Running ZooKeeper Coordination Services in Untrusted Clouds

Stefan Brenner
TU Braunschweig
brenner@ibr.cs.tu-bs.de

Colin Wulf
TU Braunschweig
cwulf@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

Abstract

Cloud computing is a recent trend in computer science. However, privacy concerns and a lack of trust in cloud providers are an obstacle for many deployments. Maturing hardware support for implementing Trusted Execution Environments (TEEs) aims at mitigating these problems. Such technologies allow to run applications in a trusted environment, thereby protecting data from unauthorized access. To reduce the risk of security vulnerabilities, code executed inside a TEE should have a minimal Trusted Codebase. As a consequence, there is a trend for partitioning application's logic in trusted and untrusted parts. Only the trusted parts are executed inside the TEE handling the privacy-sensitive processing steps.

In this paper, we add a transparent encryption layer to ZooKeeper by means of a privacy proxy supposed to run inside a TEE. We show what measures are necessary to split an application into a trusted and an untrusted part in order to protect the data stored inside, with ZooKeeper as an example. With our solution, ZooKeeper can be deployed at untrusted cloud providers, establishing confidential coordination for distributed applications. With our privacy proxy, all ZooKeeper functionality is retained while there is little degradation of throughput.

1 Introduction

Recently, there is a significant trend towards cloud computing as it offers dynamic scalability with predictable costs. However, privacy and data security concerns as well as legal reasons are slowing down the widespread adoption of cloud services. These concerns include the unauthorized access by the infrastructure providers themselves.

As a possible approach to solve these issues, chip manufacturers started integrating hardware support for implementing Trusted Execution Environments (TEEs) in their processors, like the upcoming Intel SGX technology [1, 9] or ARM TrustZone [2]. These techniques allow to split applications into trusted and untrusted parts. The former being exclusively executed inside a TEE. Thereby, the trusted parts are responsible for processing sensitive data and can be verified by means of remote attestation. Furthermore, depending on the technology and technical realization the trusted parts can be protected from unauthorized access, including the administrative staff of the cloud provider [1, 9]. In order to strengthen the security of the trusted parts and minimizing overhead due to trusted execution, these parts should possess only a minimal Trusted Codebase (TCB).

Having such TEEs available in cloud environments, offers the opportunity for secure deployment and execution of privacy-sensitive distributed applications. As a basis for these applications, we consider coordination support such as provided by ZooKeeper [8] as an essential basis. For this reason, we investigate the deployment of coordination services in the cloud, exploiting TEEs to make the coordination service itself trustworthy.

So far, various approaches for data storage [12, 5, 6] and relational databases [13, 3] at untrusted providers have been published. In case of untrusted storage works, as the provider is considered entirely untrusted, their primary focus is on integrity. In case of database systems, a much more complex interface needs to be considered when compared with a coordination service. Accordingly, none of the aforementioned solutions builds an ideal fit for partitioning the application logic of a coordination service, and all of them lack support for asynchronous callbacks.

Similar to plain storage systems and databases we consider the data stored in ZooKeeper inherently sensitive, because coordination services are critical parts of all distributed systems. While configuration management may involve confidential data stored as ZooKeeper Node (znode) payload, naming information is also sensitive, since coordination primitives in ZooKeeper often are solely based on the names of znodes.

Hence, in this paper we present the ZooKeeper Privacy Proxy (ZPP), a lightweight and transparent encryption layer for ZooKeeper. Our proxy system mediates the client connections and is deployed inside a TEE. While protecting data and naming information, we retain all original ZooKeeper functionality, including asynchronous callbacks.

With our solution, privacy-preserving ZooKeeper deployments in untrusted cloud environments are possible. This allows sensitive applications in the cloud to use ZooKeeper without privacy concerns. Because we retain all ZooKeeper functionality, we can easily adapt existing ZooKeeper applications to use our ZPPs. Our initial evaluation shows minimal overhead and proves the feasibility of our approach.

In this paper, we first summarize related work in Section 2. Next, we present our system architecture and the rationale of our approach. More details about our implementation are given in Section 4. Thereafter, we investigate possible deployment scenarios, i.e. possible implementations of a TEE. Finally, we detail initial evaluation results in Section 6 and conclude.

