Constructing a Practical Intrusion Tolerant Replication System

Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, Yair Amir
Department of Computer Science at Johns Hopkins University
{platania, dano, tantillo, rsharm22, yairamir}@cs.jhu.edu

Technical Report CNDS-2015-1 - January 2015
http://www.dsn.jhu.edu

***Abstract*—The increasing number of cyber attacks against critical infrastructures, which typically require large state and long system lifetimes, necessitates the design of systems that are able to work correctly even if part of them is compromised. We present the first practical survivable intrusion tolerant replication system, which defends across *space* and *time* using compiler-based diversity and proactive recovery, respectively. Our system supports large-state applications, and utilizes the Prime BFT protocol (providing performance guarantees under attack) with a compiler-based diversification engine. We devise a novel theoretical model that computes how resilient the system is over its lifetime based on the rejuvenation rate and the number of replicas. This model shows that we can achieve correctness and availability over 30 years with a confidence of 95% even when we transfer a state of 1 terabyte after each rejuvenation. Finally, we describe the architecture of a survivable SCADA system that uses intrusion tolerant replication to protect the master server.**

I. Introduction

Critical infrastructures such as financial, transport, or SCADA systems play an important role in everyday life. In this world, availability, reliability, and security are paramount. However, it is well known that exploits exist in software architectures and that attackers use these exploits to compromise systems. As a consequence, critical systems must be built to tolerate intrusions: they need to support consistent, large state across the infrastructure, employing algorithms that guarantee correct distributed operation over long system lifetimes even in the face of intrusions, where part of the infrastructure is controlled by the adversary.

To this end, Byzantine Fault Tolerant (BFT) protocols (e.g. [1]) can be considered a building block for the design of intrusion tolerant systems, because they are able to work correctly even if f out of $3f+1$ replicas behave in an arbitrary manner. However, BFT protocols alone do not provide a solid basis for the construction of intrusion tolerant replication systems with long lifetime (e.g. years). A smart attacker can spend the time to compromise $f+1$ replicas in order to cause inconsistencies. This is particularly true if all the replicas are identical, as is common in real-world deployments, because an identical attack surface allows an attacker to successfully compromise all the replicas in the system with the same attack. As such, it is important to diversify replicas as much as possible [2]–[4]. We refer to this approach as *defending the system across space*.

However, defense across space only increases the attacker's workload linearly by requiring the attacker to develop $f+1$ distinct attacks. The attacker can still spend the time to develop these attacks, especially if the system is long-lived. Hence, periodic rejuvenation of replicas is necessary to clean potentially undetected intrusions. The rejuvenated replica should restart as non-compromised to ensure any intrusion is cleaned, and should be diverse from all currently and previously existing replicas to ensure that the attacker has no advantage of prior knowledge. This forces the attacker to act quickly, otherwise any progress he or she has made in compromising system replicas will be undone. We refer to this approach as *defending the system across time*. Some of the current approaches in the field of intrusion tolerant systems [1], [5], [6] offer weak solutions to defend systems across space and time: they either do not diversify replicas after rejuvenation, or they use coarse-grained diversity (e.g. at the operating system level) where the strength of such a system is limited by the small number of different copies available.

Practical deployments of proactive rejuvenation raise the issue of how often to refresh replicas in order to maintain system correctness over its lifetime. Rejuvenating too often can reduce system availability, limit the size of the replicated state, and incur unnecessary overhead. In contrast, rejuvenating too infrequently can increase the likelihood that the attacker will succeed.

In this paper, we present the first practical survivable intrusion tolerant replication system that provides defense across space and time, a term first used in [7]. Our main contributions are:

• The first proactive recovery protocol that supports large state. We devise two novel state transfer strategies: one that prioritizes fast data retrieval and one that minimizes bandwidth usage, which can be used to restart a replica from a clean state. We provide a theoretical evaluation that shows which strategy works better based on the size of the state to transfer and the specific application requirements;

• A theoretical model that computes the resiliency of the system over its lifetime (e.g. 30 years) based on the rejuvenation rate, the number of replicas, and the strength of a single replica. We compare two different settings: one focuses on correctness, and the other focuses on correctness and continuous availability, at the cost of additional replicas in the system. Critical applications should consider paying this cost to continue providing availability during rejuvenations;

• The first integration of subsystems that support the assumptions of a practical survivable data replication system: the Prime BFT protocol [8], which ensures performance guarantees even while under attack; and the MultiCompiler [9] that produces different versions of the system, such that no two versions present an identical attack surface. This does not mean that the exploits have vanished, but rather that the attacker must craft a new attack for each replica. A new version of a replica is produced after its rejuvenation;

• The architecture of a survivable SCADA system that uses our intrusion tolerant solution to replicate the master server in order to make it survivable to attacks.

The work in [10] integrates Prime into the Siemens corporation commercial SCADA product for the power grid to create an intrusion tolerant system. However, that work does not provide defense across space and time. An adversary can still compromise a single replica and take down the entire system by launching the same attack on many replicas. Our work solves this problem by periodically rejuvenating Prime replicas and generating a new variant of a replica after its rejuvenation.

It is worthwhile to note that proactive recovery and diversity do not protect against protocol flaws. If the protocol has a flaw that makes it vulnerable to attacks (e.g. SQL injection), then proactive recovery and diversity have no effect. In addition, proactive recovery and diversity do not protect against DDoS attacks. Handling resource exhaustion attacks is an orthogonal and complementary problem to the one described in this paper.

The remainder of the paper is organized as follows: Section II introduces Prime and software diversity. Section III presents the system model. Section IV describes the proactive recovery algorithm. Section V presents a theoretical evaluation of different state transfer strategies and the experimental evaluation of one of these. Section VI illustrates the theoretical rejuvenation model. Section VII describes the architecture of a SCADA system that is survivable over a long period of time. Section VIII discusses previous works, and Section IX concludes the paper.

II. Background

A. Prime, a BFT protocol with performance guarantees even while under attack

In this paper we focus on Prime as the baseline to build a long-lived intrusion tolerant system. Unlike previous BFT protocols, Prime bounds the amount of performance degradation that can be caused by a malicious leader. To do so, Prime extends the typical *pre-prepare, prepare,* and *commit* phases for global ordering with a *pre-ordering* phase, where a correct replica that receives a client operation broadcasts that operation in the system together with a locally generated sequence number (see Figure 1).

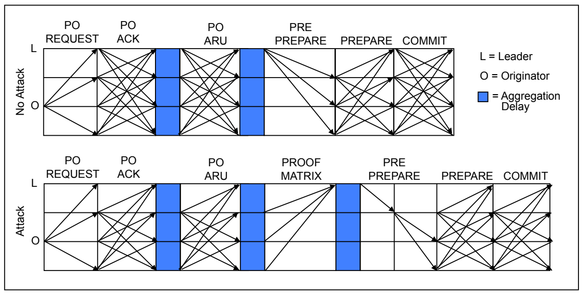


Fig. 1. The pre-ordering phase in Prime helps to detect a delay attack and replace a malicious leader.

If the operations injected by the leader during the *pre-prepare* phase do not match those in the *pre-ordering* phase, the leader is suspected by correct replicas and replaced. In addition, Prime replicas run a background protocol to estimate the round-trip time to each other in order to compute how fast the leader should inject client operations for global ordering. If the leader fails to respect timeliness constraints, it is suspected by correct replicas and replaced. With respect to other BFT protocols, the *pre-ordering* phase and the round-trip time estimation in Prime introduce a small performance hit during normal operations. However, these are necessary to monitor the behavior of the leader and replace it quickly to guarantee that even under attack the client operations introduced by correct replicas are executed within a *bounded-delay* that depends on the current network conditions (see Section III). This allows Prime to perform an order of magnitude better than previous BFT protocols in the presence of an attack.

B. Software diversity

Software diversity, such as N-version programming [11], [12], was originally introduced for software reliability. More recently, cheaper software diversity solutions [2], [3] (i.e. not involving humans in the loop) have been used to defend software systems from *code reuse attacks*, in which an attacker exploits knowledge of the code by using snippets or entire functions from the application itself to perform an attack. The goal of software diversity is to *evolve* programs into different but semantically identical versions, such that it is unlikely that the same attack will succeed on any two variants [13].

