Scaling Databases through Trusted Hardware Proxies

Kai Mast
Cornell University

Lequn Chen
Shanghai Jiao Tong University

Emin Gün Sirer
Cornell University

Abstract

Trusted execution environments (TEEs) allow asserting the integrity of previously untrusted third parties using novel hardware features. Unlike previous approaches to trusted computing, they have become readily available on most consumer devices sold today. This opens up the possibility for many novel applications, where not only the server, but also clients are equipped with trusted hardware.

This work presents a mechanism to run trusted proxies on clients in order to offload large parts of the workload from a database server. We show that none of the integrity and confidentiality guarantees provided by the database are weakened as a result this mechanism. Evaluation shows that we can improve throughput by at least an order of magnitude, when the database server itself runs in a TEE. Further we can improve performance by a factor of two, even in the case where the server is not limited by a TEE.

ACM Reference format:
Kai Mast, Lequn Chen, and Emin Gün Sirer. 2017. Scaling Databases through Trusted Hardware Proxies. In *Proceedings of SysTEX'17:2nd Workshop on System Software for Trusted Execution , Shanghai, China, October 28, 2017 (SysTEX'17),* 6 pages.
https://doi.org/10.1145/3152701.3152712

1 Introduction

Trusted execution environments (TEEs) are a newly emerging hardware feature that can enhance the functionality of demanding server applications such as databases. They can be used, for example, to ensure database integrity [7] as well as confidentiality of the stored data [1, 4].

SysTEX'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5097-6/17/10…$15.00
https://doi.org/10.1145/3152701.3152712

However, because of the unique programming model, it is non-trivial to port databases into a TEE environment. In particular, database applications typically require a large working set, whereas the current industry standard for TEEs, Intel SGX, only provides a limited amount of memory for enclaves. Not surprisingly, measurements of database performance as a function of working set size, as shown in Figure 1 show that working set sizes that exceed enclave memory limits lead to drastic reduction in performance. Thus, new methods need to be investigated on how to scale databases efficiently in a trusted environment.

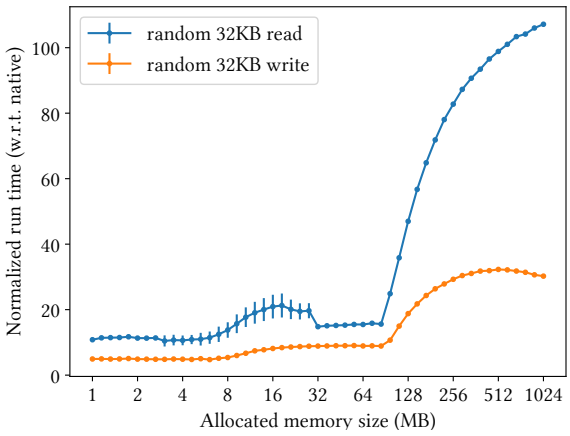

Figure 1. The overhead of memory access inside SGX. Once the memory size exceed that of the EPC overhead increases significantly.

This paper proposes a technique for offloading database functionality to clients, securely, through the use of TEEs. This is a general approach that can be used to harness the additional resources provided by clients by offloading work from centralized servers to the periphery of the network. It enables the centralized database to take advantage of greater concurrency, CPU, memory and other resources provided by the clients. Specifically in the SGX context, it enables computations that do not fit in a single enclave memory to be distributed across multiple enclaves.

The core mechanism that serves as the foundation of our approach is a proxy that implements the same service API as the server. When the proxy operates in a TEE, it can use remote attestation to establish that it is capable of maintaining the same semantics for data as the centralized server. This enables the server to outsource functionality, such as maintaining constraints, performing access controls, and the like,

to the proxy, instead of performing them on the centralized server.

In the next section, we provide a RAID-like taxonomy, useful for capturing the outsourcing decisions related to TEE-backed applications. We then describe the design of our client-side proxy mechanism. Finally, we provide a preliminary evaluation of our prototype. Our evaluation shows that this approach can achieve an improvement of an order of magnitude in throughput, without impacting latencies. Further, the design does not compromise the integrity and confidentiality guarantees provided by the TEE.