2 Related Work

Many researchers have investigated the deployment of different kinds of storage systems in untrusted cloud environments. With TrustedDB [3] a relational database system has been presented that allows the storage of data encryptedly while processing them only inside a trusted subsystem implemented using tamper-proof hardware. The Blind Stone Tablet [13] also offers relational database functionality and offloads durability tasks to an untrusted provider. Clients locally keep the database state completely replicated and forward changes to it as transactions to the durability provider. These transactions are signed and encrypted with a symmetric key, shared between all clients, for providing data privacy. Both works do not provide a suitable callback mechanism, that would allow the implementation of advanced features of ZooKeeper (see “watch callbacks” in Section 3.1) and have to support a complex interface which dictates a larger trusted component.

Venus [12] is an eventually consistent system, that offers wait-free objects to the user. It operates without trusted systems, but uses asymmetric cryptography to establish integrity. Its operations terminate optimistically while consistency is established later on using notifications to clients. FAUST [5, 6] establishes integrity of data by applying client-to-client communication. Same as Venus, FAUST also focuses on data integrity rather than privacy. Furthermore, both systems do not offer a suitable solution for callbacks.

The SPORC [7] system focuses on privacy and involves encryption of all data on the untrusted server, however, each client in this system holds a local full copy of the shared state. In our approach, all state is solely maintained by the original and unchanged ZooKeeper servers, simplifying the architecture of our ZPPs and keeping the original ZooKeeper architecture unchanged.

DepSpace [4] is a more sophisticated approach for dependable and confidential coodination. It is based on tuple spaces and is also Byzantine fault-tolerant. In contrast to our lightweight approach, DepSpace applies a secret sharing scheme in order to achieve confidentiality of tuple space data.

3 Trusted ZooKeeper Proxy

After a brief introduction of ZooKeeper, we describe our approach starting with a system architecture overview. Next, we detail the rationale behind our proxies, and how we transparently encrypt all sensitive data on-the-fly while passing it through the ZPPs.

3.1 ZooKeeper in a Nutshell

ZooKeeper [8] is a fault-tolerant (by replication) coordination service for distributed systems. It allows the implementation of coordination tasks like leader election, locks and much more. For clients it offers a simple interface, that is quite similar to a filesystem. The interface allows to manage so called znodes that are simultaneously like a file and a folder, i.e. all znodes can have child nodes as well as payload at the same time. Special functions like ephemeral znodes, that are deleted once the client that created them failed, or watch callbacks, that notify registered clients about changes to znodes, constitute the powerful API of ZooKeeper.

3.2 Attacker Model and Security Goals

Our main goal is to protect the privacy of all data stored inside ZooKeeper. By this, the fault-tolerance aspects of ZooKeeper are unchanged, because our proxy acts just the same as a normal ZooKeeper client and the server-side ZooKeeper implementation is unchanged.

The attacker may be the cloud provider, as well as anybody taking control over the ZooKeeper replicas. He is considered to be able to access and alter anything that is running in the cloud, i.e. the whole ZooKeeper replicas, but except what is running in a TEE. Our ZPP is supposed to run inside a TEE, also located at the cloud providers data center, which prevents attackers from accessing anything inside it. This allows the storage of encryption keys and processing of cleartext data there.

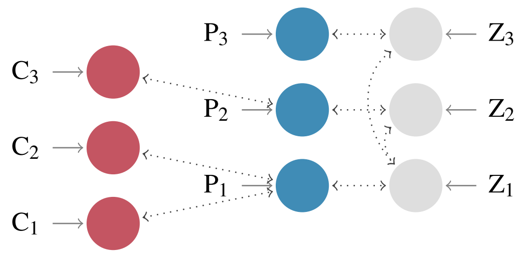

Figure 1: System Architecture

3.3 System Architecture

Figure 1 shows our system architecture, and how we place the ZPPs in between the clients and the ZooKeeper replicas. The main task of the ZPP here is the transparent encryption of all sensitive data, before handing it over to the ZooKeeper replicas. We have several clients C_n connected to one of our ZPPs P_m each, and the ZPPs then are connected to one of the ZooKeeper replicas Z_o. For our work we assume the ZPPs to be running inside a TEE, which can be implemented in several ways (see Chapter 5). All ZooKeeper-internal communication is not affected by our changes, and our ZPPs will appear to ZooKeeper replicas as regular ZooKeeper clients.

