Securing Passive Replication Through Verification

Bruno Vavala[1,2], Nuno Neves[2], Peter Steenkiste[1]
[1]Carnegie Mellon University, U.S.
[2]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

***Abstract*—We show how to leverage trusted computing technology to design an efficient fully-passive replicated system tolerant to arbitrary failures. The system dramatically reduces the complexity of a fault-tolerant service, in terms of protocols, messages, data processing and non-deterministic operations. Our replication protocol enables the execution of a single protected service, replicating only its state, while allowing the backup replicas to check the correctness of the results. We implemented our protocol on Trusted Computing (TC) technology and compared it with two recent replication systems.**

I. Introduction

Replication is a fundamental technique to guarantee service availability and reliability. Traditionally, component replicas are managed using two main design approaches: *active replication* (AR) [13], [18] and *passive replication* (PR) [1], [3]. To the best of our knowledge, all past solutions for arbitrary fault-tolerant (known as Byzantine, [14], or BFT) replication are based on the former, commonly referred to as the State Machine Replication (SMR) paradigm.

Although BFT-SMR can achieve a high level of assurance, it is expensive and requires complex coordination protocols, and deterministic operations. Service execution must be replicated and protocols such as Consensus or Atomic Broadcast are necessary to maintain state consistency [5], [11], [22]. Moreover, a well-known difficulty is the execution of operations that may cause the state on the replicas to diverge [12]. The solution is often to assume that the service is deterministic, thereby restricting its applications.

The PR design solves the complexity and efficiency issues above, but it is less reliable than the AR design in the presence of arbitrary failures. Since PR adopts a single primary executing replica—thereby supporting non-determinism by construction—such a single point of failure inherently lacks the redundancy required to withstand failures caused by malicious intrusions. As a consequence, although backup replicas may maintain a state consistent with the primary, there is no guarantee that it is untampered. Nevertheless, many practical examples give evidence about the attractiveness of the approach in the crash-only model (e.g., [15], [16]).

In this work we introduce the concept of *Verified Passive Replication* (V-PR) that combines the security guarantees of BFT-SMR with the resource efficiency of PR (Tab. I), without trading off generality. Security is achieved using a trusted component to isolate and protect the correctness of the service execution from tampering. Efficiency is attained by verifying the service results at the backup replicas, instead of re-executing the service. V-PR is fully passive, as state updates are propagated by the primary to the backup replicas, and then applied deterministically upon validation. We demonstrate these features by implementing a V-PR-ed database engine.

We make the following contributions: (1) We design a robust passively replicated system, backed by a trusted component. (2) We define secure protocols for system initialization, fault-tolerant execution, and primary change. (3) We implement the V-PR protocol based on XMHF/TrustVisor [21], [17], running the SQLite [20] database engine as a service. In addition, we compare V-PR against state-of-art protocols. We show that its performance is close to that of BFT-SMaRt and Prime [19], [2], and it will become more and more efficient as Trusted Computing technology continues to improve (e.g., using the forthcoming Intel SGX [10] instruction set).

Features	**Replication Protocols**		
	AR	PR	V-PR
Byzantine Resistant	**yes**	no	**yes**
Replicas	**2f+1** with trust assumptions	**2f+1** [7] f+1 [1]	**2f+1** with trust assumptions
Asynchronous for safety, partially synchronous for liveness	**yes**	**yes**[7] no [1]	**yes**
(Re-)Computations	$\mathcal{O}(n)$	$\mathcal{O}(\mathbf{1})$	$\mathcal{O}(\mathbf{1})$
Data Transfer Unit	$\mathcal{O}(\texttt{cmd}+\texttt{d}_{\texttt{in}})$	$\mathcal{O}(\texttt{u})$	$\mathcal{O}(\texttt{u}+\texttt{d}_{\texttt{out}})$
Centralized Control	no	**yes**	**yes**
Non-determinism	no	**yes**	**yes**

TABLE I: Comparison among Active Replication (AR), Passive Replication (PR) and our new proposal Verified Passive Replication (V-PR). cmd is the command size, $\texttt{d}_{\texttt{in}}$ (resp. $\texttt{d}_{\texttt{out}}$) is the size of the input (resp. output) data, u is the size of the updates.