2 Networks of Trusted Enclaves

Applications that are comprised of multiple connected processes can leverage TEEs in many different ways, and at various different locations of their network. In this section, we break the different setups down into 5 classes of *Networks of Trusted Enclaves (NoTEs)*. Further, we categorize related work in one of these categories.

NoTEs rely on an essential feature of TEEs: remote attestation. Remote attestation creates a bidirectional encrypted channel, that comes with a one-way integrity guarantee. At a very high level, one party sends a signed quote about its enclave, enabling the other party to verify the quote and to determine whether their counterparty is running on a trustworthy TEE. Finally they establish a secure session using Diffie-Helman Key Exchange (DHKE). For the remainder of the established session, they then have a trusted and encrypted communication channel between the two endpoints.

Two remote attestations can be chained to create a bidirectional channel that asserts integrity in both directions of the channel. Setups with more than one TEE in a network rely on such two-way channels to establish secure communication between enclaves. Application secrets can then be shared through such channels.

Applications are comprised of both a trusted, isolated component that executes inside a TEE, as well as a untrusted component responsible for interaction with the external world. The untrusted component is responsible for input/output operations, including all operating system services.

2.1 NoTE 0 and 1

The baseline architecture for a high-integrity database is one where there is a single TEE that hosts a trusted execution environment on the server-side to provide execution integrity guarantees to clients. Clients, however, do not provide trusted environments and are assumed to be non-malicious. The SCONE [3] and Haven [6] projects focused on such a setup.

NoTE 1 systems bring additional resources by sharding the single server across multiple hosts, each supporting a TEE. This enables the same integrity guarantees, while providing higher scalability. NoTE 1 systems can also describe a setup where confidential data is processed by being passed between multiple trusted servers. Such a setup is described in Ryoan [10].

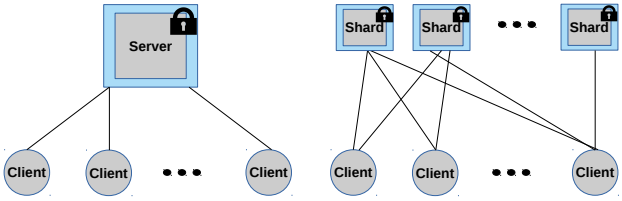

Figure 2. In a NoTE 0 setup (left) the server provides a TEE. NoTE 1 (right) extends this design by sharding the server across multiple machines, each providing a TEE.

2.2 NoTE 2

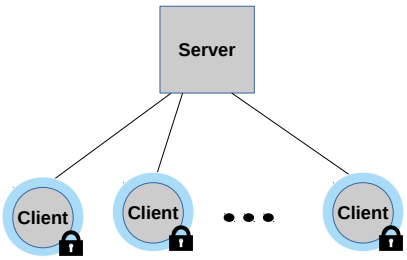

Figure 3. In a NoTE 2 setup, only the clients provide TEEs

In many environments the server wants to shield the application from byzantine clients. For example, it might want to shield application logic from clients [9], or prevent participants in an online video game from cheating [5]. Similarly, application constraints may prohibit storing the unencrypted dataset outside of the enclave. One might want to cache data at the client to increase performance. However, as soon as data is moved to the client, the application logic loses all control over it. This is not desirable as, for example, applications might want to enforce policies [14] on the data. Further, an unprotected client cache would increase the attack surface of the system considerably. Servers that have no control over the software stack of a client and cannot ensure they run a secure system that is guarded against attacks.

In such cases, the clients each can provide a TEE to assert to the server that they will not misbehave or leak private information. We classify such systems, where each client provides a TEE, as NoTE 2.