For each client session to a ZPPs we are applying a *transport encryption* to the whole packet using an individual session key per client. Once a packet is received, the ZPP extracts it and gathers all sensitive information. The ZooKeeper responses are handled respectively on the way back to the clients. From the client's perspective, a ZPP behaves exactly like a ZooKeeper replica, so the clients can just connect to any of them. The client-side wrapping of messages by a transport encryption is implemented using stunnel[1], which provides us a simple and easy to use SSL encryption.

For each client session, a ZPP will keep a separate connection to a ZooKeeper replica. After a ZPP has received a packet by the client, the ZPP encrypts the sensitive parts of the message (*storage encryption*) and forwards the packet to the ZooKeeper replica. The ZPPs are using an encryption mechanism that allows all ZPPs to read and decrypt the data later. Because of performance benefits, we use symmetric rather than asymmetric encryption here. The encryption is done on all ZPPs using the same key, which is shared between the ZPPs in advance.

Since all ZPPs behave ZooKeeper-compliantly (i.e. only using valid ZooKeeper operations), on the one hand, none of the ZooKeeper replicas will ever notice any changes by our privacy enhancements. On the other hand, we retain all ZooKeeper functionality, and thus, the clients will not experience any difference as well.

3.4 ZNode Payload Encryption

The payload of znodes can be encrypted quite straightforwardly. The ZPP will do an on-the-fly re-encryption here, forwarding the data from the transport encryption between the ZPP and the client, to the storage encryption between all ZPPs and the ZooKeeper replicas.

Encrypting the znode payload may alter the payload's length causing conflicts with its metadata. Then, processing $exists()$ operations, would require the decryption of the payload to measure its cleartext length. However, we mitigate this problem by using a cipher that returns ciphertext of the same length. Another option would be to use helper nodes to store the additional metadata (see Section 4.1).

3.5 ZNode Name Encryption

In order to encrypt the node names of znodes we chose a trivial approach that encrypts the individual znode's names of a path one by one. The advantage is that this approach is completely transparent to ZooKeeper and easy to implement, because ZooKeeper does not require knowledge of cleartext znode names to work properly. In order to mask special characters of the chiphertext like non-printable ones or slashes, we apply a Base64 encoding to the encrypted names.

4 Implementation Details

We implemented the ZPP such that it can be deployed on a wide range of TEE environments. In total, the implementation currently comprises only little more than 4000 SLOCs of C code.

4.1 Sequence Numbers

Sequential nodes basically only differ from regular nodes during the create process; ZooKeeper will append a monotonically increasing sequence number to the node name, which is returned by the *create()* method. This sequence number is maintained by ZooKeeper individually for each parent node: All create operations, not only the ones with sequential flag, will increment the parental node's counter.

ZooKeeper clients can create nodes of arbitrary name; even node names containing self-defined sequence numbers. In the original ZooKeeper, creating znode "node" with sequential flag, returning "node003" and creating the non-sequential node "node003" afterwards, will fail. However, with our encrypted approach, we would not get a "node exists" error, because the node names are different to ZooKeeper and at the same time appear equal to clients. This conflict is depicted in Figure 2.

As can be seen from the figure, both create operations $/node + seqNo$ and $/node001$ succeed, but return the same node name to the client.

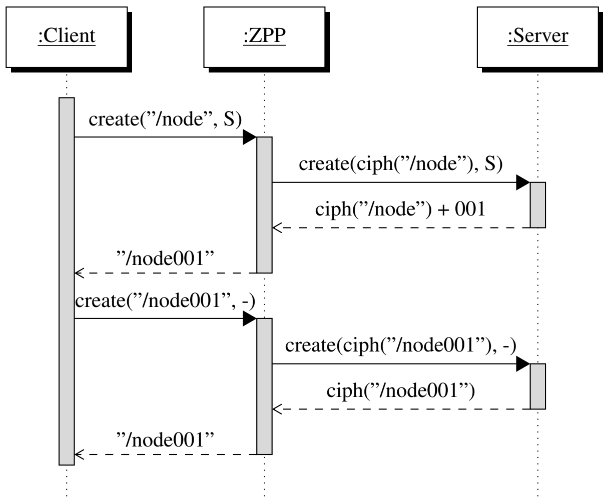

Figure 2: Node name collision situation when creating sequential nodes. *Sequential flag is denoted as “S”, no flag as “-” and the ciphertext of “x” as “ciph(x)”.*

The problem is to distinguish the node name from the sequence number appended by ZooKeeper. However, we can not use a ZPP-controlled delimiter here, that splits the node name from the sequence number, because then we would change the original ZooKeeper's behaviour. Since we do not want to keep state in our ZPPs, we describe our “dictionary”-approach in the following.