II. Overview

Architecture. V-PR is a secure passively replicated system [1], [3], [7], [6] (see Fig. 1) composed by middleware components such as: the client security middleware (Security MW, or SMW), which sets up a secure channel between the service client and the service in the primary's trusted environment; the Manager, implemented on top of the trusted computing component (TCC, Section III) — not the OS — that through the Application, Security and Replication Managers (resp. AM, SM, RM) supports the service, handling secret keys and authentication, running the replication logic and managing state updates; the U-Manager, which is an (untrusted) application executing on the OS that mediates the communication between the client, the primary's service and the backup replicas.

Operations. The service client sends a request to the replicated system (1) using the SMW to authenticate it (2). The request is delivered by the OS to the primary U-Manager (3), and then forwarded to the primary Manager (4). The request is validated by the SM and passed to the AM for the execution. The AM retrieves the reply (6) and the RM gathers the state updates (if any). Both reply and updates are protected and transferred to the U-Manager (7). Updates are broadcast to backup replicas (7a), delivered to the backup U-Manager (7b), then validated and applied deterministically (7c)[1]. After the acknowledgements from backup replicas have been received and processed by the primary (7d), the authenticated reply is forwarded to the client (8), validated by the SMW (9) and delivered to the service client application (10).

[1]State updates are handled on the TCC.

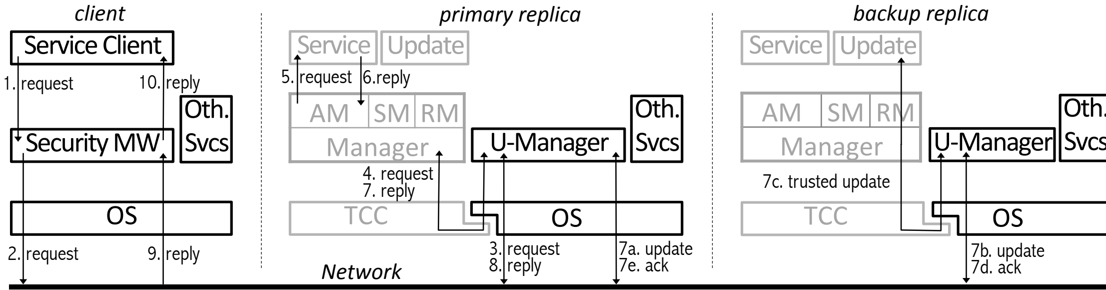

Fig. 1: Architecture of the Verified Passive Replication scheme. Light shaded parts correspond to the trusted system components.

Failure Masking vs. Resource-Efficiency. V-PR leverages Trusted Computing technology to detect any malicious behavior of the primary. Using recent advances in the area, V-PR isolates and protects the service execution on the platform by using hardware support. Service isolation allows us to: 1) reduce the attack surface, for instance by excluding the OS from the trusted computing base (TCB); 2) precisely identify the executing code, so that it can be checked remotely without re-execution. V-PR then creates a cryptographic chain that links the service results to the identity of the service and trusted hardware.

The increased resource-efficiency comes at the cost of lower failure transparency. AR [4], [5], [12], [22] in fact is able to mask malicious behavior by fully replicating the service execution through many replicas and then voting on their responses to extract the correct reply — assuming that they fail independently. PR (and so V-PR) instead does not have such redundancy. Backup replicas have to detect when the primary fails, elect a new primary, and let the client know the change. Such mechanisms can be safely implemented under periods of partial synchrony, possibly using a semi-passive approach [7].

Tolerated Failures. V-PR is robust against arbitrary failures such as: software attacks, compromised OS, message corruption, physical attacks that do not affect the TCC. However, it does not protect against programming flaws in the service. This holds even in SMR based systems unless they employ diverse service software on the replicas [8]. Moreover, it does not protect against transient or persistent hardware errors in the trusted component that disrupt the service (unless it crashes). Transient errors however could be addressed through functional hardware redundancy (e.g., the Recovery Unit in an IBM z10 maintains in a buffer the whole processor state to retry the work on error, and also instruction-processing damage checks are performed [9]); persistent errors may also affect SMR systems that do not use diverse hardware.

III. TCC Overview

We abstract the TCC through a set of primitives related to: code execution, data storage, attestation and trusted counters. This allows us to hide the complex details of the TCC (e.g., memory isolation mechanisms to protect the service execution) behind the primitives, and to focus later on the relevant aspects of V-PR. The TCC can be instantiated using the available technology (e.g., TPM, secure hypervisor) or future processor architectures such as the Intel SGX instructions set [10].

