2019 IEEE European Symposium on Security and Privacy (EuroS&P)

ReplicaTEE: Enabling Seamless Replication of SGX Enclaves in the Cloud

Claudio Soriente
NEC Labs Europe
claudio.soriente@emea.nec.com

Ghassan O. Karame
NEC Labs Europe
ghassan.karame@neclabs.eu

Wenting Li
NEC Labs Europe
wenting.li@neclabs.eu

Sergey Fedorov
NEC Labs Europe
sergey.fedorov@neclabs.eu

***Abstract*—With the proliferation of Trusted Execution Environments (TEEs) such as Intel SGX, a number of cloud providers will soon introduce TEE capabilities within their offering (e.g., Microsoft Azure). The integration of SGX within the cloud considerably strengthens the threat model for cloud applications. However, cloud deployments depend on the ability of the cloud operator to add and remove application dynamically; this is no longer possible given the current model to deploy and provision enclaves that actively involves the application owner. In this paper, we propose ReplicaTEE, a solution that enables seamless commissioning and decommissioning of TEE-based applications in the cloud. ReplicaTEE leverages an SGX-based provisioning service that interfaces with a Byzantine Fault-Tolerant storage service to securely orchestrate enclave replication in the cloud, without the active intervention of the application owner. Namely, in ReplicaTEE, the application owner entrusts application secret to the provisioning service; the latter handles all enclave commissioning and decommissioning operations throughout the application lifetime. We analyze the security of ReplicaTEE and show that it is secure against attacks by a powerful adversary that can compromise a large fraction of the cloud infrastructure. We implement a prototype of ReplicaTEE in a realistic cloud environment and evaluate its performance. ReplicaTEE moderately increments the TCB by ≈ 800 LoC. Our evaluation shows that ReplicaTEE does not add significant overhead to existing SGX-based applications.**

***Index Terms*—TEE, SGX, replication, cloud**

I. Introduction

The cloud has been recently gaining several adopters among SMEs and large businesses that are mainly interested in minimizing the costs of deployment, management, and maintenance of their computing infrastructure. Cost effectiveness is realized in the cloud by coupling multi-tenancy with so-called elastic deployment strategies that ensure unprecedented levels of scalability at low costs.

With the recent proliferation of Trusted Execution Environments (TEEs) such as Intel SGX, a number of cloud providers will soon introduce TEE capabilities within their offering (e.g., Microsoft Azure [2]). Using TEEs within the cloud allows the design of secure applications (e.g, [21], [28], [27]) that can tolerate malware and system vulnerabilities, as application secrets are shielded from any privileged code on the same host.

Although the integration of SGX within the cloud considerably strengthens the threat model for cloud applications, the current model to deploy and provision an enclave, prevents the cloud operator from adding or removing enclaves dynamically—thus effectively hampering elasticity. Namely, elastic deployment assumes that the cloud operator can swiftly add or remove replicas of an application (be it a VM or an enclave) depending on factors such as current load or throughput. Yet, the deployment model of SGX prevents a cloud provider to dynamically start new enclaves. That is, SGX enclaves bear no secrets when deployed; secrets are securely provisioned to the enclave by the application owner after he attests the application code and makes sure that it runs untampered in an enclave on an SGX-enabled platform. In a nutshell, dynamic deployment for TEE-based applications in the cloud requires the application owner to be online throughout the whole application lifetime. The only alternative for an application owner is to entrust the secrets of his application to the cloud provider. This, however, obviates the shift to deploy SGX enclaves in the cloud since it exposes all application secrets to malware that may potentially penetrate the cloud infrastructure.

Although the community features a number of studies on SGX security in the cloud [24], [10], [11], no previous work has addressed the problem of enabling seamless commissioning and decommissioning of enclaves in the cloud. Here, there are a number of challenges to overcome. One the one hand, such a service should remove the need of an online application owner. On the other hand, it should warrant owners the same security provisions of the current provisioning model, where owners attest and provision secrets to their applications.

Further, the number of running replicas of a given applications must be controlled, since unrestricted enclave replication may amplify the effectiveness of *forking attacks* [10]. In a forking attack, the adversary runs several instances of an application and provides them with different state or inputs to influence their behavior. For example, consider an authentication service running in SGX enclaves. To mitigate brute-force attacks, the service may use rate-limiting and, for example, allow up to 3 password trials per account. An adversary that manages to compromise the cloud infrastructure could launch several instances of the service in order to increase the number of trials per account and brute-force passwords. A service that automatically provisions enclaves must, therefore, control the number of running enclaves for a given application at all times, despite malware that may penetrate the cloud infrastructure.

The second author has been supported by the CyberSec4Europe European Union's Horizon 2020 research and innovation programme under grant agreement No 830929.

DOI 10.1109/EuroSP.2019.00021

In this paper, we propose ReplicaTEE, a solution that enables dynamic enclave commissioning and decommissioning for TEE-based applications in the cloud. ReplicaTEE leverages a distributed SGX-based provisioning service that interfaces with a Byzantine Fault-Tolerant (BFT) storage service to orchestrate secure and dynamic enclave replication in the cloud. Namely, in ReplicaTEE, the application owner entrusts application secrets to the provisioning service and can go offline. The provisioning service is a distributed service that runs entirely in SGX enclaves and assists the cloud to add or remove enclaves of an application on behalf of the application owner. Application secrets are, therefore, shielded away from malware that penetrates the cloud, as they are securely transferred from the application owner to the provisioning service, onto application enclaves.

The provisioning service also controls the number of running enclaves for a given application, in order to mitigate forking attacks against victim applications. Yet, a forking attack against the provisioning service itself, may allow an adversary to run an arbitrary number of enclaves of a victim application (and therefore launch a forking attack against the victim). We prevent forking attacks against the provisioning service by leveraging a distributed BFT storage service that guarantees dependable storage despite compromise of a fraction of its nodes. We design the provisioning service to duly store onto the storage service any operation regarding commissioning and decommissioning enclaves so to constantly control the number of running enclaves for each application. As a result, ReplicaTEE protects confidentiality of application secrets against an adversary that can compromise privileged code on the cloud's platforms. Forking attacks against applications are mitigated as long as the number of compromised nodes in the storage service remains below a tuneable threshold.

We design ReplicaTEE to be compliant with the existing Intel SGX SDK. We also implement a prototype of ReplicaTEE in a realistic cloud environment and evaluate its performance. Our evaluation shows that ReplicaTEE increments the TCB by approximately 800 Lines of Code (LoC) and does not add significant overhead to existing SGX-based applications.

The remainder of this paper is structured as follows. In Section II, we review Intel SGX and BFT storage solutions that leverage TEEs. In Section III, we introduce our system and threat models, we discuss our design goals, and provide a brief overview of our solution. In Section IV, we provide details of ReplicaTEE and analyze its security. In Section V, we evaluate a prototype implementation of ReplicaTEE within a realistic cloud environment. We review related work in Section VI, and we conclude the paper in Section VII.

II. Preliminaries

We now briefly review Intel SGX and we outline existing Byzantine Fault-Tolerant storage protocols that leverage TEEs.

A. Intel SGX

Software Guard Extensions (SGX) is the latest realization of Trusted Execution Environment by Intel, available on Skylake and later CPUs. It allows application to run in secure containers called *enclaves* with dedicated memory regions that are secured with on-chip memory encryption. Access to the encrypted memory is mediated by the hardware, effectively excluding the OS or any other software from the Trusted Computing Base (TCB).

Privileged code on the planform can create and add data to an enclave with instructions `ECREATE`, `EADD`, `EINIT`. After creation, the enclave code can only be invoked using a thin interface via instructions `EENTER` and `ERESUME`; enclave code returns by calling `EEXIT`, which ensures that any sensitive information is flushed before control is given back to the OS.

State persistence across reboots is available through *sealing*, i.e., hardware-managed authenticated and confidential persistent storage. Enclaves can use instructions `EREPORT` and `EGETKEY` to retrieve an enclave-specific (and platform-specific) key to encrypt data before writing it on persistent storage. Keys are uniquely bound to the identity of an enclave so that no other software including no other enclave can access them.[1] Note that the sealing functionality that offers SGX does not ensure freshness. That is, a malicious OS may present stale state information to an enclave, what is commonly referred to as a *rollback attack* [29]. This is in part mitigated by the use of monotonic counters provided by the platform. However, monotonic counters are apparently slow and the registries where they are stored wear out with usage [24].