2.3 NoTE 3 and 4

NoTE 1 and NoTE 2 systems assume a hybrid trust environment: either the server or the clients are assumed to behave

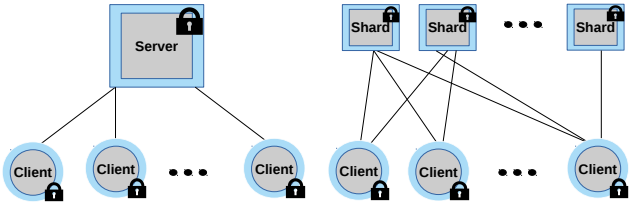

Figure 4. In a NoTE 3 setup (left), all members of a client-server setup are backed by a TEE. NoTE 4 (right) extends this by sharding the server across multiple machines where each is backed by a TEE

non-maliciously. However, in a full byzantine environment, this assumption does not hold. Clients may not trust the server, but the server may also not trust the client. NoTE 3 systems can handle full byzantine environments by requiring both the server and the clients to run in trusted environments.

Similarly NoTE 4 systems consist of a set of clients that each run a TEE and a set of server shards that each provide a TEE. This setup allows both to offload computation to clients and to scale the server horizontally.

2.4 NoTE 5

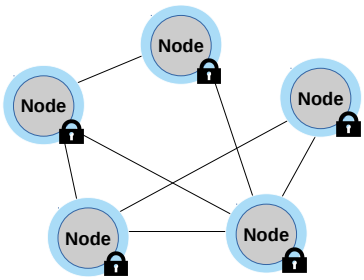

Figure 5. A NoTE 5 setup is a full peer to peer network, where each peer runs a TEE.

At NoTE levels 1 through 4, all of the servers were under the physical control of a single entity. A final, most aggressive form of outsourcing is a database configuration where servers, equipped with TEEs, are provided by multiple mutually distrusting entities, and form a peer-to-peer database cloud. NoTE 5 systems have the potential to untap large amounts of resources, though they also pose significant challenges, as the compromise of a single host can affect the integrity of the system. Examples of early NoTE 5 systems are payment networks [11] and permissionless consensus mechanisms that use TEEs as a Proof-of-Work replacement [15].

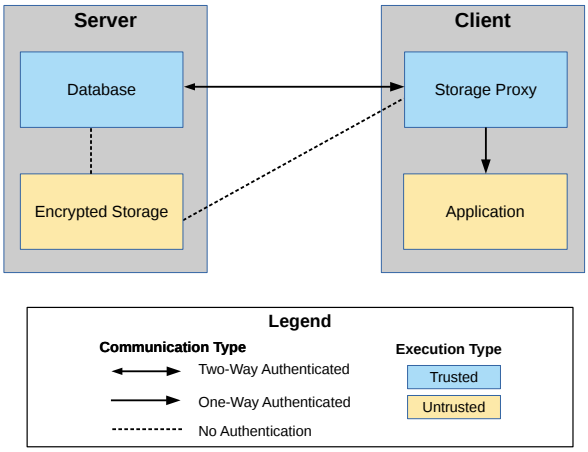

Figure 6. Our design is a NoTE 3 setup, where each client interacts with the server through a proxy it runs locally. Proxies can read data either directly from the storage layer or interact with the database.

3 Design of Client-Side Proxies

This paper focuses primarily on NoTE 2 and 3 environments. In particular, we sketch how a database can securely offload work to its clients through the use of TEEs. The database is a simple key-value store that can run either fully run inside SGX (*enclave mode*) or in a regular environment (*fake enclave mode*). To simplify this discussion, we assume a datastore that support only read and write operations. The database further enforces access policies on the data.

The proxy mechanism relies on clients running a TEE-based proxy service. The key insight is to offload processing from the server side to the clients, where the exact same functionality that would be executed inside the server is instead performed on the client, using client resources. This approach is particularly effective for any kind of processing that pertains specifically to the client's request, and is ineffective when the processing to be performed requires accessing shared data. In particular, operations such as checking parameters, maintaining invariants, and enforcing security policies that pertain solely to the values provided by the client are operations that can benefit from a NoTE 2 or 3 architecture. A security policy that checks, for instance, that fields x, y in a client transaction have a property $x \geq y$ is one that is going to yield large performance improvements in these architectures, as these checks can be performed concurrently, without coordination. A security policy that checks a field x against another object in the database, not part of the client's transaction, is still guaranteed to operate correctly, by forwarding the processing back to the server, but will not yield a performance benefit.