Primitives. The $\mathtt{d_{out}} \leftarrow \mathtt{execute}(\mathtt{c}, \mathtt{d_{in}})$ primitive makes the TCC execute some code $\mathtt{c}$ over some input data $\mathtt{d_{in}}$. It eventually returns a result $\mathtt{d_{out}}$. The TCC is responsible to maintain identification information about $\mathtt{c}$—typically its hash $\mathtt{h}(\mathtt{c})$—internally in a special register $\textsc{idr}$. This information can be used for secure storage, attestation and trusted counter management, whose operations are bound to $\mathtt{c}$'s identity.

The storage primitives are $\mathtt{d}^{[\textsc{idr},\mathtt{hr}]} \leftarrow \mathtt{put}(\mathtt{d}, \mathtt{hr})$ and $\{\mathtt{d}, \emptyset\} \leftarrow \mathtt{get}(\mathtt{d}^{[\mathtt{hs},\mathtt{hr}]}, \mathtt{hs}')$. The first secures some data $\mathtt{d}$ on the behalf of the currently running code $\mathtt{s}$, identified in $\textsc{idr}$ as $\mathtt{hs} \equiv \mathtt{h}(\mathtt{s})$. The original data can only be retrieved by a specific receiver code $\mathtt{r}$, with identity $\mathtt{hr} \equiv \mathtt{h}(\mathtt{r})$. The second primitive accepts some secured data and the identifier of some sender piece of code $\mathtt{hs}'$. The TCC identifies in $\textsc{idr}$ the code r' that raises the request. If $\textsc{idr} \equiv \mathtt{hr}' \equiv \mathtt{hr}$ and $\mathtt{hs}' \equiv \mathtt{hs}$ then the original data $\mathtt{d}$ is returned, otherwise it fails with $\emptyset$. Notice the following: given $\mathtt{d}^{[\mathtt{hs},\mathtt{hr}]}$, the TCC ensures the trustworthiness of $\mathtt{hs}$ on $\mathtt{put}$, and of $\mathtt{hr}$ on $\mathtt{get}$; it is up to the executing pieces of code to specify the correct recipient and sender identities in the respective operations.

The attestation primitive $\pi^{[\textsc{idr}]} \leftarrow \mathtt{attest}(\mathtt{t}, \mathtt{params})$ allows to convince a remote party of the current TCC's state. It accepts as input a timestamp (or nonce) $\mathtt{t}$ for freshness and other parameters supplied by the executing code $\mathtt{c}$, identified in $\textsc{idr}$. It produces a proof of execution (i.e., $\textsc{idr}$'s value), usually signed with the TCC-specific internal private key.

The primitives for trusted counter management are: $0 \leftarrow \mathtt{create_cnt}(\mathtt{sid})$, $\mathtt{c} \leftarrow \mathtt{get_cnt}(\mathtt{sid})$, and finally $\mathtt{c} \leftarrow \mathtt{incr_cnt}(\mathtt{sid})$. They all accept a unique service identifier $\mathtt{sid}$. The primitives return the last or incremented counter value. Specifically, the TCC creates, stores and modifies pairs of $(\mathtt{counter_ID}, \mathtt{value})$, where the identifier is dependent on the running code's identity stored in $\textsc{idr}$. In our case, we use $\mathtt{counter_ID} \leftarrow \mathtt{h}(\textsc{idr}||\mathtt{sid})$.

The primitive $\mathtt{get_cert}()$ returns the TCC own public key certificate. This can for instance be used to verify that the private key used in a digital signature belongs to an actual (and not emulated) TCC.

Finally, the TCC is equipped with a random number generator, used to generate random secret keys.

Verification. The verifier must check that (1) an actual trusted device issued the attestation and (2) the proof was generated correctly. The first is achieved by checking that the public key certificate associated to the attestation private key was issued by a trusted Certification Authority (CA), possibly owned by the hardware manufacturer. The second is achieved through $\{0, 1\} \leftarrow \mathtt{verify}(\mathtt{cert}, \pi^{[\mathtt{hs}]}, \mathtt{hs}', \mathtt{t}, \mathtt{params})$. The primitive accepts an attestation $\pi^{[\mathtt{hs}]}$, a certificate $\mathtt{cert}$ that vouches for the TCC's public key, the identity $\mathtt{hs}'$ of a piece of code that was supposedly executed, the timestamp (or nonce) $\mathtt{t}$ and parameters $\mathtt{params}$. The verification is meaningful provided that the verifier trusts the CA that signed $\mathtt{cert}$, the TCC

manufacturer; also, it succeeds if the right code ran (i.e., $\texttt{hs}' \equiv \texttt{hs} \Rightarrow \texttt{s}' \equiv \texttt{s}$) over the correct input/output `params`.