SGX allows a remote party to verify that a piece of code runs in an enclave on an SGX-enabled platform. This mechanisms, called remote attestation, uses a tailored group signature scheme [14] scheme that provides platform anonymity, i.e., the verifier is assured that the enclave runs on an SGX platform without being able to tell it apart from other SGX platforms. Remote attestation in SGX is a two-step process. During the first step, the enclave to be attested proves its identity to a system enclave present on every platform and called *quoting enclave*. The latter has access to the signing key and produces a publicly verifiable *quote* that allows the verifier to remotely attest the enclave. In its current implementation, attestation involves an Intel service (Intel Attestation Service, IAS) that mediates communication between quoting enclaves and remote verifiers. In particular, the IAS only allows registered parties to issue remote attestation requests. Also, the quote produced by a quoting enclave is encrypted under the IAS public key, so that only the IAS can verify a quote. The IAS then signs a publicly verifiable statement to confirm that the enclave runs on an SGX platform. As a by-product of the attestation protocol, the prover and the verifier establish a mutually authenticated Diffie-Hellman key. In particular, the verifier signs its ephemeral key and the enclave must hold the corresponding verification key to verify the signature. Also, the quote signed by the quoting enclave guarantees that the prover's ephemeral key belongs to the specific enclave being attested.

[1] Keys may also be bound to a "sealing authority" in order to allow secure storage across different versions of the same application.

B. Byzantine Fault-Tolerant Storage using TEEs

The community features a large number of Byzantine Fault-Tolerant protocols (BFT) [17], [6], [16] based on state replication across different nodes, called "replicas". Some replicas may be faulty and their failure mode can be either *crash* or *Byzantine* (i.e., deviating arbitrarily from the protocol [22]). Classical BFT protocols require $3f + 1$ nodes and $O(n^2)$ communication rounds among these nodes in order to tolerate up to f Byzantine nodes.

Since agreement in classical BFT is rather expensive, prior work has attempted to improve performance by leveraging trusted hardware. Namely, previous work showed how to use trusted hardware to reduce the number of replicas and/or communication rounds for BFT protocols [8], [20], [30]. For example, MinBFT [30] is an efficient BFT protocol that reduces the communication rounds and the number of replicas used by conventional BFT protocols, by leveraging functionality from TEEs, such as Intel SGX. As a result, the number of required replicas is reduced from $3f+1$ to $2f+1$. In MinBFT writers send *write* requests (e.g., using a PUT interface) to the replicas, which are all expected to execute the requests in the same order (i.e., maintain a common state). Readers can read content previously written onto the replica nodes. The main idea of MinBFT is to rely on the sequential monotonic counter provided by trusted hardware, in order to bind each message sent to a unique counter value. This is ensured by requiring a signature from the local TEE on all messages sent by the replica; the intuition is that the TEE will sign messages with a given counter value only once, thereby preventing replicas from assigning the same counter value to different messages—commonly referred to as *equivocation*. More details about MinBFT can be found in Appendix A.

III. Model & Overview

A. System Model

We consider a scenario where a cloud provider manages a set of SGX-enabled platforms. Application owners can upload code to be executed on such platforms. Applications could either run computation on behalf of their owners such as a map-reduce service [27], or provide public functionalities such as an online password-strengthening service [21].

Deployment. In a real-world deployment of ReplicaTEE, application owners would acquire (e.g., rent) VMs at the cloud and split the logic of their applications (e.g., by using available tools [23]) in sensitive code to be run in an enclave and non-sensitive code that can run inside the VM. Therefore, each of the cloud platforms would host VMs from different tenants and each VM would have one or more enclaves. However, for the sake of simplicity, we assume in this paper that the entire application code is executed in enclaves. Given this assumption, each of the cloud platforms hosts multiple enclaves belonging to different owners.

Dynamic Provisioning. In line with current elastic cloud settings, we assume that multiple *instances* of the same application enclave may dynamically be started or shut down. In the following, we use the term *application enclave* to refer to an instance of application code running in an enclave, and we use *application* to denote the logical entity spanning multiple enclaves running the same code.

We are agnostic on how the decision to add or remove application enclaves for a given application is made. For example, this decision may be taken by the cloud for reasons such as load, throughput, or efficient resource utilization. Alternatively, the application itself may monitor its performance and, when needed, ask the cloud to add or remove instances.

B. Threat Model

The goal of the adversary that we consider is two-fold. On the one hand, the adversary is interested to leak the secrets of the applications. On the other hand, the adversary might also be interested in deploying a large number of application enclaves in order to amplify the effect of a forking attack against a victim application.

The adversary can compromise privileged code on a node and we denote that node as *compromised*. Throughout the paper, we include SGX in the TCB and therefore assume that the adversary cannot compromise SGX components (e.g., system or application enclaves) on the compromised node. Namely, we do not take into account attacks specific to SGX, such as the ones that exploit side-channels [12]. Measures to mitigate attacks against SGX [26], [18] are orthogonal to ReplicaTEE and could be deployed alongside our solution.

Although we do not consider DoS attacks in this paper, we assume that the adversary controls the network and as such controls the scheduling of all transmitted messages.

C. Overview

To the best of our knowledge, there is no mechanism that enables enclave replication in the cloud in a way that is transparent to the application owner. Clearly, a cloud provider can autonomously start an arbitrary number of application enclaves as long as they do not require any secret material. However, if the enclavesrequires a secret key (e.g., for applications like Talos [7] or SecureKeeper [13]), the enclave owner must be involved in the enclave deployment process for attestation and secret provisioning.

Alternatively, application owners may entrust the secrets of their applications to the cloud. Nevertheless, this option is in sharp contrast with the settings of SGX where enclave secrets are to be hidden from any other software on the host. In other words, if application owners trust the cloud with handling their secret data, then SGX becomes unnecessary.

Strawman solution. A strawman solution to the problem that we consider would be to create a provisioning service that runs entirely in an SGX enclave and acts on behalf of the application owner when the cloud must deploy new application instances. One may have a single provisioning service per cloud, or a provisioning service serving multiple clouds.

Here, an application owner uploads the code of its application to the cloud. At the same time, the owner attests the provisioning service, call it Enclave Management Service

(EMS), and transfers to it the hash of the application along with the application secrets. When a new application instance needs to be started, the cloud sets up the instance and asks EMS for attestation and secret provisioning. As a result, the cloud and EMS can deploy new instances of an application enclave while the owner is no longer required to be online. Further, application secrets are shielded from the cloud (and from malware that penetrates the cloud) since secrets are securely transferred from the application owner, to EMS (which runs in an enclave) to the target application (which runs in an enclave). However, this strawman solution suffers from the following shortcomings.

Highly availability.
: EMS must be highly available because no new application instances can be started when EMS is down, and fast commissioning and decommissioning of enclaves is key to the elastic operations of the cloud provider.

Forking attacks.
: EMS should not allow for unrestricted deployment of instances. For example, an adversary may compromise the cloud and deploy a large number of instances of a victim application in order to mount a forking attack. The provisioning service must, therefore, control at all times the number of deployed instances of a given application; if this number reaches a given threshold, no further deployment requests should be served.[2] Controlling the number of running application enclaves requires EMS to keep *state*. Otherwise, a forking attack against EMS itself, would allow the adversary to launch an arbitrary number of application instances to, in turn, mount a forking attack against the victim application. We note that monotonic counters available to SGX enclaves[3] may be suitable for centralized applications to keep state. Monotonic counters, however, are not suited to keep state of an application (like EMS) distributed across different hosts. Similarly, ROTE [24] is a distributed solution to keep state of single-enclave applications and cannot be used when state must be synchronized across enclaves. The challenge is, therefore, how to keep a consistent state across all of the EMS enclaves.