The rationale behind our dictionary nodes is to store necessary additional information as (encrypted) payload of a helper node in ZooKeeper in a separate znode namespace. We are using Base64-URL after encrypting node names, so we can easily split the namespace for dictionaries and hide them from clients using non-Base64-URL characters on ZooKeeper's side. Basically, for each node that has some children we keep the next available sequence number in the dictionary node's payload. However, we do not store sequence numbers for nodes that are ephemeral, since they can not have children, and nodes that have no children yet (leaf nodes).

All ZPPs will download the dictionary node at startup and keep the data structure (hash table) containing all sequence numbers in memory. This allows to save an additional Roundtrip Time (RTT). Consistency for dictionary nodes can be established using watch callbacks, so ZPPs get notified once the dictionary changes, and the $multi()$-operation of ZooKeeper, that allows to atomically execute the actual $create()$ and the dictionary node's change. If a single dictionary node does not provide enough capacity, another one could be added in the dictionary namespace.

In Figure 3 we illustrate our solution for implementing sequence numbers in a ZooKeeper-compliant manner. The ZPPs will download the dictionary once they are started and register a watch on it. When a create is executed by a client, we will increment the respective sequence number, create the node and update the dictionary atomically with a conditional $setData()$ operation, that will fail if the dictionary has been updated in the meantime. The dictionary will be altered for each node creation (and $delete()$ calls to the last child of a parent node), even ones without the sequential flag set, in order to behave like the original ZooKeeper implementation.

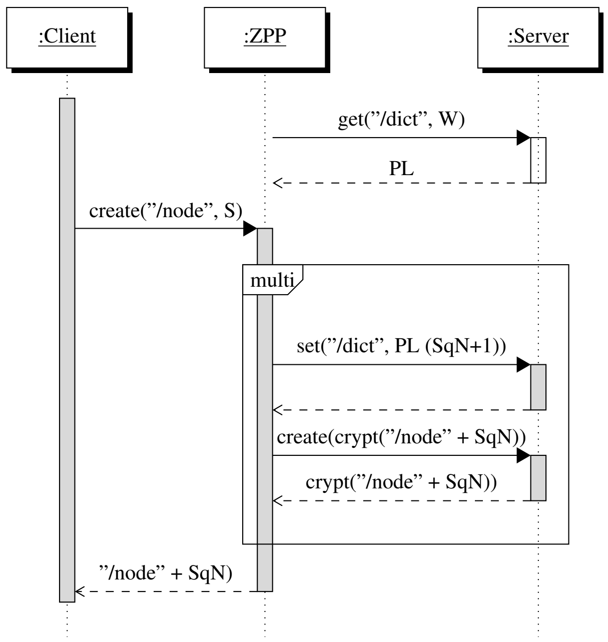

Figure 3: Node creation procedure involving sequence numbers from dictionary nodes. *We denote a watch callback registration as “W”, the dictionary's payload that will include the sequence numbers as “PL”, and the sequence number of the parent node of “/node” as “SqN”.*

4.2 Ephemeral Nodes

Ephemeral nodes are automatically deleted once the client connection of the client that created the node to ZooKeeper closes. We maintain individual connections from a ZPP to a ZooKeeper replica for each client, so we can relay the connection loss through the ZPP to the ZooKeeper replica. A failure of a ZPP, then, will lead to the client choosing another active ZPP just like it would choose another replica once it fails in the original implementation. ZooKeeper replica failures will be relayed through the ZPPs to the clients, so they can choose another ZPP connected to another Zookeeper replica.

5 Deployment Scenarios

In order to deploy the trusted ZPP in the cloud, a TEE has to be available from the cloud provider. This could be implemented applying upcoming hardware isolation technology like Intel SGX [9, 1] or ARM TrustZone [2] for example. Both, TrustZone and SGX, provide an isolated environment which can be remotely attestated, and thus, used for trusted execution. A suitable runtime for *Trustlets*, based on ARM TrustZone has been already presented by Santos et al. [10, 11], however, this approach has been done in a mobile environment. Obviously, it would require several modifications for multi-tenancy in order to be used in a cloud environment.