IV. V-PR: VERIFIED PASSIVE REPLICATION

This section presents the system model, outlines (due to space constraints) V-PR's initialization and failure recovery protocols, and gives details of V-PR's execution.

A. System Model

The system consists of a group of `n` nodes, each one equipped with a TCC. The V-PR Manager and the replicated service run in the protected environment provided by the TCC. The rest, including the V-PR U-Manager, the OS and other services, execute in the untrusted part of the node. An arbitrary number of clients can access the system, but they need to be enrolled beforehand to obtain the necessary authentication credentials (e.g., by contacting an identity management service).

The TCC only runs code when the `execute()` primitive is explicitly called by an untrusted software component (see Section III). The component can give some data as input and in the end receives the output value. While executing, the code is isolated with no access to the network or general storage (e.g., the disk). It can however use a limited set of secure primitives to create counters, encrypt data, or perform attestation. Moreover, any keys that are generated by the code cannot be observed, unless they are returned (unprotected) to the calling component.

At most a minority $f = \lfloor \frac{n-1}{2} \rfloor$ of the nodes can fail. In particular, the TCC can only suffer crashes but the rest of the system and network can experience Byzantine failures. Therefore, the code in the TCC either produces correct results or no values. Untrusted components (such as the U-Manager) may corrupt data, delay the execution or do any other attempt to maliciously break the protocol. Messages may be modified, removed or delayed. These assumptions are similar to other systems based on trusted components [5], [11], [22] .

The V-PR protocol ensures safety in the asynchronous model. Liveness is guaranteed in periods of partial synchrony—when messages are delivered and processed within a fixed but possibly unknown time bound. This can be achieved through retransmissions and acknowledgments.

B. The Context Data Structure

The Manager only runs when the `execute()` operation is called by the U-Manager. While inactive, it relies on the U-Manager to keep context information about the state of the protocol execution (since no general persistent storage exists in the TCC, in order to minimize the trusted computing base). The Manager expects to receive as input a context data structure (`ctx`), which is returned back to the U-Manager as one of the outputs. The structure's most relevant fields are:

- `id`: the identifier of the node
- `nreplicas`: the number of nodes
- `clientCreds`: for client authentication
- `state`: a description of the current service state, i.e., $\texttt{hash}(\texttt{state}^{j})$. It identifies the state unambiguously when associated with a trusted state counter, i.e., (`state counter`, `state`) is unique
- `K`: a system-wide shared key, created by the primary during the initialization and forwarded to the backups
- `auth`: authenticator to protect the `ctx` structure while it is stored by the U-Manager (e.g., a MAC)

The integrity of `ctx` and the confidentiality of `K` (a secret shared only between Managers in the trusted environment during the initialization phase) are critical for the system's operations. They are protected by each Manager before returning to the U-Manager: (i) an authenticator (e.g., a MAC) is computed with `K`; (ii) `K` is encrypted using the TCC `put` primitive, specifying the Manager's own identity as the intended receiver; consequently, `K` cannot be accessed from the untrusted environment by U-Managers. Later on, when the Manager is re-executed, it calls the `get` primitive to decrypt `K`, and then verifies the integrity of the received context. We will omit these steps while presenting the protocols.

The TCC maintains a *trusted state counter* and a *trusted view counter* between executions of the Manager. The former is used to assign unique and ordered values to state updates (and so to state versions); it is incremented whenever requests cause an update on the service state. The latter is used to determine a replica's role at a given time — a replica is the primary if $\texttt{viewcounter} \mod \texttt{nreplicas} \equiv \texttt{id}$, otherwise it is a backup — by counting the number of primary changes.

C. System Initialization

System initialization must ensure three main goals: (i) every correct node of the system is able to join the group, while malicious nodes are excluded; (ii) a system-wide secret key `K` is securely shared among the nodes; (iii) all nodes finish the initialization with the same state ($\texttt{state}^{1}$). Some of the challenges that have to be overcome are the lack of network access by the Managers (making them vulnerable to U-Manager misbehavior) and the absence of data protection primitives that can be utilized between TCCs (e.g., storage primitives can only be used locally and the TCC public key cannot be employed to encrypt general data as the private key is saved internally and never made available).