Small TCB.
: The only effective and workable way to securely maintain state for an application that spans multiple instances, while assuming a potentially malicious OS, would be to leverage a reliable consensus mechanism. One option would be to fit the consensus logic within EMS. However, this design choice leads to a large code-base which, in turn, becomes a large attack surface. That is, a large footprint of the enclave code that implements EMS, essentially weakens the assumption that enclaves cannot be compromised.

We tackle the above challenges as follows. We design ReplicaTEE as a two-tiered approach. At the first tier, EMS (i) acts on behalf of application owners and supports the cloud in commissioning and decommissioning application enclaves, and (ii) mitigates forking attacks against application enclaves by controlling the number of running enclaves for a given application. EMS is a distributed SGX-based service that leverages a master-slave approach to ensure high availability. Master-slave is arguably the simplest distributed architecture and its small code-base allow us to fit its entire logic within an enclave. Master and slaves exchange beacons to monitor one another; in case the master fails, one of the slaves becomes the new master.

At the second tier, a BFT Storage Service (BSS) provides EMS with reliable storage and allows to prevent forking attacks against EMS itself (which, in turn, may lead to forking attacks against applications). We opt to separate EMS from the consensus logic and create BSS as a dedicated service in order to keep EMS code-base small. Yet, the consensus logic is complex and fitting it entirely in an enclave may open the door to exploits. A better option is to design consensus that leverages the isolated execution feature made available by SGX while, at the same time, keeps the enclave code at bare minimum. When instantiating the consensus service, we therefore resort to TEE-based consensus protocols like MinBFT [30] that can tolerate up to $\frac{n-1}{2}$ out of n faulty nodes but still have a very small TCB ($\approx$ 250 LoC).[4] Nevertheless, as the consensus logic is now split between enclave and non-enclave code, we must account for an attacker that compromises the part of the logic running outside of SGX. In other words, the nodes of the consensus service may now become Byzantine. We show that by carefully designing the interaction between EMS and BSS, a master-slave application like EMS can be shielded from forking attacks.

A note on enclave termination. The above overview covers enclave provisioning. However, controlling the number of running enclaves for an application, requires EMS to also be aware of the enclaves that terminate. We note that, given our threat model, there is no means for EMS to tell if an application enclave has been stopped by C. We tackle this issue by implementing a lease-based approach. When provisioned, enclaves receive an end-of-lease timestamp and should stop running if that time is reached and EMS has not renewed the lease. In other words, we do not rely on the cloud to terminate enclaves. The length of a lease is a tunable parameter and represents a trade-off between security and overhead due to lease renewal.

[2]The threshold may be set by the application owner as part of the deployment policy that usually allows owners to decide parameters such as maximum load per instance, geographical deployment restrictions, etc.

[3]https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter

[4]Traditional consensus protocols that do not leverage TEEs can only tolerate $\frac{n-1}{3}$ out of n faulty nodes.

IV. Protocol Specification

We start by outlining the process of remote proxied attestation which constitutes an essential building block that will be used in our solution.

A. Remote Attestation by Unregistered Verifiers

As mentioned in Section II, the Intel Attestation Service (IAS) controls that remote attestation is not abused by verifiers and, in particular, that SGX platforms are not tracked—which constitutes one of the main goals of the signature scheme underlying SGX attestation [14]. Nevertheless, involving IAS as an intermediary in each remote attestation protocol limits the adoption of this mechanism, especially by parties who are not registered with IAS. This limitation becomes especially relevant if the enclave runs a public service like a mail server. Indeed, it is rather unrealistic to assume that all users interested in setting up a mail account are registered to IAS; yet, users may want to attest the mail server and ensure it runs in an enclave on an SGX-enabled platform.

In order to overcome this limitation and enable remote attestation with unregistered verifiers, we utilize a proxy registered to IAS. The proxy can be deployed by the cloud provider or by a third-party. Our proxied attestation protocol is depicted in Figure 1. There, we only provide an overview of the protocol; detailed message contents refer to the ones defined in the Intel SGX SDK Developer Reference [4]. Attestation via our proxy comes in two flavors, depending on whether the prover enclave "knows" (i.e., holds the public key of) the remote verifier. If the verifier is known to the prover, the proxy simply relays messages between prover and verifier; when the prover outputs and encrypted quote, the proxy (registered to IAS) forwards the ciphertext to IAS in order to get back the cleartext and provides the latter to the verifier. In case the verifier is unknown to the prover, the proxy also signs the ephemeral DH key chosen by the verifier. Therefore, the prover enclave must embed the public key of the proxy.[5]

Our proxied attestation protocol allows any party to remotely attest an enclave and to establish an unilaterally or mutually authenticated DH key—depending on whether the identity of the verifier is known to the prover.

Note that our protocol is compliant with the standard attestation protocol that leverages the SDK provided by Intel for SGX and only requires the enclave developer to include the public key of the proxy, in cases where attestation requests are expected from unknown verifiers. Also, note that a malicious proxy trying to man-in-the-middle the key establishment between the prover and the verifier may be easily detected by the latter. This is because the ephemeral key of the prover is authenticated by the quoting enclave.

B. ReplicaTEE Protocol Details

Setup. Recall that ReplicaTEE is made of two service: an enclave provisioning service named EMS and a BFT storage service named BSS. We assume that BSS is setup initially by the provider conforming to the setup procedure of MinBFT [30]. The setup of EMS unfolds as follows. The cloud provider C (or a third party) starts N enclaves, each running an instance of EMS. The enclaves must attest each other and agree on a group key for secure group communication. For this task, we require each EMS enclave to be aware of the identity of its peers (in order to attest them) and of the number N of enclaves that form EMS. We denote by k_{EMS} the established group key. Note that attestation enables authenticated key exchange. That is, SGX attestation ensures that only an instance of EMS enclave running in an SGX environment can participate in the key agreement protocol. Once EMS has been set up, the enclaves jointly generate a key-pair for a signature scheme and publish the verification key. Application owners must embed this key in their applications, in order to enable application enclaves to verify the legitimacy of the messages received from EMS during attestation.

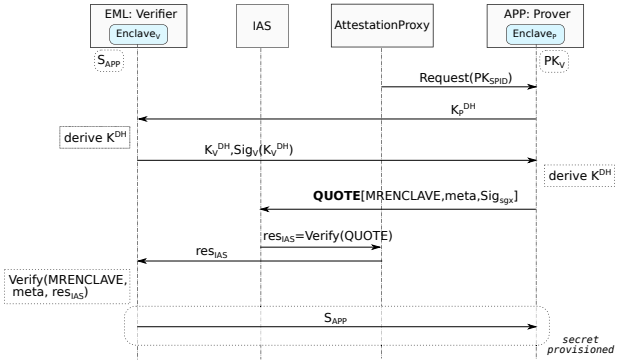


Fig. 1. Proxied attestation protocol. The DH ephemeral key of the verifier (K_V^{DH}) is signed either by the verifier itself (as shown in the figure) or by the proxy. In the former case, the enclave must have the public key of the verifier. If the ephemeral key is signed by the proxy, the enclave must have the public key of the proxy.

EMS enclaves are organized in a Master-Slave approach. By default, the master enclave is the enclave that has the largest enclave identifier. During normal operation, the master is in charge of carrying out the main operations in EMS while slaves simply assume a passive role.

The master EMS implements a variant of the so-called "node guarding" protocol to keep track of the availability of the slaves; this essentially consists of the master exchanging alive messages with the slaves at regular intervals. The master enclave periodically sends a beacon request to the slaves to transmit information about their current state (e.g., stopped, active). If a slave does not respond to the request of the master within a certain timeout, the master considers the slave to be crashed and relays this information to the remaining slaves. On the other hand, the slaves also use this protocol to monitor availability of the master; if a request from the master is not received after a certain timeout, the slaves assume that the master itself has failed. In this case, the slave with the highest identifier from the set of active slaves, assumes the role of

[5]Remote attestation using the standard SDK requires the ephemeral DH key of the prover to be signed and it also requires that the prover has the corresponding public key.

a master and starts issuing the beacon requests. This process ensures a continuous operation of EMS in spite of potential crash failures. Needless to mention, this entire lightweight protocol runs within the enclaves of EMS nodes. Further, if one of the EMS nodes crashes, it can be restarted and it can recover its state (e.g., k_{EMS}, EMS node endpoints, etc.) from the BSS storage service.