In this paper we use the compiler presented in [9] to diversify Prime replicas. This MultiCompiler is based on the LLVM 3.1 compiler and makes use of no-operation insertion, stack padding, shuffling the stack frames, substituting instructions for equivalent instructions, randomizing the register allocation, and randomizing instruction scheduling to obfuscate the code layout of an application. After each rejuvenation the MultiCompiler takes the Prime bitcode, i.e. a compiler-generated intermediate representation of the source code, and a 64-bit random seed and generates a diverse copy of a Prime replica from a large entropy space. Hence, if an adversary attacks all replicas in parallel, the probability to defeat more than f replicas is low. Diversity obtainable with the MultiCompiler complements diversity obtainable at the operating system level, e.g. using different distributions or different versions of the same distribution.

III. System model and properties

The system is composed of n replicas, which run the Prime BFT protocol and communicate by exchanging messages. Replicas may suffer from Byzantine or benign faults. We characterize replicas based on their behavior:

- Correct replica: a replica that follows the algorithm, is consistent, and is not being partitioned or rejuvenated.
- Malicious (or compromised) replica: a replica that exhibits arbitrary (i.e. Byzantine) behavior not according to the algorithm.
- Crashed replica: a replica that stops working. A crashed replica can restart the application from the state on the disk.
- Rejuvenating replica: a replica that is experiencing a benign fault. In particular, even if the replica was malicious, during rejuvenation it switches to a crashed replica.
- Partitioned replica: replica that cannot communicate with at least γ correct replicas (see Sections III-B and III-C).

Replicas are periodically rejuvenated to clean any intrusions. A fundamental condition for success is that a correct replica completes recovery before the rejuvenation of the next replica. We define the time between two consecutive rejuvenations of any replicas as the *inter-rejuvenation period*. A *rejuvenation cycle* is n times the inter-rejuvenation period. In addition, we require that client operations, also referred to as updates, are not injected into the system faster than correct replicas can execute them. In this way, we ensure that a correct replica that rejuvenates eventually catches up.

All messages sent among replicas are digitally signed. We assume that digital signatures are unforgeable without knowing a replica's private key. We also make use of a collision-resistant hash function for computing message digests. In addition, we require that each replica is equipped with tamper-proof cryptographic material to generate and store the replica's private key and sign messages without revealing that key. To this end, we use the Trusted Platform Module (TPM). The TPM also comes with a random number generator and a monotonically increasing counter. Many modern computers are sold with a TPM module built-in.

In the following, we first specify the attack model, and then we describe the system model in the presence of $3f+1$ replicas, as in typical BFT protocols, and in the presence of $3f+2k+1$ replicas [5], where k is the maximum number of crashed, rejuvenating, and partitioned replicas tolerated in the presence of f malicious replicas. In the presence of $3f+2k+1$ replicas the algorithm is guaranteed to make progress despite malicious/benign failures and the rejuvenation of correct replicas.

A. Attack model

We allow for a very powerful adversary that can exploit vulnerabilities of the operating system and the application to compromise and control a replica. The adversary can delay the sending and receipt of messages at malicious replicas, but it cannot affect communication among correct replicas. In addition, the adversary can leak the private key of a malicious replica and disseminate that key to other malicious replicas in order to send forged messages. The adversary can also compromise the state of a malicious replica. However, we assume that the adversary has no physical access to the system and is computationally bounded, such that it cannot subvert the cryptographic mechanisms described above.

B. $n = 3f+1$ *replicas*

We consider f to be the maximum number of replicas that can concurrently experience malicious and/or benign faults (including recovering, crashed, and partitioned replicas) within a *vulnerability window*, i.e. the maximum time T between when a replica fails and when it recovers from that fault [1]. We also assume that the network may experience temporary partitions. If the number of faulty plus partitioned replicas exceeds f, then the system halts until the partitioned replicas reconnect and catch up. In the presence of $3f+1$ replicas we define a partitioned replica as a replica that cannot communicate with another $\gamma = 2f$ correct replicas. In the following, we discuss how the Prime properties presented in [8] change with proactive recovery.

Property 1: Safety: If two correct replicas execute the i^{th} update, then these updates are identical.
To guarantee Safety in presence of proactive recovery, a correct replica has to implement a persistent memory that survives across rejuvenations, whose content has to be validated after each rejuvenation. We require that each correct Prime replica stores on the disk all messages that it sends to or accepts from other replicas. This prevents a correct replica from sending the same message twice or accepting two different messages with the same sequence number. After rejuvenation, the replica can reload the messages into main memory.

Property 2: Network-Stability: There is a time after which the following condition holds for a set S of at least $2f+1$ correct replicas (stable replicas): for each pair of replicas r and s, there exists a value $Min\_Lat(r,s)$, unknown to the replicas, such that if r sends a message to s, it will arrive with a delay $\Delta_{r,s}$, where $Min\_Lat(r,s) \leq \Delta_{r,s} \leq Min\_Lat(r,s) \cdot K_{Lat}$, with K_{Lat} a known network-specific constant accounting for latency variability.
In those executions in which Network-Stability is met, Prime guarantees the following Liveness property.

Property 3: Liveness: If a stable replica initiates an update, all stable replicas eventually execute the update.
This property guarantees that if the network is sufficiently stable each update is eventually executed by all correct replicas. To meet Liveness we must guarantee that recovery eventually completes and partitions eventually heal. Because we require that updates are not injected into the system faster than correct replicas can execute them, we guarantee that eventually a recovering replica will catch up. The Liveness

property does not specify how fast the updates need to be executed. When NETWORK-STABILITY is met, Prime provides a stronger performance guarantee:

Property 4: BOUNDED-DELAY: There exists a time after which the latency for any update initiated by a stable replica is upper bounded.
In the presence of proactive recovery even a correct and fast leader will be eventually rejuvenated. In this case we can still guarantee BOUNDED-DELAY, but for at most $3f$ occurrences the worst-case bound on BOUNDED-DELAY becomes $t = 2f \cdot \alpha + \beta$, with α the time to detect and replace a malicious or slow leader, and β the time for a correct replica to complete recovery. We calculate t will be approximately hours for real systems with large state, with t dominated by β. Under normal operations (i.e. no recovery in progress), if in the presence of f failures an additional replica partitions away for a while and then rejoins, we are not able to guarantee BOUNDED-DELAY until the partitioned replica catches up. In this case we are only able to guarantee LIVENESS.

Moreover, the selection of the Prime leader imposes a tradeoff between availability and performance that depends on the specific application. An application can decide to spend more time in order to find the Prime leader that ensures the fastest progress, or it can wait less and elect the first leader that meets the specific timeliness requirements.

C. $n = 3f + 2k + 1$ *replicas*

In this section we investigate how to reduce t, the worst-case bound on BOUNDED-DELAY after the rejuvenation of the leader. We avoid waiting for the complete recovery after the rejuvenation of a correct replica by adding other replicas in the system. Specifically, we need $3f + 2k + 1$ replicas, as previously described in [5]. k is the maximum number of crashed, recovering, and partitioned replicas tolerated in the presence of f malicious replicas during a *vulnerability window* T. In the presence of $3f+2k+1$ replicas we define a partitioned replica as a replica that cannot communicate with another $\gamma = 2f + k$ correct replicas. Augmenting the number of replicas requires at least $2f+k+1$ replicas to order updates. Hence, certificates collected during pre-ordering and ordering phases are composed of $2f + k + 1$ messages.

The definitions of SAFETY and LIVENESS do not change in the presence of $3f + 2k + 1$ replicas. The only difference in NETWORK-STABILITY is that at any time the set S of stable replicas can be populated by any $2f + k + 1$ correct replicas, i.e. the minimum number required to order updates and elect a new leader. Hence, in a system with $3f + 2k + 1$ replicas, if NETWORK-STABILITY holds, we can still guarantee BOUNDED-DELAY over the rejuvenation cycle, but for at most $3f+2k$ occurrences the worst-case bound on BOUNDED-DELAY becomes $t = (2f + k) \cdot \alpha$. We calculate t will be subsecond for real systems with large state.