The first goal is achieved in two steps. First, the system administrator supplies all nodes with a vector of TCC certificates, each belonging to one specific node. Second, the Managers at the backup replicas begin to run a protocol for mutual attestation with the primary replica's Manager. All Managers are assumed to execute the same code, and thus they have the same code identity. They differentiate among themselves through the index in the certificate vector that points to the public key of the local TCC. By including in the attestation the certificate vector, the protocol guarantees that the Managers can securely authenticate each other.

The second goal is attained by setting up a secure channel between the backup and the primary Manager to exchange a secret key `K`. Each backup Manager generates a fresh public/private key pair, and includes the public part in the attestation. The primary Manager verifies each backup's attestation, and thus each backup's public key, and uses it to encrypt the shared (for all replicas) key `K`. The primary then performs an attestation, including the hash of the key, so to link the key to its identity and the TCC. The primary U-Manager then broadcasts the attestation, the encrypted key and the primary's context structure. Together with `K`, the Manager also distributes the context structure `ctx`. When each backup Manager receives the data, it verifies the primary's identity, decrypts `K` and copies the primary's context to configure the local replica.

The third goal is achieved by making the backup Managers

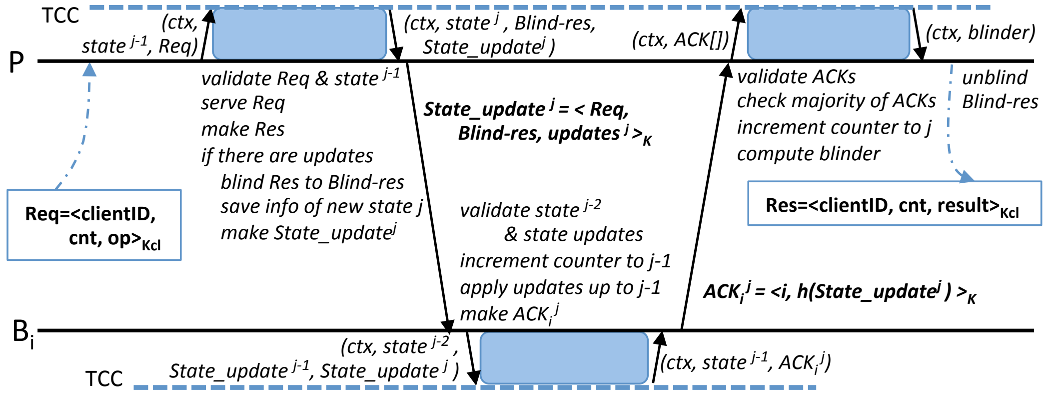

Fig. 2: Procedure to serve a client request requiring state updates.

send an acknowledgment to the primary, authenticated with K. The primary Manager waits for the arrival of enough ACKs in order to install the initial service state ($\texttt{state}^1$). This state is then propagated to the backup Managers through the initial state updates, as described in the next section.

D. Normal execution

V-PR's processing guarantees that: (i) client requests are correctly served; (ii) state updates are properly installed; (iii) recoverability and consistency are ensured in the case of failures. Read requests are only handled by the primary since they do not update the state. For write requests, the procedure requires a simple majority of replicas to be available, and uses all their TCCs for three sequential Manager executions (two at the primary Manager for processing a request and the respective acknowledgments, and one at the backup Manager to process the state updates).

The description below assumes that client and servers share a key Kcl. Depending on the client authentication solution, this key may be derived on-the-fly by combining the identifier clientID and the credentials supplied by the system administrator (clientCred), or through a key distribution protocol.

The procedure is displayed in Fig. 2. Whenever the client calls a service operation, it creates a request message (Req) with the clientID, a session counter (cnt) and an operation (op) including the associated parameters. The message is authenticated and integrity protected with Kcl (e.g., by adding a MAC), and transmitted to the primary.

When the message is delivered, the primary U-Manager executes the Manager with the last context (ctx), the current state version ($\texttt{state}^{j-1}$) and the request. The Manager validates the request and the state, and runs the operation. Next, it produces a response message (Res) for the client using the value(s) returned by the service. The response to a read request is simply authenticated and sent back to the client. The response to a write request instead raises the challenge of propagating the updates to the backups before the client receives a response. In fact, consistency issues may arise if the primary fails or the primary U-Manager is compromised.