Notation. We denote an application and its binary by α and b_α, respectively. Also, p_α denotes the deployment policy defined by the application owner. In this paper, we assume the owner simply sets an upper bound to the number of enclaves that can run simultaneously. However, ReplicaTEE can be easily extended to account for more complex deployment policies.

EMS assigns identifiers to applications and enclaves. An identifier for an enclave of application α looks like $eid = \alpha||mr_\alpha||h_\alpha$, where mr_α is the MRENCLAVE of the application, and h_α corresponds to the hash of the key established between EMS and the enclave during attestation.

In order to keep track of applications and enclaves, EMS leverages the storage functionality offered by BSS. For each application α, EMS keeps track of the following metadata:

1) p_α: Upper bound to number of running enclaves.
2) mr_α: MRENCLAVE.
3) k_α: Application secret key.
4) enc_α: A list of tuples (eid, key, st, eol) where eid is an enclave identifier, key is the key established between EMS and the enclave during attestation, st is a status variable, and eol is the current end-of-lease timestamp for that enclave. Variable st can take values in {att, run, tbd, tbs, sus}. An enclave has status "attested" (att) after being attested by EMS. The status is changed to "running" (run) when the enclave is provisioned with the application secret key. The enclave status is set to "to be deleted" (tbd) or "to be suspended" (tbs) when the cloud requests the enclave to be deleted or suspended, respectively. Finally the status is set to "suspended" (sus) when the enclave has been suspended.

EMS exports the identifiers of applications and enclaves to the cloud C in order to efficiently manage enclaves for a given application. Note that application and enclave identifiers do not bear any sensitive information apart from the number of enclaves running for a given application—an information already available to the cloud.

We assume that the integrity and the confidentiality of data written/read to BSS (via PUT/GET) are protected by means of an authenticated encryption scheme. The key material for authenticated encryption is derived from the key shared by all EMS enclaves, namely k_{EMS}, by using a suitable key-derivation function.

We use application identifiers as the keys to the storage service and for each application we store a "flat" database to keep information of that application and its enclaves. In order to ease exposition, we slightly overload the PUT/GET interface as follows. We write GET(α; $attr$) to fetch only the value of attribute $attr$ for application α. Similarly,

Algorithm 1 Attestation of the EMS Service and Initial Upload of Code

```
1: [APPLICATION OWNER]
2: function ATTEST-UPLOAD
3:     k ← PROXIEDATTESTATION(e_EMS, mr_EMS)
4:     if k ==⊥ then
5:         return -1
6:     end if
7:     SEND ⟨p_α, mr_α, k_α⟩ to e_EMS
8:     SEND ⟨p_α, b_α⟩ to C
9: end function
10: [EMS ]
11: function INITIALIZE(α, p_α, mr_α, k_α)
12:     PUT(α; p_α, mr_α, k_α, ⊥)
13: end function
```

PUT(α; $attr := value$) sets $attr$ to $value$, leaving all other attributes at the same key unchanged. We also write GET(α; $enc_\alpha : eid$) to fetch only the enclave information related to eid (i.e., key, st, eol) from the list enc_α. Also, PUT(α; $enc_\alpha : \langle eid', key', st', eol' \rangle$) writes to the list of enclaves enc_α of application α; if enc_α already has a tuple with $eid == eid'$, this operation only updates the remaining fields to key', st', and eol', respectively. If enc_α has no tuple with $eid == eid'$, then a new tuple $\langle eid', key', state', eol' \rangle$ is appended. We stress that this notation is only to improve the readability of our pseudocode. In reality, we always read and write the whole data associated to a given key.

Attestation of EMS Service and Initial Upload of Code. Algorithm 1 lists the main steps carried out when an application owner wants to upload his application to the cloud. Before the application owner can entrust the management of his application to EMS, he must verify the identity of the EMS enclave and establish a secure channel. This is captured by the function PROXIEDATTESTATION(e_{EMS}, mr_{EMS}) that takes as input the endpoint of the enclave to be attested and the expected MRENCLAVE. The function returns the key established with the prover enclave, if attestation is successful; otherwise it signals an error by returning $\perp$.

Once the application owner has established a secure channel with EMS, he uploads p_α, mr_α, k_α to EMS and p_α, b_α to C. EMS writes p_α, mr_α, k_α, $\perp$ to BSS while the cloud stores p_α, b_α. Both EMS and C send an acknowledgement message to the application owner.

From this moment on, the application owner goes offline, while EMS cooperates with C in order to increase or decrease the number of enclaves allocated to that application. C can, at any time, issue requests to EMS to deploy or remove an enclave. Similarly, C can ask to suspend a running enclave or resume a previously suspended enclave. EMS writes requests to storage in order to serialize them. Then, EMS periodically reads from BSS in order to identify pending requests and dispatch them.

Algorithm 2 Deployment Request

```
1: function PROVISION_REQUEST(α, e)
2:     mr_α ← GET(α; mr_α)
3:     k_α,EMS ← PROXIEDATTESTATION(e, mr_α)
4:     if k_α,EMS ==⊥ then
5:         return -1
6:     end if
7:     h_α ← H(k_α,EMS)
8:     eid ← α||mr_α||h_α
9:     PUT(α; enc_α : ⟨eid, k_α,EMS, att, ⊥⟩)
10:    SEND(ack, α, e, eid) to C
11: end function
```

Algorithm 3 Termination Request

```
1: function TERMINATE_REQUEST(eid)
2:     Parse eid as α||mr_α||h_α
3:     ⟨key, st, eol⟩ ← GET(α; eid)
4:     if st == run then
5:         PUT(α; enc_α : ⟨eid, key, tbd, eol⟩)
6:     end if
7:     SEND(ack, tbd, eid) to C
8: end function
```

Deployment Request. At this stage, the cloud provider creates a new enclave e on an SGX platform and loads the code b_α. It then contacts the EMS enclave that is acting as master to trigger the attestation and provisioning of the enclave. The pseudocode of the steps carried out is provided in Algorithm 2. Upon receiving a request, EMS enclave attests the application enclave (line 3) and assigns it an identifier made of the application identifier, the enclave identity, and the hash of the key established with that enclave during attestation (line 8). Next, EMS enclave writes to storage tuple $\langle eid,\ k_{\alpha,\text{EMS}},\ \text{att},\ \perp\rangle$ to reflect the fact that enclave eid was attested and it is ready for provisioning. Finally, EMS enclave acknowledges to C the end of the operation. If C does not receive an acknowledgement within a given timeout, then C may infer that the EMS enclave handling the request has crashed and that the request should be issued to another EMS enclave.

Termination/Suspension/Resumption Requests. The pseudocode to terminate, suspend or resume an enclave is provided in Algorithms 3, 4 and 5, respectively. Requests are invoked by C providing the enclave identifier eid as an argument. The EMS enclave handling the request extracts the application identifier from eid and fetches from BSS attributes key, st, eol of enclave eid. For enclave termination, the EMS enclave checks that st is "run" and sets it to "tbd" (i.e., to be deleted). For enclave suspension, the EMS enclave checks that st is "run" and sets it to "tbs" (i.e., to be suspended). For enclave resumption, the EMS enclave checks that st is "sus" and sets it to tbr (i.e., to be run).

For termination and suspension of an enclave, EMS only takes note of the request by setting the status variable of that specific enclave; the operation is actually completed at the beginning of the next lease. This is because, as we argued above, there is no guarantee that the cloud is effectively terminating or suspending the enclave at the time of the request. However, the enclave will stop working at the end of the current lease.

Algorithm 4 Suspension Request

```
1: function SUSPENSION_REQUEST(eid)
2:     Parse eid as α||mr_α||h_α
3:     ⟨key, st, eol⟩ ← GET(α; eid)
4:     if st == run then
5:         PUT(α; enc_α : ⟨eid, key, tbs, eol⟩)
6:     end if
7:     SEND(ack, tbs, eid) to C
8: end function
```

Algorithm 5 Resumption Request

```
1: function RESUMPTION_REQUEST(eid)
2:     Parse eid as α||mr_α||h_α
3:     ⟨key, st, eol⟩ ← GET(α; eid)
4:     if st == sus then
5:         PUT(α; enc_α : ⟨eid, key, tbr, eol⟩)
6:     end if
7:     SEND(ack, tbr, eid) to C
8: end function
```

For enclave resumption, once again EMS persists the request to storage by setting the status variable of that specific enclave; the enclave will be resumed by the main routine of EMS that dispatches provisioning and resumption requests persisted to storage (see next).