Compared to the case of $3f+1$, in the presence of $3f+2k+1$ replicas we have a higher number of occurrences in which the worst-case bound on BOUNDED-DELAY is t. However, the rejuvenation cycle is longer, and we also expect that the time $(2f+k)\cdot\alpha$ to settle on a correct and fast leader is much smaller than the time β to complete recovery after the rejuvenation of a correct replica.

In addition, under normal operations (i.e. no recovery in progress), in the presence of f faults we can tolerate the temporary partition of at most k replicas, while still guaranteeing BOUNDED-DELAY. Unlike the $3f + 1$ replicas case, where the protocol can ensure only LIVENESS, the system does not need to wait for the partitioned replicas to catch up to also guarantee BOUNDED-DELAY.

IV. PROACTIVE RECOVERY ALGORITHM

The proactive recovery algorithm depends on a component trusted to periodically initiate proactive recovery in a round robin manner. We describe in Section V how this component can be implemented. After each rejuvenation we diversify replicas as described before. The main operations the protocol executes are:
1) Replica rejuvenation: the server that hosts a replica is periodically rebooted;
2) Key replacement: the private/public keys of the rejuvenated replica are refreshed with the help of the TPM, invalidating all previous keys that could be compromised;
3) State validation: we assume the presence of a database to maintain replicated state. It needs to be validated before applying new updates;
4) State transfer: if the state is compromised, a clean copy of the state must be transferred from the other correct replicas;
5) Prime certificate validation: Prime messages are persistently stored on the disk to ensure SAFETY across rejuvenations. They are reloaded into main memory and validated before accepting or sending new Prime messages; and
6) Prime certificate transfer: if some certificate is compromised or missing, it must be retrieved from the other correct replicas.

Next, we describe each operation of the proactive recovery protocol for a system with $3f + 1$ replicas. The modifications for a system with $3f+2k+1$ replicas are described in Section IV-F.

A. *Replica rejuvenation*

Periodically, the trusted component that runs a proactive recovery scheduler selects one replica to rejuvenate. Each replica is equipped with a physical read-only medium (e.g. CD-ROM) that stores a clean copy of the operating system, the Prime bitcode, the MultiCompiler, and the public keys of the TPMs of the other servers. Note that periodically a system administrator can replace the operating system with the latest version, including security patches. Hence, the rejuvenated replica restarts the application from a correct version of the operating system and recompiles the Prime bitcode with the MultiCompiler, which takes as input a random seed generated by the TPM. The replica is then ready to restart Prime and execute the next steps of the proactive recovery protocol. During recovery the replica does not execute pre-ordering and ordering operations.

B. Key replacement

Each Prime replica has two private/public key pairs: one pair is generated by the TPM at deployment time, the other one is generated after each rejuvenation and used to sign/verify Prime messages. The private key generated by the TPM is stored in the TPM itself and cannot be leaked or deleted unless the adversary has physical access to the system and clears the TPM internal registers (we assume the adversary has no physical access to the system). The public key is disseminated to other replicas, so they can verify what this TPM signs.

Private/public keys to sign/verify Prime messages, also referred to as session keys, are refreshed after rejuvenation. Indeed, a replica can be impersonated if the attacker leaks the private session key and sends it to other malicious replicas. Session key replacement is the first operation to execute. If the replica was malicious before rejuvenation, after invalidating old keys we consider that replica to be experiencing a benign fault. Specifically, the rejuvenated replica generates a new private/public session key pair; then it uses the TPM to sign a message containing the new public key. The TPM also generates and attaches a monotonically increasing sequence number to that message before signing it to avoid replay attacks. The message is then sent to all other replicas in the system.

A correct replica accepts (and appends on a file) a new key if and only if the sequence number generated by the sending TPM is higher than the sequence numbers associated with the old public session keys of the same replica. From this moment until the next rejuvenation, the recovering replica will sign all messages with the new private session key. When at least $f+1$ correct replicas (at least $2f+1$ replicas in total) accept the new key, all the previous keys of the rejuvenated replica are invalid. At this point, updates injected by an impersonator and signed with the old private key cannot be ordered by Prime. To ensure that a new key reaches all correct replicas, we use a forwarding mechanism: the first time a correct replica receives a new key it propagates the message to all other replicas. Note that a correct replica accepts a message signed with the TPM if and only if it contains a new public session key. Other messages signed with the TPM that could be injected by malicious replicas are dropped.

Since old public keys may be required to decrypt older pre-ordering and ordering messages during certificates validation (see Section IV-E), the file with session keys must be validated after rejuvenation. Because we expect this file to be small in size (e.g. 70 MB in a system with 10 replicas, a rejuvenation rate of one replica per day, and a system lifetime of 30 years), this operation is straightforward: the rejuvenated replica computes a digest of the file, and broadcasts a message to request the digest of the same file stored by other replicas. The rejuvenated replica waits for $f+1$ replies with the same digest: if this value matches the one computed locally, then the file is correct. Otherwise, the replica requests the file from those $f+1$ replicas one by one until it receives a file that matches the digest.

C. State validation

We assume that each Prime replica applies updates to a database. The content of the database represents the replicated state, which must be validated after rejuvenation. Correct replicas take a checkpoint of the state every x updates. This operation can be executed in several ways: modern databases allow users to take periodical snapshots using copy-on-write techniques, or by dumping the database content to a text file. In addition, third party tools also allow users to execute checkpointing operations efficiently. In this work we do not specify the technique used to take a database snapshot, we only consider the output of a checkpointing operation, i.e. a blob of data. Correct replicas maintain ck checkpoints on the disk, with ck a parameter of the algorithm. Each checkpoint is logged in a file, which contains, for each checkpoint, the sequence number of the last executed operation and a digest of the state.

After rejuvenation and key replacement, the fresh replica reads the state at the most recent checkpoint, e.g. ck_i, and computes the digest. Then, a request for the digest of ck_i, together with the sequence number of the last operation executed before ck_i, is sent to all other replicas. A correct replica that receives such a request replies back with the digest of that checkpoint if it has that digest, otherwise it replies with a $null$ value (the requested checkpoint may be too old or may refer to an invalid or non-existant checkpoint if the rejuvenated replica was compromised). The rejuvenated replica waits for $f+1$ replies with the same sequence number and digests, or $f+1$ $null$ replies. In the former case, the replica compares the received digests with the one computed locally: if they match, the state at checkpoint ck_i is correct, otherwise state transfer is necessary. In the case of $f+1$ $null$ replies, instead, the rejuvenated replica selects from the log file an older checkpoint, e.g. ck_{i-1}, and repeats the state validation process.

If the rejuvenated replica has no valid checkpoints, it broadcasts a request for the most recent checkpoint taken by other replicas. The replica waits for $2f+1$ replies and selects the checkpoint ck^* with the $f+1^{th}$ highest sequence number.

D. State transfer

In a system with a potentially large state, the state transfer mechanism must be efficient to guarantee that the recovery of a compromised replica completes as quickly as possible to allow the system to rejuvenate more often, to support the assumption that the adversary does not have enough time to compromise more than one third of the entire system. In the following, we propose two different strategies, one that prioritizes fast recovery at the cost of bandwidth overhead, and one that minimizes bandwidth usage. A system administrator can choose the strategy that best satisfies the application requirements. We present a theoretical analysis of these strategies in Section V-A. We logically partition the state into several data blocks of fixed size. The rejuvenated replica recovers correct state by transferring these data blocks.

1) State transfer reducing latency: The recovering replica requests a data block b_i of $f+1$ replicas, while another f replicas are selected to send just a digest of that block. The recovering replica then computes a digest for each received copy of the data block until a correct copy is found, that is, one whose digest matches at least another f digests. After that, the recovering replica moves on to recover data block b_{i+1}. This approach reduces the time to complete state transfer, because each block is collected in a single round (we are guaranteed that at least one copy out of $f+1$ is correct), at the cost of bandwidth overhead because each block is sent $f+1$ times.