Our solution is to blind the response (Blind-res)[2], thereby ensuring that the client cannot accept it. Moreover, the context structure needs to reflect the changes, namely the state field becomes the hash of the new state. Additionally, a $\texttt{State_update}^{j}$ message is created for the backups, which includes the client request, the blinded reply and the update information ($\texttt{updates}^{j}$) to bring the state from version $\texttt{j}-1$ to j. Request and reply are useful in case of failure to match the retransmitted client request to its associated response, without re-doing the operation in $\texttt{state}^{j}$. If these are not available, the new primary proceeds from $\texttt{state}^{j-1}$.

Each backup Manager needs to validate the messages (for state update $\texttt{j}-1$ and j) to check for corruptions and to ensure that they are applied in the correct order[3]. In particular, for these updates, the trusted state counter needs to be equal to $\texttt{j}-2$, indicating the last state update that was performed. This counter has the benefit of allowing backup replicas to coordinate without explicit message exchanges. In fact, when the state update j arrives, the replicas have no knowledge of which replicas have also received it. However, since the primary Manager is trustworthy, the arrival of j-th update actually implies that state $\texttt{j}-1$ was accepted by a majority of (primary and backup) replicas. Hence, the update $\texttt{j}-1$ may be applied and the trusted counter can be incremented up to $\texttt{j}-1$. Each backup Manager i uses then the acknowledgement $\texttt{ACK}_{\texttt{i}}^{\texttt{j}}$ to inform the primary that state update j has been validated and is locally available, but not yet applied.

The primary U-Manager waits for the arrival of a majority of these acknowledgements. Then, it executes the Manager, who checks for a majority of valid acknowledgements — this ensures that at least one correct replica has the latest state. The Manager increments the state counter (to j), thereby installing the last state, and outputs the blinder. The U-Manager then unblinds the response and forwards it to the client.

E. Fault Handling

If the primary node fails, the backup replicas have to select a new primary in order to make progress. The recovery procedure is timeout-driven and is triggered by the U-Managers, as the Managers do not execute continuously and have no access to network resources (§IV-A), so they cannot perform system monitoring. Recall from the model that, although untrusted, a majority of the U-Managers is assumed to behave correctly. Therefore, each one begins the recovery procedure when it

[2] A blinder value is XORed with the message authenticator to prevent its validation. It is computed using a secure pseudo-random number generator seeded with the secret K, the state counter and the current state hash.

[3] Notice that for the first state update j is equal to 2; $\texttt{State}^{0}$ is the uninitialized service state, and $\texttt{State_update}^{1}$ is actually represented by the initial state update from the initialization procedure.

stops receiving valid state updates for a period[4].

The selection of a new primary is guided by two principles: *Unique Majority* and *Progress Evidence*. First, a primary is effectively changed when a majority of replicas increment their trusted view counter to the same value. This guarantees the uniqueness of a primary that is able to make progress, since a majority of the backups recognize its authority to issue state updates. This also forces the other nodes to move to the newer view because client requests, or updates, cannot be successfully processed in the older view.

Second, a replica can safely apply all state updates up to state $\mathtt{j}-1$ when state update `j` is received, independently of the view number. In fact, for the primary Manager to issue the state update `j`, it must have received the acknowledgements for state $\mathtt{j}-1$. Consequently, replicas may change view, but they can still make progress with any two consecutive updates.

Protocol. When the timeout at a backup replica expires, the U-Manager executes the local Manager to obtain a `Probe_change` message. The message contains an authenticated description of the current configuration (including the state and view counter values) and it is multicast to all nodes.

Replicas wait for a majority of `Probe_change` messages that match the *same* configuration of their local Manager before moving to the next phase. In the meanwhile, they continue to participate in the system as usual, in case more recent state updates are delivered.

The Manager is called again when enough `Probe_change` messages are delivered. It validates the messages, and then returns a `Probe_reply` also containing the current configuration. A set with a majority of valid `Probe_reply` messages that match the same backup Manager's configuration triggers the view change. The Manager is called to increment the trusted view counter, and a final `New_primary` message is produced and multicast to inform about the new view.

