusted execution environments such as Intel SGX. In order to show the practical feasibility, our ideas would have to implemented, possibly re-using existing approaches such as SCONE [2] or SGX-MySQL [1] together with the existing DB proxy implementation.

While we expect that implementation limitations such as the limited enclave page cache size with Intel SGX will be removed in future processor generations or competing solutions, questions regarding the trustworthiness of the trust anchor (usually the manufacturer like Intel) will prevail, making it necessary to keep using distributed approached like the one presented in PaaSword for highly sensitive data due to privacy concerns. Thus, our findings are still of relevance even when running whole DBMS without size limitations inside trusted execution environments becomes feasible.

Acknowledgment

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644814, the PaaSword project (www.paasword.eu) within ICT-07-2014: Advanced Cloud Infrastructures and Services.

References

[1] C. Fetzer, D. Goltzsche, S. Brenner, T. Knauth, S. Arnautov, K. F. W. Barcynski, P. Pietzuch, N. Weichbrod, M. Verbur, and C. Escoffier, "Design of system support for secure enclaves (initial) d1.3," 2016.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with Intel SGX," in *USENIX*, 2016.

[3] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: trustworthy data analytics in the cloud using SGX," in *IEEE Symposium on Security and Privacy*, 2015.

[4] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic Evaluation of the AES Circuit," Cryptology ePrint Archive, Report 2012/099.

[5] D. Achenbach, M. Gabel, and M. Huber, "Mimosecco: A middleware for secure cloud storage," in *ISPE International Conference on Concurrent Engineering*, 2011.

[6] R. Dowsley, M. Gabel, K. Yurchenko, and V. Zipf, "A database adapter for secure outsourcing," in *IEEE International Conference on Cloud Computing Technology and Science*, 2016.

[7] M. Huber, M. Gabel, M. Schulze, and A. Bieber, "Cumulus4j: A provably secure database abstraction layer," in *ARES Conference*. Springer, 2013.

[8] M. Huber and G. Hartung, "Side Channels in Secure Database Outsourcing on the Example of the MimoSecco Scheme," in *Trusted Cloud Computing*. Springer, 2014.

[9] M. Abdelraheem, T. Andersson, and C. Gehrmann, "Inference and record-injection attacks on searchable encrypted relational databases," Cryptology ePrint Archive, Report 2017/024.

[10] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016.

[11] R. Pass, E. Shi, and F. Tramer, "Formal abstractions for attested execution secure processors," Cryptology ePrint Archive, Report 2016/1027, 2016.

[12] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," *arXiv preprint arXiv:1702.08719*, 2017.