The mechanism is effective for workloads that are read-heavy. Sine writes have to be processed by the centralized

server, secure clientside processing is unlikely to yield significant benefits on update-heavy workloads. This means, while many operations can be served by the proxies, not all can. For example, a service that wants to limit the number of accesses to an object has to do so by tracking its state at the server. Thus, in such a setup, proxies cannot permit reads without contacting the server. Proxies can, however, serve other policies set by the datastore, such as limiting access to a specific set of users. NoTE 2 and 3 systems provide failure isolation guarantees. Proper implementation of the proxy can ensure that a proxy failure shall only affect clients connected to it. In order to achieve this, proxies need to forward all updates to the server, and only handle reads directly.

Any time data is replicated, for instance, between the server and client proxies, there is the potential for nodes to operate on stale data. Two design decisions ensure that the server always operates on fresh data. First, the database stores entries in an append-only log, partitioned into blocks. This enables a straightforward paging mechanism, where data that cannot fit into the server's enclave can be safely sealed and offloaded out of the enclave. Efficient paging is enabled by storing all changes to the data in an append-only log, which is sequenced into blocks. Only the most recent block can be modified, and is kept in memory at all time. Previous work has shown that such a custom paging mechanism can yield a performance gain over the built-in SGX mechanism [13]. This mechanism does not prevent from loading stale data after an enclave has been restarted, which is out of scope for this paper.

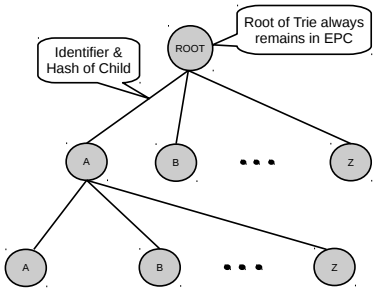

Figure 7. The index is stored as "Merkle-trie" ensuring small index size and high integrity.

In order to access the append-only log efficiently, an authenticated index stores the location of the most recent value of an object. A memory-efficient way to implement such an index is using a trie [8]. The index might only fit in memory partially, in which case the database will start to evict parts of the index as well. To ensure trie-nodes loaded from disk are the most recent, each nodes hold hashes for its children forming a Merkle tree [12]. Figure 7 shows a sketched layout of the Merkle-trie. An update then has not only to update the leaf representing the object, but also update Merkle-hashes.

Figure 6 shows how clients, server, and proxies interact with each other. The proxies establish two-way authenticated channels to the server using an attestation handshake. Once such a channel is established, both sides can trust the integrity of the other party's computing environment. Further, the client connects to a proxy using a one-way authenticated channel. Here, only the client can trust the server's integrity, but not the other way around. Because of this, servers must check the databases access policy before handling a clients request.

Once trust, and a secure channel, is established between a proxy and the server, the server can hand over application secrets to the proxy, without harming application integrity. In particular, the server hands off the encryption key for data blocks. The proxy can then bypass the TEE on the server side and access encrypted data blocks directly from the storage layer. This is the key mechanism that allows us to increase performance of the overall system.

We propose a simple push mechanism to notify proxies of updates to the data. Once the server has processed an update, it forwards the changes to the proxies using the authenticated channel. This way, updates to the index are propagated to proxies efficiently, without the need to proxy to poll the storage layer. This mechanism ensures that once an update u_1 has been received by a proxy. It knows that all updates that were processed by the server before u_1 have also been received. Further, because writes are still only processed at the centralized server, this setup maintains strong consistency.