A corner case. Let us assume that the old primary crashes while broadcasting the $\mathtt{j}+1$-th state update (`u1`). If the new primary receives this message, then it re-multicasts the same update to be processed by backup replicas. However, if it does not receive it, then the client times out and re-issues the request. The new primary can therefore make progress by computing a new $\mathtt{j}+1$-th state update (say $\mathtt{u2} \neq \mathtt{u1}$). Since the backup replicas have not yet increased their trusted state counters to $\mathtt{j}+1$, they can safely favor the last `u2` issued by the new primary Manager and drop `u1`. In fact, this update will only become definitive when the update $\mathtt{j}+2$ is broadcast. Computation then proceeds as usual.

V. Experimental Evaluation

We evaluate V-PR in a cluster of servers and also compare its performance with two open-source BFT-SMR libraries, namely BFT-SMaRt [19] and Prime [2].

A. Implementation and Experimental Setup

We instantiated our TCC using TrustVisor [17], a security hypervisor implemented in the XMHF framework [21]. XMHF/TrustVisor offers fast trusted services, and its security is cryptographically bound to a hardware TPM.

We chose to replicate the well-known and widely deployed SQLite [20] database engine. Our SQLite version is self-contained to run in the isolated environment (i.e., no OS/library support) provided by XMHF/TrustVisor. Moreover, we implemented a module for fast in-memory database operations, that enables fast state update interception whenever SQLite modifies the database.

Our testbed is a set of Dell PowerEdge R420 servers, equipped with Intel Xeon E5-2407 CPUs, 3GB of memory and a TPM v.1.2. These machines run Ubuntu 12.04 with a kernel version 3.2.0-27, and they are connected with a 1 GB/s Dell PowerConnect 5448 switch. In order to tolerate one fault, we used 3 machines for V-PR, and 4 for BFT-SMaRt and Prime.

B. Analysis

We evaluate the application-level CPU savings in V-PR, and compare its performance with state-of-the-art tools. We also study V-PR's end-to-end latency in a realistic scenario.

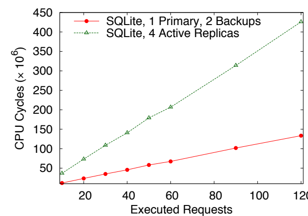

Fig. 3: Average system-wide application-level CPU cycle consumption for passively and actively replicated SQLite deployments.

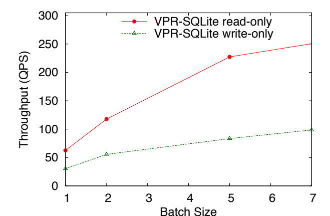

Fig. 4: End-to-end performance of a VPR-ed SQLite implementation.

CPU savings. Fig. 3 compares the application-level CPU consumption (in cycles) between a passive replication and an active replication of SQLite. The amount of savings is proportional to the processing effort performed by the replicated service. In the experiments, the write workload is created with simple delete queries that produce state updates, in order to use all three V-PR replicas — recall that read-only queries are executed just at the primary. As we grow the number of requests, the gap between the two curves increases. More expensive queries, like grouping and sorting operations, would make the gap wider, thus being more favorable to V-PR.

End-to-end measurements. Fig. 4 presents the performance of our V-PR-ed SQLite. We executed read-only requests (such as `select` operations) and write-only requests (such as `delete` operations) over a table of 200 items. Reads are at least twice as fast as writes due to the additional TCC call to process the acknowledgements from backup replicas. Batching enables to amortize the TCC latency per-request by more than three times for both types of operations.

Basic performance comparison. We analyze inherent features of V-PR, BFT-SMaRt and Prime at run-time (Tab. II). The exchanged messages measure the coordination overhead to maintain state consistency (as shown in [4]). Each message exchange phase counts as one hop. V-PR outperforms BFT-SMaRt and Prime because it avoids expensive coordination protocols for request ordering, such as Consensus.

In V-PR, the primary avoids any interaction with the backups for read requests — client request and reply (2 hops) are the only messages. V-PR recognizes *a-posteriori* which

[4]The U-Manager, however, should adjust the timeouts to reflect the current situation where the primary is taking longer to transmit messages (e.g., due either to processing delays or network congestion). To prevent the case where a malicious primary U-Manager tries to delay the whole system, the timeouts are only increased up to a certain value defined by the system administrator.