2) State transfer reducing bandwidth usage: This strategy reduces the impact of malicious replicas that try to hamper the state transfer process by using a blacklisting mechanism: when a replica is blacklisted it is not contacted any more during state transfer. The recovering replica requests a data block b_i from one replica, while another f replicas are selected to send just a digest of that block. The recovering replica then computes the digest of the received block and compares this value with the f received digests. If they match, the data block is correct. This represents the best case: the data block is retrieved in a single round, with no bandwidth overhead (i.e. just a single copy of that block is sent). If digests do not match at least one more round is required. We propose two different variations.

a) Variant 1: An additional replica, different from the previous $f+1$, is selected to send another copy of the data block. This process repeats until the recovering replica finds a correct copy of the data block (at most $f-1$ times). This minimizes the bandwidth usage at the cost of additional delay. The received responses (copies of the block and digests) are also used to identify and blacklist malicious replicas that sent invalid information.

b) Variant 2: f additional replicas, different from the previous $f+1$, are selected to send a copy of the data block. Then, the recovering replica has $2f+1$ responses (i.e. $f+1$ blocks and f digests) and uses the same approach described in Section IV-D1 to find a valid block. In addition, these responses are also used to identify and blacklist malicious replicas that sent invalid information. This approach ensures that in at most two rounds the recovering replica finds a correct data block, optimizing latency compared with the previous variant, at the cost of a higher bandwidth overhead.

When a valid copy of the data block is found, the recovering replica moves on to recover block b_{i+1}. It is worthwhile to note that the blacklisting mechanism ensures that variants 1 or 2 are executed no more that $f-1$ times during a state transfer instance. Because we expect f to be much smaller than the number of data blocks, the impact of malicious replicas on state transfer is negligible.

The strategies we propose aim to efficiently retrieve a large state in the expected scenario: when a replica under the control of an adversary has state that is entirely compromised. However, one can implement more conservative strategies, in which a data block is retrieved only if it is compromised. In this case, the recovering replica will collect the digest from $2f+1$ other replicas first, compare this value to the digest of the same block stored locally, and then transfer the block if necessary.

Finally, note that modern database systems implement state transfer strategies (typically in a client-server or publish-subscribe manner) to maintain up-to-date consistent copies of the same database. However, these solutions are not intrusion tolerant and hence not suitable for our system.

E. Prime certificates validation

Pre-order and order certificates, combined with the state, represent the memory of the system. A pre-order certificate is composed of $2f+1$ messages: a client operation o and $2f$ acknowledgments with a digest of that operation. Each operation is uniquely identified by the pair (id, seq_num), where id is the identifier of the replica that injected the update in the system, and seq_num is a local sequence number generated at that replica. As in other BFT protocols, an order certificate in Prime is composed of: (i) a prepare certificate, with a pre-prepare message sent by the leader and $2f$ prepare messages; (ii) a commit certificate, with $2f+1$ commit messages. A pre-prepare message in Prime contains a summary matrix that assigns an order to client operations, while prepare and commit messages contain a digest of that matrix. In addition, a pre-prepare message is uniquely identified by the pair $(view, i)$, where $view$ is the current view number and i is the i^{th} pre-prepare during that view. During Prime operations, when a certificate is ready, a correct replica saves that certificate on the disk. These messages are reloaded into main memory after rejuvenation. In particular, pre-order certificates contain the updates that a rejuvenated replica has to apply to a correct copy of the state at some checkpoint in order to catch up the same execution point as before rejuvenation, if that replica was correct. Order certificates, instead, specify in which order those updates must be applied. Because the adversary can compromise pre-order and order certificates stored on the disk, they must be validated.

1) Validation of order certificates: Order certificates are validated first, because they determine the set of updates that the rejuvenated replica expects to find during the pre-order certificate validation, and the order in which correct updates must be applied to the state.

Because the recovering replica may be missing some messages when it is rejuvenated, it is necessary to estimate the current execution point to catch up. We use the same approach as in [1]: the rejuvenated replica broadcasts a request for the sequence number of the last pre-prepare message for which each replica obtained a prepare certificate. The replica collects sequence numbers, including its own number, and selects the value (v, p) such that at least $2f+1$ different replicas reported a value greater than or equal to p for view v. This is necessary to ensure SAFETY if the rejuvenated replica was correct: p is guaranteed to be no smaller than the value p' proposed by the rejuvenated replica.

Verifying whether a prepare or commit certificate is valid requires the replica to authenticate the pre-prepare message, compute the digest of the summary matrix, and compare this value to the digests contained in prepare or commit messages. If the validation fails for some message, or there is a gap in the pre-prepare sequence, the rejuvenated replica sends a request to another $f+1$ replicas with the sequence number and view of the compromised/missing pre-prepare. Replies include the pre-prepare message, prepare and commit certificates. In this way we are guaranteed that at least one reply contains valid certificates. The verification process is executed for all pre-prepare messages with sequence number up to (v, p).

Note that the validation of an order certificate may involve just the pre-prepare and the $2f+1$ commit messages, because the collection of a correct commit certificate implies that the ordering proposed by the leader has been previously accepted. However, the validation of prepare certificates is necessary to ensure that all Prime messages saved in the persistent storage are correct after rejuvenation.

2) Validation of pre-order certificates: The procedure to validate pre-order certificates is similar to the one described in the previous point. Verifying whether a pre-order certificate is valid requires the replica to compute the digest of the client operation and compare this value with digests in ack messages. Note that in Prime the complete set of update identifiers is obtained from pre-prepare messages.

If the validation fails for some message, or there is a missing update, the rejuvenated replica sends a request to another $f+1$ replicas with the sequence number of the compromised/missing update. Replies include the client operation and $2f$ ack messages. In this way we are guaranteed that at least one reply contains a valid certificate. When pre-order validation completes, the correct set of updates is applied to the state. After that, the Prime replica is correct and is ready to resume Prime operations.

F. Extension to a system with $3f+2k+1$ replicas

Augmenting the number of system replicas requires some minor changes to the Prime protocol. As introduced in Section III-C, the number of messages to order updates and, consequently, the certificates size changes from $2f+1$ to $2f+k+1$. This is necessary because, otherwise, f malicious replicas can send different messages to two distinct sets of $f+1$ replicas, violating SAFETY. The proactive recovery protocol requires some minor changes.

Key replacement: A new session key invalidates all previous keys of the same replica when at least another $2f+k$ replicas receive it. In this way we guarantee that Prime does not order updates signed with old keys.

State validation: During the state validation the recovering replica still requires $f+1$ replies with the same digest. Even in the case when the recovering replica has no valid checkpoint, it waits for $2f+1$ replies and selects the checkpoint ck^* with the $f+1^{th}$ highest sequence number.

State transfer: The strategy described in Section IV-D1 still needs $f+1$ copies of a data block and f digests. At most f replies, in fact, may contain invalid information, and the remaining $f+1$ replies are enough to find a correct copy of the block. The strategy described in Section IV-D2, including the two variants, requires no additional change.

Certificate validation: The size of Prime certificates increases from $2f+1$ to $2f+k+1$, while the transfer of missing/compromised certificates still requires $f+1$ replies.

V. EVALUATION

A. State transfer strategies comparison

In this section we show which state transfer strategy works better based on the size of the state to transfer. We present a theoretical comparison in terms of latency and bandwidth consumption between the two state transfer strategies described in Section IV-D. We also consider a third strategy, in which a recovering replica only retrieves a correct copy of a data block when that block on its disk has been compromised.

We define the following parameters:

- f, the number of compromised replicas;
- b, the number of data blocks into which the state is partitioned;
- s, the size of each data block;
- t_b, the time to collect and write to disk a valid copy of a data block;
- t_d, the time to collect f digests;
- c, the fraction of compromised data blocks, which is used to compute latency and bandwidth consumption when using the third state transfer strategy mentioned above.

We ran an experiment on a local cluster of seven machines to compute the average t_b and t_d. These values were $t_b = 113$ ms and $t_d = 5$ ms. The hardware and software configuration of the local cluster is described in Section V-B.