4 Preliminary Prototype Evaluation

Our preliminary prototype follows the design described above. We implemented the full functionality described in the paper, except for the hash verification in the index. The server implementation runs the trusted database server and the untrusted storage layer in the same process. The encrypted storage layer caches data in memory to increase read throughput, but also stores it on disk to achieve durability.

Our benchmark models each client workload individually. Database workloads are often approximated using a single Zipf distribution [2], which models the accumulated load by all clients in the system. In a system where each client has its own proxy, the workload has to be generated on a per-client level. We opt for a simple mathematical model, where each client shifts its Zipf distribution by an individual offset. In the future, we may need to develop more sophisticated mathematical models for such a client centric workload estimation.

In our evaluation up to one hundred clients issue reads to a central server. Each client runs a trusted proxy inside

a TEE. The server process was hosted on a machine running Ubuntu 17.04, equipped with 32GB of RAM and an Intel Core i7 6700K CPU offering 8 logical cores. The current prototype uses version 1.9 of the Intel SGX SDK and is compiled using GNU g++7. The client workload is distributed across multiple machines to make sure only the server's processing power can be a possible bottleneck. Because at the time of the evaluation the number of machines with SGX support available to use is still limited, we evaluate the setup using the simulator provided by the SGX SDK. We observed that simulation mode performs at roughly twice the throughput of hardware. Thus, while the proxies in such a setup might be slower, the server itself also would be and may yield even a higher benefit from offloading its work. To make it a fair comparison, we also run the baseline without proxies in simulation mode. We further provide a second baseline that shows how the system performs without trusted environment at all.

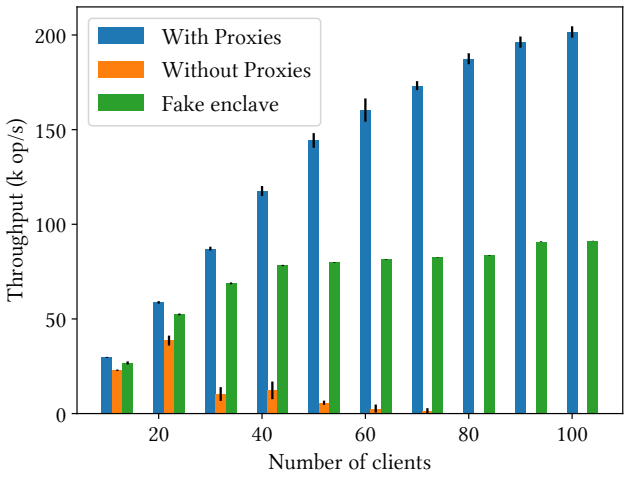

Figure 8. Throughput of read operations. The setup with proxies vastly outperforms that without.

Figure 8 shows the throughput of a proxy setup compared to that without proxies and a setup that does not provide a trusted execution environment. Even with only a handful clients the proxy mechanism performs better. This confirms our assumption that the TEE itself, not the storage layer is the bottleneck. When we increase the number of clients, the proxy setup outperforms that of the non-proxy one by an order of magnitude. After about 40 clients, the non-proxy setup is too congested to make much progress and its throughput drops. Further, we show that the proxy setup, where both clients and server, run in a TEE, performs even better than a setup where the server is not constrained by a TEE at all. We also observed that the proxy mechanism does not add significant network overhead, as only index updates, and not the data itself, are broadcasted to the proxies.

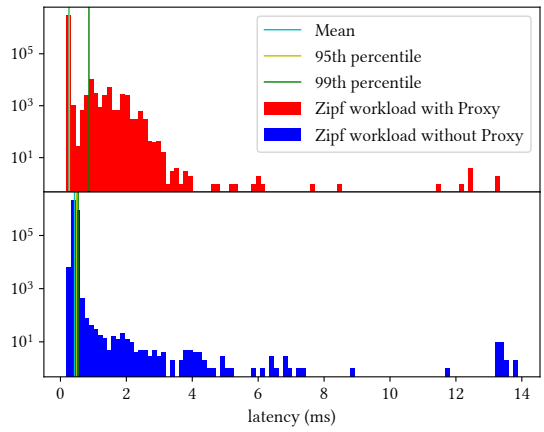

Figure 9. Distribution of latencies under a Zipf workload.