Enclave Provisioning/Resuming. The pseudocode to dispatch requests to provision or resume enclaves is shown in Algorithm 6. This code is periodically executed by the EMS enclave acting as master. Function FINDNEXT(enc_α) on line 3 takes as input the list of tuples storing information about the enclaves of application α and returns the first tuple $\langle eid,\ key,\ st,\ eol\rangle$ such that the status variable st is either "att" or "tbr". Status "att" means that the enclave has been attested and it is ready to be provisioned with the application secret key. Status "tbr" reflects a suspended enclave that must be resumed. Before dispatching the request for eid, the EMS enclave checks that the number of running enclaves is below the upper bound set by application owner and that provisiong/resuming eid does not violate the owner's constraints. Counting is carried out by function COUNTRUNNING(enc_α) on line 5. An enclave is considered as running if its status variable is set to "running, "to be suspended", or "to be deleted". Next, EMS enclave either provisions eid with the application secret key and the current end-of-lease timestamp, or it sends to eid a "resume" directive with the current end-of-lease timestamp. EMS writes to BSS that the enclave has been served and notifies C.

From this moment on, the application enclave runs as expected, e.g., executing computation on behalf of the application owner or serving requests from clients. However,

Algorithm 6 Dispatch

```
1: function RUN(α)
2:     ⟨p_α, mr_α, k_α, enc_α⟩ ← GET(α)
3:     ⟨eid, key, st, eol⟩ ← FINDNEXT(enc_α)
4:     if eid! =⊥ then
5:         if COUNTRUNNING(enc_α) < p_α then
6:             if st == att then
7:                 SEND(k_α, eol) to eid
8:             else                                  ▷ st == tbr
9:                 SEND(resume, eol) to eid
10:                PUT(α; ⟨eid, key, run, eol⟩)
11:            end if
12:            SEND(ack, run, eid) to C
13:        end if
14:    end if
15: end function
```

Algorithm 7 Lease Renewal

```
1: function RENEW(α)
2:     ⟨p_α, mr_α, k_α, enc_α⟩ ← GET(α)
3:     for ⟨eid, key, st, eol⟩ in enc_α do
4:         if st == tbs then
5:             PUT(α; ⟨eid, key, sus, eol⟩)
6:         else if st == tbd then
7:             DELETE(enc_α, eid)
8:         else if st == run && eol < eol′ then
9:             SEND(renew, eol′) to eid
10:            PUT(α; enc_α : ⟨eid, key, run, eol′⟩)
11:        end if
12:    end for
13: end function
```

we require the application to halt its execution if the current time has passed the current end-of-lease timestamp received by EMS. A secure source of time is currently available on all SGX platforms via the `sgx_get_trusted_time()` API.

Lease Renewal. The pseudocode shown in Algorithm 7 is run by the EMS enclave acting as master when the current end-of-lease timestamp is approaching. At this stage, the EMS enclave scans through the list of enclaves belonging to application α and checks their status in order to determine whether the application must be suspended (line 4-5), deleted (lines 6-7), or whether its lease must be renewed. In the latter case, the application enclave receives the new end-of-lease timestamp eol' with a "renew" directive. Regardless of the operation, the EMS enclave pushes the changes to BSS in order to persist the fact that the request was handled. Note that function DELETE(enc_α, eid) on line 7 removes from enc_α the tuple referring to eid and writes the updated list of tuples to storage.

C. Dealing with Application Shared State

Recall that some applications need to keep state to ensure its correct operation. Indeed, in a model where the cloud runs applications that span several enclaves, a shared storage service might be required. This is because the sealing functionality of SGX is designed only to keep *local* state and does not allow state to be shared across enclaves. In this case, newly provisioned enclaves should maintain a consistent view of such a state—otherwise the security of the overall service might be at risk. For example, in S-NFV [28], the adversary could run two separated instances of the application and route state updates only to one instance, while exclusively pushing traffic flows to either instances. Hence, the outcome of processing a given flow may be different and dependant on whether it is carried out by one instance or the other. Similarly, password-strengthening services like Safekeeper [21] rely on rate-limiting to keep passwords secret. Having access to multiple isolated application instances, allows the adversary to infringe the restriction imposed by the rate-limiting policy.

ReplicaTEE's BSS can be used by such applications to share consistent state among their enclaves. Namely, whenever needed, authorized applications in ReplicaTEE can read/write their latest state from/to the storage service using the offered PUT/GET interface. That is, our storage service acts as consistent storage medium for various application enclaves to synchronize on their latest application state. For example, an enclave providing password strengthening service can continuously write the number of trials attempted on the storage service. This allows to enforce rate-limiting across all application enclaves running the same service. In Section V we complement the evaluation of ReplicaTEE by assessing the overhead of using a BFT storage service for applications that span across several enclaves.

D. Security Analysis

As mentioned in Section III-B, the adversary's goal is to leak secrets of victim applications or launch an arbitrary number of application enclaves in order to mount a forking attack against that application.

Application secrets. We note that application secrets (e.g., a secret key) are transferred from the owner to EMS and finally to the application enclave.

Before transferring the secret to EMS, the application owner must attest the EMS enclave and establish a secure channel with it. This is done by leveraging the proxied attestation protocol of Section IV-A. At this time, the EMS enclave cannot attest the application owner (since the two parties may have not had any previous interaction). Therefore, EMS accepts application metadata (i.e., the application secret key, the policy, etc.) from *any* party. Nevertheless we assume the cloud to authenticate application owners and that only authenticated application owners can contact EMS. This is a reasonable assumption since the cloud must authenticate application owners in order to bill them.

Once the application owner has securely uploaded the application secret key to EMS, the security provisions of SGX guarantee the confidentiality of that key while it is stored in the memory of the EMS enclave. If written to storage, the key is encrypted and authenticated with keys that are only available to EMS. Finally, EMS securely delivers the key to the application enclave after attesting its code and establishing

a secure channel. Here again, attestation may use the proxied protocol of Section IV-A. We assume an application enclave to hold the public key of EMS so that the former can authenticate the latter.— in other words, the application enclaves only accepts provisioning from EMS.

We therefore conclude that ReplicaTEE keeps confidentiality of application secrets at all times, despite an adversary that can compromise privileged code running on the hosts of the cloud infrastructure.

Deployment threshold. We now analyze how ReplicaTEE ensures that the number of running enclaves for a given application is always below the threshold set by the application owner. Controlling the number of application instances, allows to mitigate forking attack against that application.

ReplicaTEE controls the number of running instances of an application by ensuring that any progress made by EMS while serving a deploy/decommission request is always registered as an event onto the storage service. The latter implements a consistent BFT service, thereby ensuring total ordering of the events. This design tolerates possible asynchrony or network partitioning that could arise in EMS. Namely, since EMS enclaves do not run a consistent protocol (they only execute a lightweight node guarding protocol), consistency is guaranteed by the facts all operations handled by EMS enclaves are duly registered on a consistent storage service. The number of running enclave of an application is, therefore, controlled by EMS as long as EMS can rely on the correctness of the storage service. Since ReplicaTEE instantiates a BFT service that leverages SGX, correctness is guaranteed up to $\frac{n-1}{2}$ compromised BSS nodes. That is, ReplicaTEE ensures that the number of running application enclaves for a given application respects the deployment threshold as long as no more than $\frac{n-1}{2}$ storage nodes are compromised. In what follows, we explain this in greater detail.

Provisioning/Resuming. Provisioning of an enclave eid is only executed after the enclave has been attested and the request has been registered by writing the tuple $\langle eid,\ key,\ \text{att},\ \perp\rangle$ to BSS (Algorithm 2, line 9). Similarly, resuming of enclave eid only occurs after the request has been registered by writing the tuple $\langle eid,\ key,\ \text{tbr},\ eol\rangle$ to BSS (Algorithm 5, line 5). For both cases, the tuple written to storage carries the key established with the eid at the time of attestation—hence,any EMS enclave can establish a secure channel with the application enclave and carry out the request.