The formulas to compute the latency L and the bandwidth consumption B when using the state transfer strategy that reduces latency (Section IV-D1) are:

$$L = b \cdot t_b$$

$$B = (f+1) \cdot s \cdot b$$

For the state transfer strategy that reduces bandwidth overhead (Section IV-D2), we apply the solution described in Section IV-D2a if the recovering replica obtains an invalid copy of a data block or digest. In this case, the recovering replica collects another copy of the data block at a time until it finds a correct copy. The formulas to compute the latency L and the bandwidth consumption B become:

$$L = (b \cdot t_b) + ((f-1) \cdot t_b)$$

$$B = (s \cdot b) + ((f-1) \cdot s)$$

The first product in the two equations refers to the best case, when the recovering replica obtains valid replies. The second product refers to the case in which the recovering replica receives invalid replies. Due to our blacklisting mechanism, this can happen at most $f-1$ times. In these two strategies the time to collect f digests overlaps with the time to collect a data block.

The third state transfer strategy we consider transfers a data block only if necessary. The recovering replica retrieves a valid digest from at least $f+1$ correct replicas and compares this value with the digest of the data block on its local disk. A request for a copy of that data block is sent only if the digests do not match. Even in this case, we use the blacklisting mechanism to reduce the impact of malicious replicas. Because the recovering replica sends requests for digests in parallel, we assume that the time to collect a valid digest from at least $f+1$ correct replicas is still t_d. The formulas to compute the latency L and the bandwidth consumption B depend on the fraction c of compromised data blocks:

$$L = (b \cdot t_d) + (c \cdot b \cdot t_b) + (c \cdot (f-1) \cdot t_b)$$

$$B = c \cdot s \cdot b + (c \cdot (f-1) \cdot s)$$

The first product in L refers to the collection of digests, while the second product refers to the recovery of correct data blocks. The third product in L and the second product in B refer to the case in which the recovering replica receives invalid data blocks.

Figures 2 and 3 show the results of our analysis for $f = 2$, $b = 1$ megabyte, and $c = \{0.2, 0.6, 1\}$. Figure 2(a) and 2(b) depict the latency (in hours) to transfer states of different sizes. We refer to the state transfer strategy that minimizes latency as *Min Lat* and to the one that reduces bandwidth consumption as *Red Bdw*. We refer to the state transfer strategy that transfers data blocks only when necessary as *Min Bdw*. Thanks to the blacklisting mechanism that we use in *Red Bdw*, the impact of malicious replicas is negligible, and the difference in time between the *Min Lat* and *Red Bdw* is negligible as well (the curves in the figures overlap). *Min Bdw* reduces latency when the state is not entirely compromised. If the state is totally compromised, this approach achieves the worst performance because digests and data blocks are transferred in two different rounds. In the presence of a state of 1 terabyte, *Min Bdw* spends additional 83 minutes to transfer a correct copy.

The big difference between *Min Lat* and *Red Bdw* lies in the bandwidth consumption, as shown in Figures 3(a) and 3(b). *Min Lat* recovers the state $f+1$ times and this is very expensive when the size of the state is large. On the contrary, *Red Bdw* transfers a single copy, while *Red Bdw* achieves optimal bandwidth usage.

This analysis tells us that *Min Lat* can be used only when the size of the state is small (e.g. up to 10 gigabytes). Figure 3(b) shows how the bandwidth usage grows rapidly when using *Min Lat*. The implementation of *Red Bdw* is less straightforward because it requires a backup strategy when a recovering replica receives invalid replies and a blacklisting mechanism to reduce the impact of malicious replicas. However, *Red Bdw* is almost as fast as *Min Lat* and drastically reduces the bandwidth consumption. *Min Bdw* should be used when bandwidth really matters. As shown in Figures 3(a) and 3(b) it achieves optimal bandwidth usage. However, it is slower than *Min Lat* and *Red Bdw* when the state is totally compromised (e.g. when an adversary deletes the entire contents of a database with a single command).

In Section V-C we present an experimental evaluation based on a real implementation that uses *Red Bdw* to transfer the state. We extend this technique to retrieve multiple data blocks in parallel and balance the load across correct replicas.

B. System deployment

In this section we describe the deployment of the survivable Prime system in a physical and virtualized environment.

1) Deployment in a physical environment: The deployment in a physical environment is depicted in Figure 4. The proactive recovery logic runs in a separate computer and includes the scheduler that periodically rejuvenates one replica at a time, in a round robin fashion. The computer is connected to a network switch that, in turn, connects to netbooters (i.e. remotely activated power switches), one for each physical server that hosts a Prime replica. The network that connects the computer with the proactive recovery logic, switch, and netbooters is completely isolated from outside communication, and thus cannot be accessed by the attacker. Physical servers are connected to netbooters through power wires. Periodically, the proactive recovery scheduler activates one of the netbooters to cycle the power and reboot the corresponding server. After rebooting, the replica executes the proactive recovery protocol as described in Section IV.

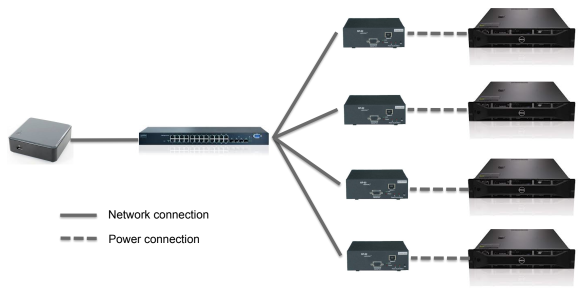
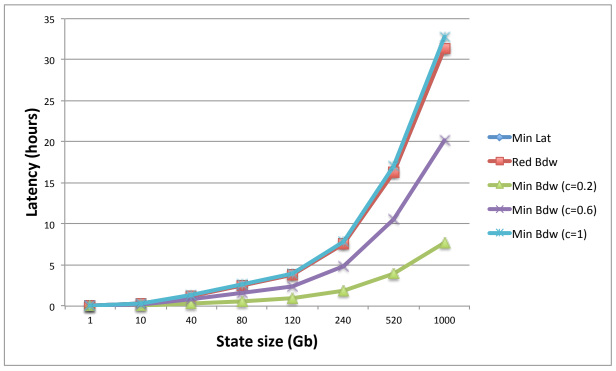


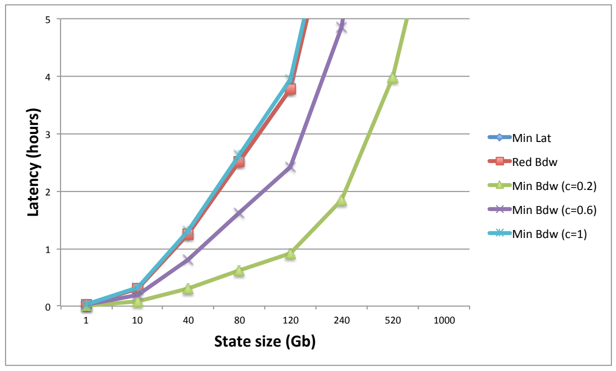
Fig. 4. Deployment in a physical system, with the proactive recovery logic isolated from the external network.

2) Deployment in a virtualized environment: The deployment in a virtualized environment is depicted in Figure 5, and is based on Xen Cloud Platform (XCP). The proactive recovery logic runs in the privileged *Dom0* domain on a separate physical server, while Prime replicas run in Xen *guest domains*. XCP allows creating a *resource pool*: a virtualized system that can be managed from a single hypervisor, i.e. the master, which can instruct other hypervisors to execute specific operations (e.g. creating/destroying virtual machines). The proactive recovery logic runs in the master hypervisor, which communicates with other hypervisors through a private network.