	BFT-SMaRt	Prime	V-PR
Messages (r)	1+1		1+1
Messages (w)	4+3+16+16+4	3+16+16+3+12+16	1+3+3+1
Hops	(r) 2 (w) 5	6	(r) 2 (w) 4
Replicas (f=1)	4	4	3
Executions	4 (active)	4 (active)	1 (active) 2 (passive)

TABLE II: Normal request execution in BFT-SMaRt, Prime and V-PR.

requests contain read or write operations by tracking the changes to the database. BFT-SMaRt has a similar optimized execution for read-only operations—namely through the `invokeUnordered` primitive—which avoids the call to the atomic multicast protocol. However, such capability has to be explicitly programmed (*a-priori*) into the client application by calling a specific read-only operation. For Prime, although client operations can be classified as read-only or read/write, no such optimization is mentioned in [2], and we noticed no significant difference with respect to write-only requests.

Write requests must be ordered in all systems. In V-PR this process is centralized at the primary: client requests are ordered through a session counter, while state updates are ordered through the state counter. Consequently, the primary's update message and the backup acknowledgements are the only transmitted messages—2 hops more than read requests. In BFT-SMaRt and Prime, instead, the ordering process is a protocol to cope with potentially malicious replicas.

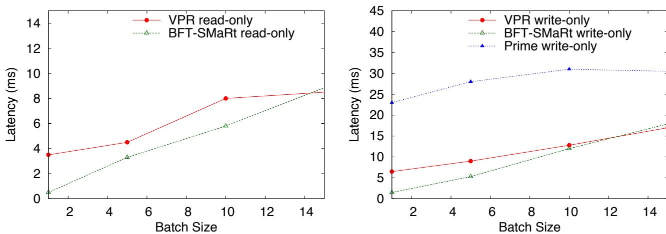

Fig. 5: End-to-end latency, measured at the client, of a replicated zero-overhead service.

Fig. 5 shows that V-PR is slower than BFT-SMaRt for single requests. This is primarily due to the TCC. Prime is mainly delayed by the heavy use of signatures. However, all systems take advantage of request batching to improve their efficiency. Noticeably, V-PR matches BFT-SMaRt latency when the batch size reaches around 12 requests. This experiment is based on a simple zero-overhead service in fault-free runs.

Future trusted computing technology is expected to reduce TCC costs, and can be easily integrated below V-PR. In fact, V-PR is not bound to XMHF-TrustVisor, since the TCC primitives (§III) provide a suitable abstraction to separate V-PR from the TCC. Another TCC could be based for instance on Intel SGX [10], which provides on-CPU trusted computing services based on CPU instructions rather than hypercalls.

VI. Conclusions

We presented the V-PR scheme, which is a fully passive replication protocol based on the concept of computation verification. V-PR shows how to leverage secure hardware and trust assumptions to deal with arbitrary failures. V-PR only requires one executing replica, and thus enables resource saving and non-deterministic executions.

Acknowledgments

This work was partially supported by the EC through project FP7-607109 (SEGRID), by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/CEC/00408/2013 (LaSIGE), and by the research grant SFRH/BD/51562/2011. Thanks to Vinicius Cogo for providing support in setting up the testbed, and to André Nogueira for interesting discussions on implementation details.

References

[1] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *In Proc. of the 2nd International Conference on Software Engineering (ICSE)*, page 562, 1976.

[2] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine Replication under Attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, July 2011.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press, May 1993.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[5] M. Correia, N. F. Neves, and P. Verissimo. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, Apr. 2008.

[7] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, 1998.

[8] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, June 2014.

[9] IBM. System z10. *http://www.redbooks.ibm.com/redbooks/pdfs//sg247516.pdf*.

[10] Intel. Software Guard Extensions. *https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf*.

[11] 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 European Conference on Computer Systems (EuroSys)*, page 295, 2012.

[12] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 237–250, Oct. 2012.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[15] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation (OSDI)*, page 8, Dec. 2004.

[16] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson. Niobe: a Practical Replication Protocol. *Journal ACM Transactions on Storage (TOS)*, 3(4):1–43, Feb. 2008.

[17] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 143–158, 2010.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[19] J. Sousa, E. Alchieri, and A. N. Bessani. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the IEEE Conference on Dependable Systems & Networks (DSN)*, pages 355–362, 2014.

[20] SQLite. http://www.sqlite.org/.

[21] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 430–444, May 2013.

[22] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient Byzantine Fault-Tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan. 2013.