Another approach for implementing TEEs based on tamper-proof hardware systems has been presented by Bajaj et al. [3]. Here, the trusted system is a PowerPC platform which could be also used to run our ZPPs.

Finally, it would also be possible to deploy the ZPPs on the client side – a trusted environment by definition. However, in this case we would have many ZPPs which could be problematic when it comes to watch callbacks of helper nodes simultaneously on all ZPPs.

6 Evaluation

We evaluated our approach to measure the possible throughput when our ZPPs are mediating the connection between ZooKeeper clients and replicas. Our setup comprises a client Virtual Machine (VM) executing ZooKeeper operations directed to one of the ZPPs in front of normal ZooKeeper replicas. In our setup we place all ZPPs and ZooKeeper replicas inside individual VMs all connected to the same network. All VMs are based on Ubuntu 13.10 and running in OpenStack, the ZooKeeper replicas are equipped with 2 GB memory and 2 VCPUs each, while ZPPs are using a smaller flavor with 512 MB memory with a single VCPU.

First, we execute requests directly to ZooKeeper, and then via our ZPP to ZooKeeper to measure the throughput difference. In order to minimize the impact of background load of our virtualization infrastructure, we process multiple requests in batches and show the average results of 15 repetitions. Operations are executed synchronously and with different payload sizes from 0K to 2.5K. Our evaluated operations comprise several write methods ($create, setData, delete$) and read methods ($getData$) in each iteration.

Figure 4 shows our evaluation of $create()$ and $delete()$ requests. The use of our ZPP slightly decreases the throughput by a constant factor, compared to the throughput directly sending requests to a ZooKeeper replica and without encryption. By using our ZPPs, obviously an additional RTT is added to normal communication, apart from the additional computation time inside the ZPP. However, in our evaluation setup this did not significantly lower the throughput, because we have relatively short RTTs. Since our intention was to bring ZooKeeper to the cloud, and thus, close to its clients (supposed to run in the cloud also), we can realistically assume low local area network RTTs ($< 1ms$ in our setup) between the individual parties. The throughput of $create()$ operations, in contrast to $delete()$, is depending on the payload size of the nodes, because the payload is sent piggy-backed with the $create()$. Hence, a decreasing throughput can be seen when the payload size increases, both with and without our ZPP.

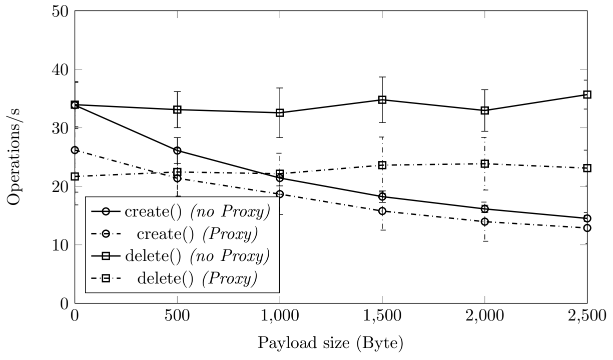

Figure 4: $create()$ and $delete()$ sync. throughput.

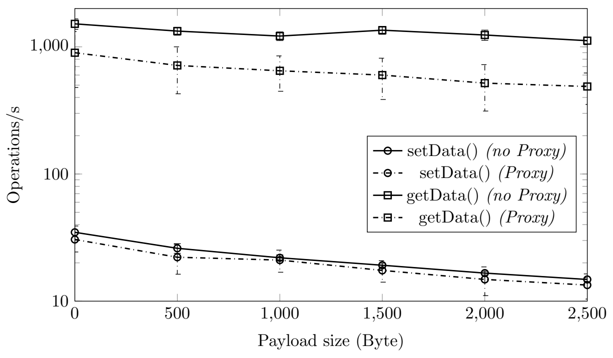

Figure 5: $getData()$ and $setData()$ sync. throughput.

In Figure 5 the throughput of $setData()$ and $getData()$ requests is illustrated. Clearly, the throughput of read-only operations is significantly higher, because no ZAB agreement is involved here. The throughput of $setData()$ is again a write operation, that requires ZAB agreement inside ZooKeeper, and thus, is much lower. Again, the use of our ZPP only slightly decreases throughput for both kinds of operations. Both operations are affected by the payload size of the respective nodes, thus, we can see decreasing throughput same as in Figure 4 here as well.