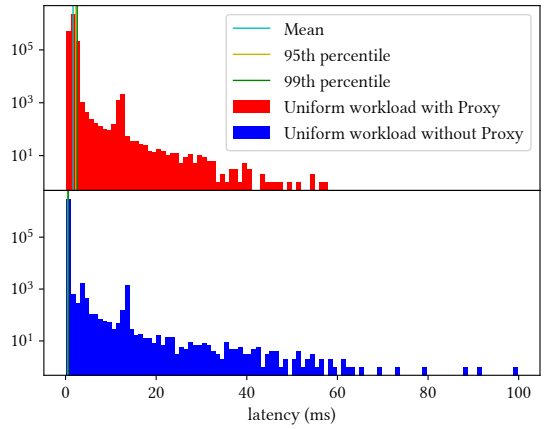

Figure 10. Distribution of latencies under a uniform workload.

We further investigate how the proxy mechanism affects the client latencies using a single client benchmark. This configuration favors the setup without proxies the most. In Figures 9 and 10, we show the resulting latencies for both uniform and Zipf workloads. While the 95 and 99 percentiles are a little higher, due to the additional network hop, the mean of the proxy setup is lower, as more requests can be served from the cache directly. Overall the latency change is negligible.

The setup without proxies usually has very low latencies except for when blocks need to be loaded or evicted. This means that there is a long tail for the latencies. The setup with proxies has higher mean latencies but does not have

this long tail as enough data can be cached at the proxies. We conclude that using the proxy mechanism the latencies are slightly higher but more predicable, than when not using proxies.

5 Conclusion and Future Work

This work first provided a taxonomy for different strategies for outsourcing functionality from a TEE, and then presented a novel approach to how TEE-backed applications are able to scale without comprising any of the TEE guarantees. It showed that this mechanism is compatible with databases running in a TEE itself and can also be beneficial in a setup where the database does not run in a TEE. Further, we showed that the approach is orthogonal to other scaling techniques such as sharding the datastore across multiple machines (NoTE 4).

There are still many open questions on how such a design can accommodate all API calls of a database efficiently. In particular, our design for updates is still very simplistic: whenever the server processes an update, it will forward it to all proxies. We imagine a future system where proxies can subscribe to objects of interest and only get updates about these forwarded to them. Further, a primitive to ensure consistency over a set of operations is needed for future work in this space. An efficient transaction primitive has to be developed that works well with the proxy architecture.

The presented results further suggest that TEEs may enable novel peer-to-peer applications with stronger integrity guarantees. In such a setup, instead of a centralized server that is used as a backbone for all operations, data is sharded across the peer to peer network. Unlike in conventional peer-to-peer applications, nodes using this architecture can rely on other parties to not misbehave.

References

[1] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase.. In *CIDR*. Citeseer.

[2] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1185–1196.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[4] Sumeet Bajaj and Radu Sion. 2014. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2014), 752–765.

[5] Erick Bauman and Zhiqiang Lin. 2016. A Case for Protecting Computer Games With SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX '16)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3007788.3007792

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 267–283. http://dl.acm.org/citation.cfm?id=2685048.2685070

[7] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 189–204.

[8] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM, 295–298.

[9] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. 2017. TrustJS: Trusted Client-side Execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3065913.3065917

[10] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 533–549. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[11] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. 2016. Teechan: Payment Channels Using Trusted Execution Environments. *arXiv preprint arXiv:1612.07766* (2016).

[12] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 369–378.

[13] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 238–253. https://doi.org/10.1145/3064176.3064219

[14] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. 2015. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 13.

[15] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert van Renesse. 2017. REM: Resource-Efficient Mining for Blockchains. *IACR Cryptology ePrint Archive* 2017 (2017), 179.