Provisioning or resuming is carried out by Algorithm 6. Since BSS ensures that write/read operations are serialized, no other enclave will be provisioned or resumed before the request for eid is dispatched. This holds despite the fact that the EMS enclave in charge of handling the request for eid, say e_{EMS}, may fail, and despite the fact that multiple EMS enclaves may concurrently act as masters.

If eid is to be provisioned and e_{EMS} fails right after provisioning the application enclave (Algorithm 6, line 7), the new master EMS enclave will use the same secret key key to establish a secure channel with eid and provision the application secret key once again. Similarly, if eid is to be resumed and e_{EMS} fails right after sending the "resume" command (Algorithm 6, line 9), the new master EMS enclave will use the same secret key key to establish a secure channel with eid and send once again the "resume" command. We stress that provisioning or resuming *the same* enclave does not violate the security provisions of ReplicaTEE.

We point out that even if two (or more) EMS enclaves acting as masters take in charge the request at the same time, they will both provision (or resume) eid. Also, they will both write the tuple $\langle eid,\ key,\ \text{run},\ eol\rangle$ (Algorithm 6, line 10) to BSS. Once again, provisioning/resuming *the same* enclave and writing *the same* tuple to storage does not bring ReplicaTEE to an inconsistent state.

Only after the enclave status is set to "run" in BSS, EMS enclaves will start provisioning/resuming another enclave. Hence, provisioning/resuming of enclaves is carried out in strict sequential order so that EMS enclaves can be always aware of the running enclaves for a given application.

Terminating/Suspending. As discussed before, once C issues a request to terminate or suspend an enclave eid, there is no guarantee that the enclave has been effectively deleted or suspended. This is due to the fact that any attempt from EMS to contact eid may be dropped by the adversary that controls the communication network. For this reason, we resort to leases and require application enclaves to stop as soon as the current lease expires, unless EMS renews it.

EMS therefore treats an enclave eid as suspended and sets its status accordingly (Algorithm 7, line 5) only at the end of the lease. At the time EMS receives the request to suspend eid, it simply writes the request to BSS by setting eid's state to "to be suspended" (Algorithm 4, line 5). However, the enclave is considered as running until the end of the current lease. A similar approach is taken for requests to delete an enclave eid. The request is written to BSS by setting eid's status to "to be deleted" (Algorithm 3, line 5), however the enclave will be considered as running until the end of the current lease. Once the current lease expires, the enclave metadata is deleted from storage (Algorithm 7, line 7).

Note that enclaves considered as running (i.e., the ones with status set to "running", "to be suspended", or "to be deleted") affect the decision of whether a request to provision/resume an enclave should be honored. That is, an application enclave is provisioned/resumed only if the number of enclaves considered as running for that application is below the threshold set by the application owner (Algorithm 5).

Lease Renewal. At the end of a lease, EMS proceeds to renew the lease to all application enclaves with status "running". If the EMS enclave carrying out the operation crashes after renewing the lease to a given enclave eid, but before writing to BSS that the operation was completed (Algorithm 7, line 9), then eid will receive the same renewal message from another EMS enclave taking up the master role. Repeating the lease renewal operation by issuing the same end-of-lease timestamp to the same enclave does not constitute a security breach.

Application	LoC
MinBFT	339
Proxied attestation (prover)	200
Proxied attestation & provisioning (verifier)	800
DupLESS integrated with ReplicaTEE	80

TABLE I
LINES OF CODE (LOC) OF THE ROUTINES IN OUR PROTOTYPE

V. PERFORMANCE ANALYSIS

A. Implementation Setup

We deployed the storage service of ReplicaTEE on five identical servers with SGX supports. Each server is equipped with Intel Xeon E3-1240 V5 (8 vCores @3.50GHz) and 32 GiB RAM. The EMS instances were deployed on a machine with Intel Core i5-6500 (4 Cores @3.20GHz) and 8 GiB RAM. All these machines are equipped with SGX to run enclaves and are connected with a 1Gbps switch in a private LAN network. We argue that this setting emulates a realistic cloud deployment scenario where the compute servers and their corresponding storage servers communicate over the cloud's private LAN (e.g., Amazon AWS and S3).

As mentioned earlier, we instantiate the atomic storage service of ReplicaTEE using MinBFT. Our implementation of MinBFT uses 2 interface functions (`createUI`, `verifyUI` [30]) and a total 339 LoC of enclave code.[6] We argue that this is small enough to make formal verification of the consensus service code base as needed. In our evaluation, we relied on HMAC-SHA256 to achieve authentication between replicas and clients [15], [30].

We implemented the proxied attestation procedure described in Section IV-A based on the libraries provided by the SGX SDK [4]. The prover's code in the enclave requires around 200 lines, while the verifier's code in the EMS enclave is around 800 lines (cf. Table I).[7]

B. Evaluation Results

In what follows, we evaluate the performance of ReplicaTEE. Namely, we measure the latency incurred in the provisioning of enclaves and in termination, suspension, resumption and lease renewal. Note that we do not evaluate the overhead incurred in the initial setup phase of EMS and the initial code upload by application owners, since the setup is carried out only once and the overhead for application owners to upload their code to the cloud is not particular to ReplicaTEE and is incurred by all applications that leverage cloud-based SGX deployments.

We also measure the latency incurred in the provisioning of enclaves with respect to the achieved throughput. We measure the throughput as follows. The master EMS enclave invokes operations in a closed loop, i.e., enclaves may have at most one pending operation. We require that the master EMS enclave performs a series of back-to-back operations (requests) and measure the end-to-end time taken by each operation. We then increase the number of provisioning requests in the system until the aggregated throughput is saturated.

Enclave Provisioning. In Figures 2(a) and 2(b), we evaluate the throughput vs latency for the enclave provisioning process given different storage failure threshold f. We see that when $f = 1$ (3 storage servers), the system achieves a peak throughput of 85 op/s with a latency of 270 ms. On the other hand, when $f = 2$ (5 storage servers), the latency remains almost the same, while the peak throughput is reduces to 75 op/s. Our findings suggest that the remote attestation process is the dominant factor in the operation latency. Notice that even if increasing the fault-tolerance threshold of BSS reduces the peak throughput (since it requires more communication rounds), it has limited impact on the witnessed latency.

In Figure 2(c), we further measure the constituent latencies incurred in the enclave provisioning process. In both cases when $f = 1$ and $f = 2$, we see that the time for remote attestation is around 260 ms while the state update only takes 10 ms without noticeable difference in either cases. Namely, the state update only comprises up to 3.7% of the whole provision process even when $f = 2$.

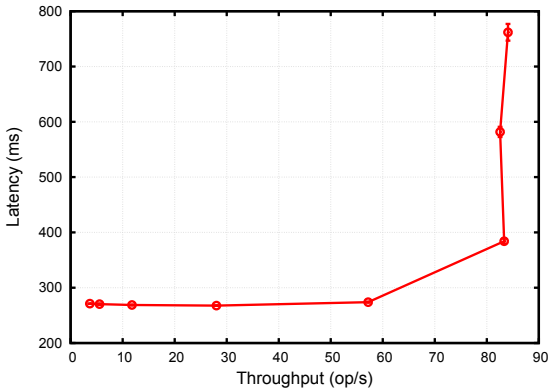
Termination/Suspension/Resumption/Renewal Requests. Recall that termination, suspension, resumption, and renewal requests basically consist of the EMS enclave updating the records corresponding to the target enclave on the storage service. These requests are practically instantiated by a PUT request issued by the EMS primary enclave to update the associated record. Such requests only take 0.86 ms with a peak throughput of 9800 op/s or 4700 op/s when $f = 1$ or $f = 2$, respectively.