The advantage of a deployment in a virtualized environment is that rejuvenation is much shorter than in a physical environment: a shadow virtual machine, in fact, can be instantiated in advance, with a clean copy of the operating system and a new version of Prime. When the shadow virtual machine is ready to

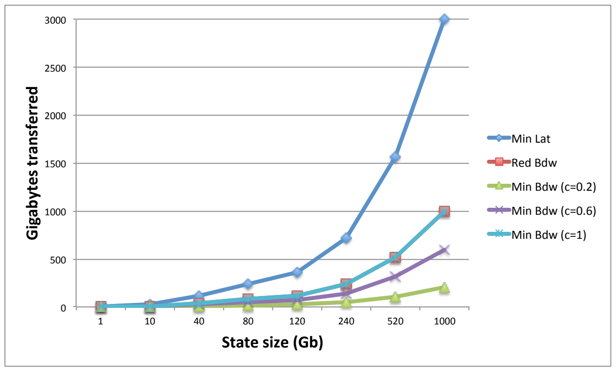


(a)

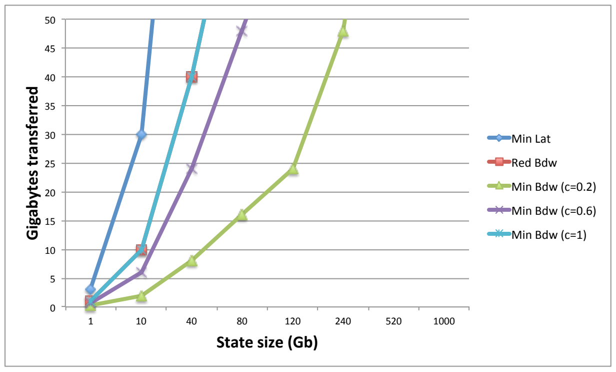


(b)

Fig. 2. (a) Theoretical computation of the time taken to transfer states of various sizes by using different state transfer strategies. (b) Zoom into the time interval 0-5 hours.



(a)



(b)

Fig. 3. (a) Theoretical computation of the bandwidth consumed to transfer states of various sizes by using different state transfer strategies. (b) Zoom into the bandwidth usage interval 0-50 gigabytes.

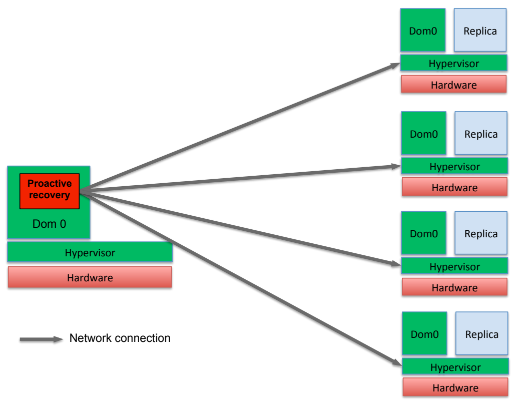


Fig. 5. Deployment in a virtualized system based on XCP.

start, the old one can be destroyed. We built a prototype that, as soon as the new virtual machine is ready to take over, shuts down the old virtual machine, detaches the virtual disk with the state and the virtual network interface from that machine, and attaches them to the new one. We measured the time taken to complete these operations to be approximately 8 seconds. Note that XCP does not allow destroying a virtual machine before shutting it down; the shut down operation is what most impacts the time to transition. Finally, the old virtual machine is destroyed, while the second one can initiate the proactive recovery protocol.

The virtualized environment poses two drawbacks: (i) the server that runs proactive recovery logic can be compromised if the attacker compromises the hypervisor; (ii) a virtualized environment offers poor or no support for TPM (i.e. solutions present in literature are either not implemented or only partially implemented), which is a fundamental component in our survivable system. Regarding point (i), a deployment in a virtualized environment is suggested if one trusts the hypervisor. This can be substantiated, as an example, by rebooting physical servers from time to time and verifying the status of the servers through a chain of trust (i.e. digest of BIOS, operating system kernel, ...) computed by the TPM. Regarding point (ii), instead, we exploit the fact that *Dom0* has full access to the underlying hardware to implement a TPM manager that mediates the interaction between the Prime replica in a virtual machine and the physical TPM. Each time the Prime replica has to access the TPM, it contacts the TPM manager in *Dom0*, which, in turn, accesses the TPM and replies back to the Prime replica with the outcome of the specific operation.

TABLE I
STATE VALIDATION AND TRANSFER MEASUREMENTS FOR DIFFERENT STATE SIZES AND NUMBER OF SYSTEM REPLICAS.

state size	state reading	state transfer		
		4 replicas	7 replicas	10 replicas
1 Gb	9 sec	36 sec	25 sec	23 sec
10 Gb	1 m, 27 sec	6 m	4 m	4 m
40 Gb	5 m, 47 sec	24 m	15 m	15 m
80 Gb	11 m, 30 sec	48 m	31 m	31 m
120 Gb	17 m, 15 sec	1 h, 12 m	48 m	48 m
240 Gb	34 m, 30 sec	2 h, 24 m	1 h, 38 m	1h, 36 m
520 Gb	1 h, 14 m	5 h, 9 m	3 h, 28 m	3 h, 24 m
1 Tb	2 h, 24 m	9 h, 50 m	6 h, 17 m	6 h, 17 m

C. *Experimental results*

We deploy the system in a physical cluster with $3f + 1$ servers. Each server is a Dell PowerEdge R210 II, with an Intel Xeon E3 1270v2 3.50 GHz processor and 16GB of memory. All servers are connected by Solarflare 5161T 10GbE cards and an Arista 7120T-4S switch, providing 10 Gigabit Ethernet. All servers run CentOS 6.2 as the operating system. In the following we measure the time to complete state validation and transfer in systems with 4, 7, and 10 replicas. For these experiments we use Prime 2.0, which includes proactive recovery and state transfer and is available at [14]. Prime 2.0 uses an intrusion tolerant communication substrate built on top of Spines 4.0 [15].

1) State validation and transfer: Table I shows the measurements for state reading and transfer for different state sizes, from 1 gigabyte to 1 terabyte. These state sizes are quite common in SCADA systems, military command and control, banking systems, communication systems, to name a few. We evaluate the state transfer strategy described in Section IV-D2 that minimizes the bandwidth usage. The whole state is fragmented into data blocks of 1 megabyte. We transfer these blocks in parallel, 5 at a time. During the experiment, we noticed that the time to transfer the state grows linearly with the number of data blocks. The performance hit obtained in a system with 4 replicas is due to the fact that some replicas are sending more data blocks at the same time, which produces a higher CPU consumption that limits the processing speed of the physical servers that host those replicas.

Based on the obtained results, in Section VI we will show how to set the rejuvenation rate in our theoretical model. The model will output the minimum security requirement that a replica has to guarantee in order to achieve a desired confidence in the system over its lifetime.

VI. THEORETICAL REJUVENATION MODEL

In this section we present a theoretical model that allows a system administrator to determine the deployment parameters needed to reach the desired confidence in the system.

As a building block, we present an equation that computes the resiliency of the system based on the rejuvenation rate and the number of replicas. The equation takes as input: (i) the probability c that a replica is correct over a year; (ii) the rejuvenation rate r, i.e. the number of rejuvenations per day across the whole system, which is limited by the system's state size; (iii) the number of replicas n; and (iv) the system lifetime y. A system administrator has control of r and n, but he does not have control of c, which represents the strength of a replica. The strength of a replica can be estimated with CERT alerts, bug reports, or other historical information, as in [16]. The output of the computation is the probability the system will remain correct over its lifetime.

By iterating this computation many times with different possible values of r and n, the system administrator can determine the replica strength required to reach the desired confidence in the system. Alternatively, if the replica strength is fixed, the same computation can be iterated to find the required values of r and n to reach the desired confidence.

A. *Model description*

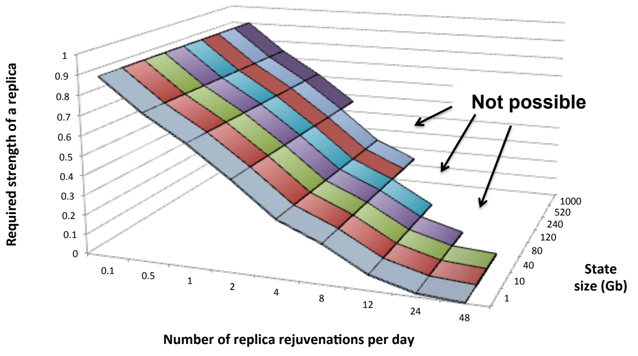
In our model we consider only failures related to software aspects, not hardware failures. We assume that every replica has the same probability of being compromised over a year. In addition, we assume that each replica fails independently. We support this assumption by making extensive use of diversity as explained before. Replicas are periodically rejuvenated one at a time in a round robin fashion. For simplicity's sake we optimistically assume that recovery is instantaneous. However, in Section VI-C we extend this model to relax this assumption at the cost of an additional $2k$ replicas in the computation and pessimistically assume that a recovery operation spans over an entire rejuvenation cycle.