We also evaluated the execution of asynchronous requests in the same way. Throughput of $create()$ and

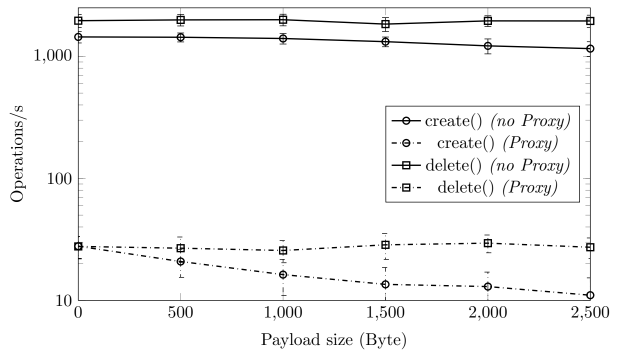

Figure 6: *create*() and *delete*() async. throughput.

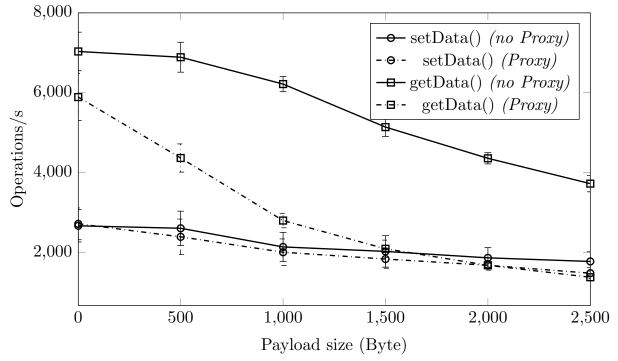

Figure 7: *getData*() and *setData*() async. throughput.

delete() (Figure 6) is similar to the synchronous case, because these operations require a dictionary access, which causes the operations to be relayed synchronously to the ZooKeeper replicas. Finally, Figure 7 shows the throughput of *setData*() and *getData*() operations in the asynchronous case, were increased impact of the additional network RTT for *getData*() can be seen.

7 Conclusion and Ongoing Work

In this work we describe our ZooKeeper Privacy Proxy (ZPP) that allows to establish secure coordination services in untrusted cloud environments. Our system adds a transparent encryption layer to the ZooKeeper cluster, that encrypts all sensitive data inside ZooKeeper on-the-fly. On the one hand, the ZPP acts as a surrogate ZooKeeper replica for the clients. On the other hand, it will forward client requests to the original ZooKeeper replicas once all sensitive data has been encrypted. We encrypt the payload of ZooKeeper nodes, as well as the node names and we retain all the functionality and behaviour of a regular ZooKeeper cluster.

However, our current solution still allows the inference of usage details based on client access patterns and the znode hierarchy. A possible approach to mitigate this problem is to apply the concept of helper nodes to store the znode's hierarchy, and store the znodes themselves in a flat hierarchy in ZooKeeper.

The evaluation based on our prototype implementation essentially proves the feasibility of our idea and shows that it only causes little throughput degradation.

References

[1] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative Technology for CPU Based Attestation and Sealing. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[2] ARM TrustZone. `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[3] BAJAJ, S., AND SION, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering* (2013).

[4] BESSANI, A. N., ALCHIERI, E. P., CORREIA, M., AND FRAGA, J. S. Depspace: a byzantine fault-tolerant coordination service. In *In Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems* (2008).

[5] CACHIN, C., KEIDAR, I., AND SHRAER, A. Fail-Aware Untrusted Storage. *Proceedings of the International Conference on Dependable Systems and Networks* (2009).

[6] CACHIN, C., KEIDAR, I., AND SHRAER, A. Fail-aware untrusted storage. *SIAM Journal on Computing* (2011).

[7] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (2010).

[8] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference* (2010).

[9] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[10] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (2011).

[11] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[12] SHRAER, A., CACHIN, C., AND CIDON, A. Venus: Verification for Untrusted Cloud Storage. *Proceedings of the 2010 ACM workshop on Cloud computing security* (2010).

[13] WILLIAMS, P., SION, R., AND SHASHA, D. The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties. *Proceedings of Network and Distributed Systems Security Symposium* (2009).

Notes

[1]`https://www.stunnel.org/index.html`