DupLESS instantiation. In Figure 2(d), we evaluate the performance overhead incurred by ReplicaTEE on applications that require shared mutable state for their correct operation. To this end, we implement a variant of DupLESS [9] and integrate it with ReplicaTEE in the case where $f = 1$. DupLESS is a server-aided encryption scheme that enables data deduplication over encrypted data. In this scheme, users interested in deduplicating their files first contact the DupLESS gateway to obtain an encryption key that is derived to the file digest. This key is essentially a blind signature on the file digest that allows client to obtain encryption keys while keeping privacy of their files. By using a deterministic encryption scheme and a key derived from the file digest, two users with the same file will produce the same ciphertext that, as such, can be deduplicated by a storage service. By involving the gateway in the key generation process, brute-force attacks on predictable files can only be slowed down by rate-limiting the requests to the server. In our variant implementation, we integrate DupLESS's blind signature scheme within SGX enclaves and use it as an exemplary application of ReplicaTEE.[8] Namely, we rely on

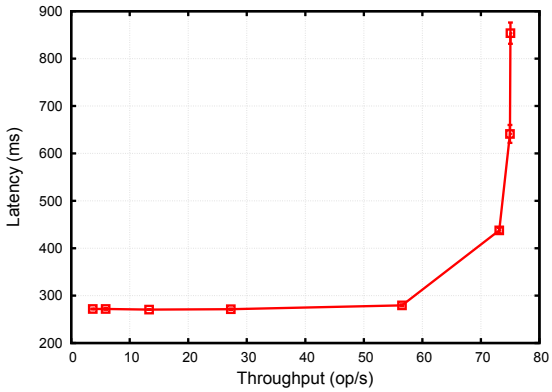
[6]We contrast this to Paxos (based on LibPaxos [31]) which requires around 4,000 LoC.

[7]The verifier enclave also includes JSON and Base64 decoder libraries [5], [3] in order to decode the response from IAS.

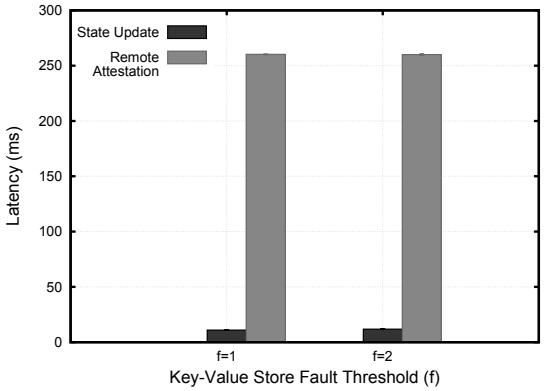
[8]We chose DupLESS because it incurs minimum I/O and allows us to clearly evaluate the computational overhead of ReplicaTEE.



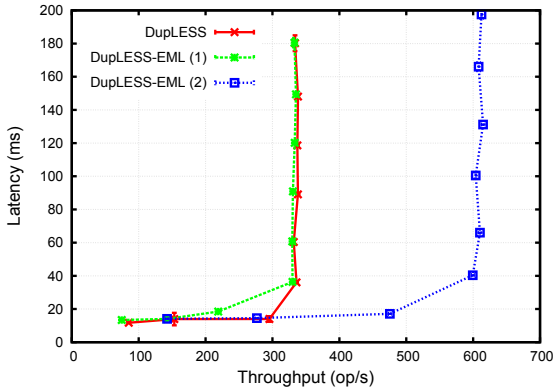
(a) Throughput vs. latency for enclave provisioning when $f = 1$.



(b) Throughput vs. latency for enclave provisioning when $f = 2$.



(c) Latency witnessed in the enclave provisioning process of ReplicaTEE.



(d) Throughput vs. latency for DupLESS w/ and w/o integration with ReplicaTEE.

Fig. 2. Evaluation of the performance of ReplicaTEE in our setup. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

ReplicaTEE to automatically commission and decommission DupLESS enclaves and to allow running enclaves to synchronize on their latest state to effectively enforce rate-limiting across all running enclaves. Since DupLESS leverages RSA-based blind signatures, we utilize the SGX-SSL library [1] to implement the signing functionality (with 4096-bit RSA) with ˜80 lines of code. We deploy the DupLESS servers on a machine with Intel Xeon E3-1240 V5 and evaluate the overhead introduced by ReplicaTEE in this setting when compared to a standalone DupLESS gateway that does not leverage any functionality from SGX (i.e., the standard DupLESS gateway described in [9]).

Our results show that the latency incurred by a standalone DupLESS gateway is 18 ms with a peak throughput of 330 op/s. On the other hand, integrating a single DupLESS instance in ReplicaTEE achieves almost the same performance. This confirms that ReplicaTEE does not add significant overhead to existing SGX-based enclaves. Notice that adding an additional DupLESS enclave almost doubles the peak throughput by reaching around 600 op/s (for 2 DupLESS instances). The throughput exhibited by a distributed DupLESS instantiation will be however limited by the peak throughput exhibited by BSS which is roughly 9800 op/s; in this case, BSS can accommodate for roughly 30 DupLESS instances.

VI. Related Work

To the best of our knowledge, no previous study has addressed the problem of enabling seamless replication of SGX enclaves in the cloud. We now briefly review related work in the area.

Gu *et al.* [19] provide an SDK to enable enclave migration in the cloud. Here, enclaves are augmented with a thread that carries out state transfer. The thread in the source enclave brings other threads to a quiescent state and ships the internal state to the target enclave; a thread in the target enclave receives the state, installs it and recover execution. Since some state information is only available to the platform, the authors use a number of heuristics to estimate that part of the state and transfer it to the target platform. The authors show that their heuristic are indeed effective in few application scenarios. However, the effectiveness of this heuristic for general SGX applications remains to be assessed.

Matetic *et al.* [24] proposed a scheme, ROTE, to enable rollback protection for SGX enclaves. Recall that the sealing functionality of SGX provides confidentiality and integrity but does not guarantee freshness of sealed data. In a rollback attack, a malicious host leverages this shortcoming to provide enclaves with stale state information. In ROTE, a set of *ROTE Enclaves* running on different platforms, help *one* application enclave to maintain monotonic counters that, when used in conjunction with the sealing functionality of SGX, provide state freshness. The set of ROTE enclaves is static and must be setup by an administrator before applications can leverage the service. Notice that ROTE does not deal with applications that span across several enclaves and requires that the application enclave runs on one of the platform that hosts ROTE enclaves.

ICE [29] is another proposal that addresses rollback attacks in SGX. Differently from ROTE, ICE is a "standalone" solution that relies on hardware modifications to the platform.

Brandenburger *et al.* [10] address forking attacks on TEEs in scenarios where multiple clients interact with an enclave running at a malicious host. In order to counter forking attacks, they require an enclave to create a hash chain with the history of all performed operations. When combined with monotonic counters shared with all clients, such an approach can ensure fork linearizability [25].

Proxied attestation was first proposed in [21]. Here, the proxy is registered with IAS and acts on behalf of the (unregistered) verifier towards the IAS. Notice that [21] leverages a *proactive* attestation scheme where the enclave itself requests a quote from the platform and binds it to its ephemeral DH key *before* seeing the ephemeral DH key of the verifier. This design saves round-trips during attestations but is not compliant with the SDK of Intel SGX; namely, a quote is provided *after* the ephemeral DH key of the verifier has been received and a shared key established.[9] Therefore, the scheme of [21] requires application developer to update their code in order to account for changes in the attestation protocol. Furthermore, the attestation protocol proposed in [21] only provides an unilaterally authenticated DH key exchange, since the enclave cannot be sure that the ephemeral DH key is the one chosen by the verifier and not by the proxy. Mutually authenticated DH key exchange would require the enclave to embed the verification key of the verifier. However, this is not viable if the enclave is meant to be verified by any (previously unseen) user of the cloud service.

VII. Conclusion

In this paper, we presented a novel solution, ReplicaTEE, that enables dynamic commissioning and decommissioning of TEE-based applications in the cloud. ReplicaTEE leverages an SGX-based provisioning service that interfaces with Byzantine Fault Tolerant storage service to orchestrates dynamic application replication in the cloud without the active intervention of the application owner. We showed that ReplicaTEE withstands a powerful adversary that can compromise a large fraction of the cloud infrastructure. By means of a prototype implementation, we also showed that ReplicaTEE moderately increments the TCB and does not add significant overhead to existing applications. ReplicaTEE, therefore, emerges as the first secure and practical solution to support elasticity of TEE-based applications in the cloud.