In order to compute the probability the system is correct over its lifetime, we first define p as the probability that a replica is correct at the end of an inter-rejuvenation period by simply scaling the probability c to the appropriate time interval: $p = \sqrt[365 \cdot r]{c}$. The following equation then gives the probability the system will survive over a lifetime of y years.

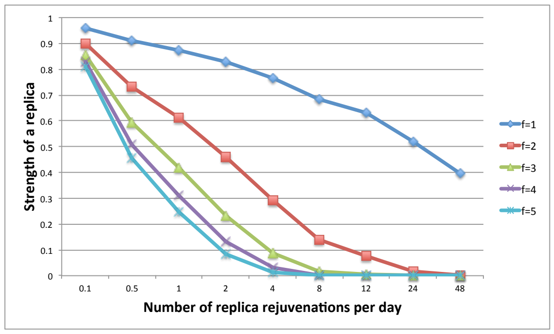
$$\left(\sum_{i=n+1-f}^{n+1} \text{coeff} \left(\prod_{j=1}^{n} \left(\left(1 - p^j\right) + p^j x \right) \right) [i] \right)^{y \cdot 365 \cdot r}$$

The innermost part of the equation is a product of the form $\prod_{j=1}^{n} \left(\left(1 - p^j\right) + p^j x \right)$. Each term of this product corresponds to one replica, where p^j is the probability that the replica will remain correct to the end of the current rejuvenation round and $1 - p^j$ is the probability that the replica will be compromised at the end of current rejuvenation round. This is because that replica has not been rejuvenated for the last $j - 1$ rounds and must have survived all of them in addition to the current round in order to remain correct at the end of the current round. By taking the product of these terms, the i^{th} coefficient in the resulting polynomial is formed by the sum of all possible combinations of the $p^j x$ components of i of the terms with the $(1 - p^j)$ components of the remaining $n - i$ terms. Thus, the i^{th} coefficient represents the probability that exactly i replicas will remain correct at the end of the current rejuvenation round.

We sum these probabilities for all the cases when the system would survive the round, i.e. the coefficients of the polynomial from the $(n + 1 - f)^{th}$ coefficient to the $(n + 1)^{th}$ coefficient.

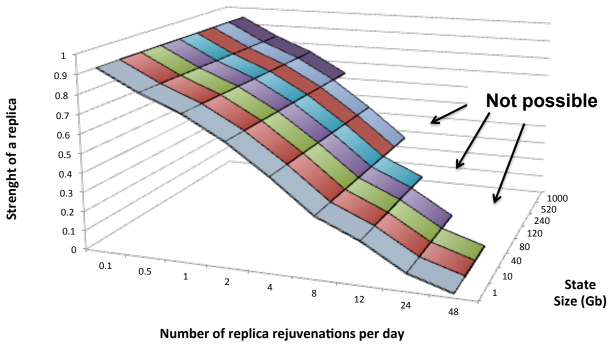


(a)

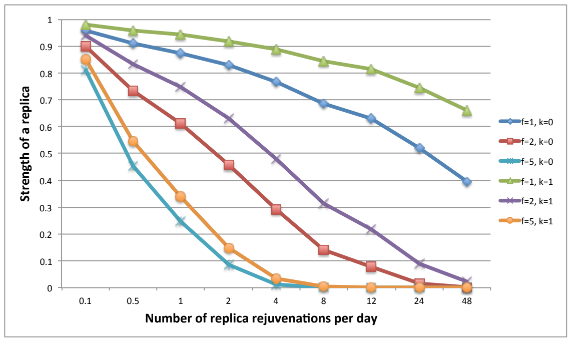


(b)

Fig. 6. Required strength of a replica to achieve a confidence in the system of 95% over 30 years in a system with $3f + 1$ replicas. We vary: (a) the rejuvenation rate and state size, while the number of replicas in the system is 7; (b) the rejuvenation rate, for different number of replicas in the system.



(a)



(b)

Fig. 7. Required strength of a replica to achieve a confidence in the system of 95% over 30 years in a system with $3f + 2k + 1$ replicas, where $k = 1$. In Figure 7(a) the number of replicas is 9 ($f = 2$). The required strength of a replica increases compared to the case with $3f + 1$ replicas (cfr. Figures 6(a) and 6(b)).

This excludes any cases when more than f replicas have failed. This sum then gives the probability that the system will survive one rejuvenation round.

In fact, there are many rejuvenation rounds in the system's lifetime, which can be treated as a repeated experiment. In order for the system to succeed for its entire lifetime, it must succeed for every rejuvenation round in its lifetime. The probability of success, then, is the product of the probabilities of success for all the rejuvenations rounds, here represented as the exponent $30 \cdot 365 \cdot r$, to capture the y year lifetime, 365 days per year, and r rejuvenations per day.

B. Application to realistic settings

Let us apply the model to a concrete example based on the measured time to complete state transfer reported in Section V, Table I. Figure 6(a) shows the required strength of a replica to achieve a confidence in the system of 95% over 30 years in a system with 7 replicas when varying the number r of rejuvenations per day and the state size. The time to transfer the state determines the maximum rejuvenation rate possible for each state size. In the presence of large state, some of the configurations are not possible. As an example, we cannot have $r > 2$ when the state size is 1 terabyte, because the total time to validate and transfer that state is 8 hours and 41 minutes. Figure 6(a) shows that increasing r, when possible, allows the system to survive in the presence of weak replicas.

In Figure 6(b) we vary the rejuvenation rate r, for different numbers of tolerated failures f. We assume the state size is small enough to allow multiple rejuvenations per day (e.g. 1, 10, or 40 gigabytes). Increasing the number of replicas (by increasing f) allows the system to survive in the presence of weaker replicas, but the benefit of adding replicas decreases as f increases.

The results presented in this section show that proactive recovery is possible in the presence of large state. Some papers in the literature [1], [5] use an aggressive rejuvenation rate (one rejuvenation every 5, 10, or 15 minutes), but these rates limit the size of the state to transfer and incur unnecessary overhead. Our model shows that a rejuvenation rate of one replica per day requires a replica strength of 0.54 when the state size is 1 terabyte. In addition, our model helps to determine the number of replicas to deploy. Many BFT protocols in literature are evaluated with 4 replicas (i.e. $f = 1$). The model shows that a system with 4 replicas requires stronger replicas, even in the presence of multiple rejuvenations per day. At the same time, deploying systems with more than 10 replicas (i.e. $f > 3$) does not increase the resiliency of the system remarkably, and it generates additional overhead in the BFT protocol.

C. *Extension to a system with* $3f + 2k + 1$ *replicas*

So far, the model has assumed that rejuvenations occur instantaneously. Since the system has state, which takes time to verify or transfer, rejuvenations actually take some time to complete. In this section, we extend the model by conservatively assuming that each rejuvenation takes an entire rejuvenation round to complete.

A system with $3f+2k+1$ replicas can tolerate f Byzantine faults and k benign faults simultaneously, as previously shown in [5]. The changes to the protocol necessary to support the additional $2k$ replicas are specified in Section IV-F. Since each rejuvenating replica behaves as a replica experiencing a benign fault until it finishes its rejuvenation, we choose k to be 1, reflecting the fact that we rejuvenate one replica at a time. Then, the system is able to tolerate f Byzantine failures while rejuvenating some other correct replica without sacrificing availability.

To extend our model from $3f + 1$ to $3f + 2k + 1$, we change the value of n accordingly. The equation in the model is unchanged.

We now show the results of the same analysis as in the previous section, but with this extended model. The salient difference here is that we seek a 95% confidence that the system will remain *correct and available* for 30 years, when in the previous section we sought a 95% confidence that the system will remain *correct but not necessarily always available* for 30 years. Critical applications that need performance guarantees under attack should consider paying the cost of $2k$ additional replicas to continue providing availability during rejuvenations.