[9]The data structure providing the quote is referred to as `msg3` in the SDK[4] which is returned by `sgx_ra_proc_msg2()` that processes the ephemeral DH key of the verifier and a valid signature on that ephemeral key.

References

[1] "Intel software guard extensions ssl," https://github.com/intel/intel-sgx-ssl, 2017.

[2] "Introducing Azure confidential computing," https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017.

[3] "Implementation of base64," https://goo.gl/6kB48E, 2018.

[4] "Intel software guard extensions (intel sgx) sdk," https://software.intel.com/en-us/sgx-sdk, 2018.

[5] "Jsmn – minimalistic json parser," https://github.com/zserge/jsmn, 2018.

[6] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2658994

[7] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves," Imperial College London, Tech. Rep. 2017/5, 2017.

[8] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. ACM, 2017, pp. 222–237. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064213

[9] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage." *IACR Cryptology ePrint Archive*, vol. 2013, p. 429, 2013.

[10] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *International Conference on Dependable Systems and Networks, DSN*, 2017, pp. 157–168.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[12] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[13] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel SGX," in *International Middleware Conference*, 2016, pp. 1–14.

[14] E. F. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Conference on Computer and Communications CCS*, 2004, pp. 132–145.

[15] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[16] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, "PoWerStore: Proofs of writing for efficient and robust storage," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516750

[17] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *Principles of Distributed Systems: 18th International Conference*, 2014.

[18] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017, pp. 217–233.

[19] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *International Conference on Dependable Systems and Networks, DSN*, 2017, pp. 225–236.

[20] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient Byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168866

[21] K. Krawiecka, A. Kurnikov, A. Paverd, M. Mannan, and N. Asokan, "Protecting web passwords from rogue servers using trusted execution environments," in *International Conference on World Wide Web, WWW*, 2017, pp. 1–16.

[22] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[23] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *USENIX Annual Technical Conference, (USENIX ATC)*, 2017, pp. 285–298.

[24] S. Matetic, ansoor Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: rollback protection for trusted execution," in *USENIX Security Symposium, USENIX Security*, 2017, pp. 1289–1306.

[25] D. Mazières and D. E. Shasha, "Building secure file systems out of byantine storage," in *Symposium on Principles of Distributed Computing, PODC*, 2002, pp. 108–117.

[26] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX Annual Technical Conference, (USENIX ATC)*, 2017, pp. 227–240.

[27] 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, S&P*, 2015, pp. 38–54.

[28] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: securing NFV states by using SGX," in *International Workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFV@CODASPY*, 2016, pp. 45–48.

[29] R. Strackx, B. Jacobs, and F. Piessens, "ICE: a passive, high-speed, state-continuity scheme," in *Annual Computer Security Applications Conference, ACSAC*, 2014, pp. 106–115.

[30] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine fault-tolerance," *IEEE Transactions on Computers*, Jan 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6081855/

APPENDIX A
MINBFT

MinBFT comprises four routines and unfolds as follows:

1) **Request**: Clients send their request messages asking the replicas to execute certain operations. A client $\mathcal{C}$ prepares its requested operation op in message $\langle \mathsf{REQUEST}, \mathcal{C}, seq, op \rangle_{\sigma_\mathcal{C}}$, where seq records the (local) message sequence from each client to prevent re-execution of the operations, and $\sigma_\mathcal{C}$ is the client signature.
2) **Prepare**: This phase is triggered when the primary $\mathcal{S}_p$ receives a request message m. Once the request is validated, the primary asks its TEE to generate a unique message identifier $UI_p = \langle c, m \rangle_{\sigma_p}$. Note that the counter c is monotonically increasing and the signature σ_p is from the TEE. Subsequently, $\mathcal{S}_p$ multicasts $\langle \mathsf{PREPARE}, v, \mathcal{S}_p, m, UI_p, \rangle$ to the other replicas.
3) **Commit**: This phase serves to acknowledge a valid PREPARE message. Each replica $\mathcal{S}_i$ responds with a COMMIT message. In particular, each replica multicasts $\langle \mathsf{COMMIT}, v, m, \mathcal{S}_i, UI_i, \mathcal{S}_p, UI_p \rangle$, where UI_i is a unique identifier that S_i gets from its TEE.

Client		Gateway
$h \leftarrow H(M)$		
$r \xleftarrow{R} \mathbb{Z}_N^*$		
$x \leftarrow h \cdot r^e \mod N$	$\xrightarrow{x}$	
	$\xleftarrow{y}$	$y \leftarrow x^d \mod N$
$z \leftarrow y \cdot r^{-1} \mod N$		
If $z^e \mod N \neq h$ then ret $\perp$		
Else ret $K \leftarrow H(z)$		

Fig. 3. RSA blind-signature scheme adapted from [9]. $H : \{0,1\}^* \rightarrow \mathbb{Z}_N$ denotes a hash function, N the RSA modulus, e the RSA public exponent and d the RSA private exponent.

4) **Reply**: A request is *committed locally* and can be executed once a replica has received enough (i.e., $f + 1$) consistent commits, because it is ensured that any request that commits locally on a correct replica will be committed on at least $f + 1$ correct replicas eventually. Therefore, the replica can execute the operation op and send the reply $\langle \mathsf{REPLY}, \mathcal{S}_i, seq, res \rangle$ with the execution result res back to the client.
5) **View-Change**: When a primary is suspected to be misbehaving, a replica can request a replacement of the primary through the view-change procedure. For example, when a received request failed to be executed within a certain timeout, a replica multicasts a view-change request $\langle \mathsf{REQ-VIEW-CHANGE}, \mathcal{S}_i, v, v' \rangle$, where v' is the new view number and $v' = v + 1$. If a replica receives $f + 1$ $\mathsf{REQ-VIEW-CHANGE}$, it moves to view v'. At this stage the replica multicasts $\langle \mathsf{VIEW-CHANGE}, \mathcal{S}_i, v', CP, O, UI_i \rangle$, where CP is the latest certificate and O is the set of all messages sent by the replica since CP. Once the new primary of view v' receives $f + 1$ valid $\mathsf{VIEW-CHANGE}$ messages with consistent system state, the view change is executed by the new primary who broadcasts message $\langle \mathsf{NEW-VIEW}, \mathcal{S}_{p'}, v', V_{vc}, s, UI_{p'} \rangle$, where V_{vc} is the view-change certificate that includes all the received $\mathsf{VIEW-CHANGE}$ messages, and s is the current system state which will serve as the initial state of view v'.

The correctness of MinBFT holds as long as there is at least one honest node involved in any two quorums, thus only $2f+1$ replicas are required to tolerate f faulty nodes. Further details on MinBFT can be found in [30].

APPENDIX B
DUPLESS

DupLESS [9] allows clients to derive encryption keys for secure deduplication in cloud-based storage. Key derivation is performed in DupLESS by means of an interactive protocol between a client and a gateway based on RSA blind-signatures. The protocol is sketched in Figure 3. The client secret input is a file M, while the server secret input is the private exponent of an RSA key-pair. The corresponding public exponent is available to both parties. The client computes the hash of the file M and blinds it with a random value r that he raises to

the public exponent e. He transmits the blinded hash value to the gateway. The gateway now signs the blinded value with its private exponent d. The gateway finally transmits the signed blinded hash back to the client. As $ed \equiv 1 \mod \varphi(N)$, we have that $y \equiv (hr^e)^d \equiv h^d r^{ed} \equiv h^d r \mod N$. The client can compute the $r^{-1} \mod N$, remove the blinding from y and obtain the signed hash $h^d \mod N$. The client needs now to check the validity of the signature using the public exponent of the gateway e. If the signature is valid, the generated symmetric key will be the hash of the signed hash of the file

$K = H(z) = H(h^d)$

The benefits of such a key generation protocol are two-fold:

- Since the protocol is oblivious, it ensures that the gateway does not learn any information about the file. On the other hand, this protocol enables the client to check the correctness of the computation performed by the gateway (i.e., verify the gateway's signature).
- By involving the gateway in the key generation process, brute-force attacks on predictable messages (i.e., files) can be slowed down by rate-limiting key-generation requests to the gateway.