The results of this new analysis are shown in Figures 7(a) and 7(b). In Figure 7(a) the number of replicas is 9 ($f = 2, k = 1$). Note that the curves with $k = 0$ in Figure 7(b) represent the same system configuration as reported in Figure 6(b). Compared to Figures 6(a) and 6(b), in Figures 7(a) and 7(b) we note an increased required strength of a replica to achieve the desired confidence in the system over its lifetime. This increase is driven by the $2k$ additional replicas in the system, required to maintain both correctness and availability, while the number of tolerated faults still remains the same as in the case of $3f + 1$ replicas. However, Figure 7(b) shows that this difference decreases as we increase the number of tolerated faults, f.

VII. Construction of a survivable SCADA system

Critical infrastructures rely on SCADA systems for advanced monitoring and control capabilities. A master server connects to remote terminal units (RTUs) to gather information about the status of remote equipment. This information is processed at the SCADA master and sent to a human-machine interface (HMI) for visualization. The SCADA master represents the most important component of the system. It must work correctly and at an acceptable level of performance at all times.

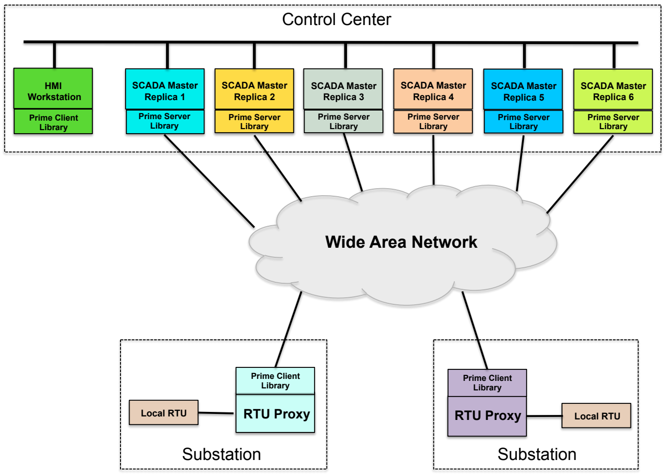


Fig. 8. Architecture of a survivable SCADA system with $3f+2k+1$ replicas, with $f = 1, k = 1$. The master server is replicated using Prime.

The work in [10] integrates Prime into the Siemens corporation commercial SCADA product for the power grid. Prime is used to replicate the SCADA master in order to tolerate malicious attacks, while in real-world deployments of SCADA systems the master is replicated through a hot-standby configuration that survives only benign failures. That work focuses on how to address the architectural mismatch for the integration of Prime into the SCADA system. As with other BFT protocols, Prime takes action in response to unsolicited clients requests, while in SCADA systems the master server also executes operations in response to requests pulled from RTUs. To solve this problem, [10] extends Prime with an intrusion tolerant *logical timeout protocol* to synchronize replicas in order to pull information from RTUs at the same time, and an *intrusion tolerant reliable channel* to guarantee that legitimate messages are delivered exactly once, in sequence number order.

The work in [10] is still vulnerable to intrusions: replicas have an identical attack surface and the system is expected to operate over a long period of time. In this section we use defense across space and time to build a SCADA system that is survivable over long lifetime.

Figure 8 shows the architecture of a SCADA system with $3f + 2k + 1$ replicas, so to continue providing availability during rejuvenations. The SCADA master is replicated through Prime. Each machine (physical or virtualized, depending of the chosen deployment strategy) has access to a Prime server library, which implements the replication logic.

Periodically SCADA master replicas are rejuvenated and new variants of the SCADA master and Prime server library are created with the MultiCompiler. To do so, each physical server has a read-only medium that contains a fresh copy of the operating system, the TPM public keys, the MultiCompiler, the Prime bitcode, and the SCADA master bitcode. The MultiCompiler guarantees with very high probability that a master replica and its Prime server library are different from all previous, current, and future variants. We require that physical servers are equipped with the TPM, in order to replace session keys after rejuvenation.

Each SCADA master replica has a copy of the *historian*, a database that contains timestamped data, events, and alarms that can be queried by the HMI workstation to display the status of the critical infrastructure. After rejuvenation of a SCADA master replica, the newly generated Prime server library takes care of validating the content of the historian and transferring (if necessary) a clean copy. A system administrator can decide how often SCADA master replicas should be rejuvenated, based on the size of the historian.

RTUs and HMI workstation use a Prime client library to communicate with the replicated SCADA master. The Prime client library implements the intrusion tolerant reliable channel described in [10]. This library gives the HMI workstation and RTUs the impression that they interact with a single server. Each message sent by the HMI workstation or an RTU is forwarded to $f + 1$ different replicas of the SCADA master, such that at least one of them is correct and we are guaranteed that the message is introduced into Prime for global ordering. In the same way, the library forwards a message to the HMI workstation or an RTU after it received $f + 1$ identical replies [1] from different SCADA master replicas.

In order to improve the resiliency of the whole system, we can replicate the RTU proxies that connect local RTUs to the replicated SCADA master. Each proxy replica has its own Prime server library to replicate the state of a local RTU and agree on the order of the values sent to the SCADA master. In the same way, we can apply defense across space and time also to local RTU proxies. Moreover, each replica of a proxy uses the Prime client library to communicate with the replicated master. This solution guarantees intrusion tolerance also at substation level, at the cost of additional bandwidth usage: each master replica receives the same message from at least $f + 1$ replicas of an RTU proxy. However, each operation is ordered exactly once, because Prime replicas discard duplicate messages.

VIII. Related work

A. Proactive rejuvenation

Software rejuvenation [17] was introduced to increase the reliability and availability of continuously running applications in order to prevent failures over time due to software aging. Both [17] and [18] formulate the rejuvenation model through a Markov chain and derive the optimal rejuvenation schedule.

More recently, software rejuvenation has been used to proactively recover replicas in a malicious environment. Castro in [1] was the first to present a proactive recovery protocol for BFT systems, addressing issues such as the need for unforgeable cryptographic material, rebooting from read-only memory, and efficient state transfer. In our paper we extend the work in [1] by describing how to obtain fine-grain diversity and efficient state transfer in the presence of a large state.

Rodrigues et al. in BASE [7] use data abstraction techniques in order to mask software errors. The abstraction layer hides implementation details and allows the reuse of off-the-shelf software components. Replicas are periodically rejuvenated, rebooting from a clean state. Authors in [19] propose splitting the system into a synchronous component that activates periodic rejuvenation and an any-synchronous subsystem that includes the payload application. This model has been adopted later in other papers [5], [6], [20], [21]. In particular, authors in [5] enhance proactive rejuvenation with reactive recovery, which allows the recovery of replicas as soon as they are detected or suspected of being compromised. The proposed solution guarantees the availability of the minimum number of replicas required to make progress by using $2k$ additional replicas, where k is the maximum number of replicas that recover at the same time. In our paper we also describe how to adapt our protocol to work with $3f + 2k + 1$ replicas, and separate the proactive recovery logic from the replication engine. However, in contrast to [5] we avoid any direct interaction between these two modules because the bidirectional communication between the trusted proactive recovery logic and the untrusted application could open the possibility for potential attacks.

Theoretical models for proactive recovery have been presented in [22] and [23]. The model in [22] computes the system availability by varying the time between rejuvenations of different replicas, adopting parallel and serial rejuvenation schemes. [23] instead presents a theoretical assessment of FOREVER, a service that enhances the resiliency of intrusion-tolerant replicated systems through recovery and diversity, which is implemented through operating systems and configuration diversity rules. The theoretical model evaluates the system failure probability by varying the rejuvenation rate. The authors also consider the probability of common vulnerabilities in their model. In our work we concentrate on serial rejuvenations and do not consider the common vulnerabilities among replicas due to the high entropy among variants generated by the MultiCompiler. In addition, we define the rejuvenation rate taking into account the size of the state that may need to be transferred and the time to transfer it, while the models in [22] and [23] are designed for stateless systems.

The closest work to ours is presented by Schneider et al. in [4]. The paper presents *proactive obfuscation*, a method that uses semantic-preserving code transformations to create replicas that are likely to share few vulnerabilities. The proposed solution relies on a component trusted to periodically initiate proactive obfuscation (as in our system), during which one replica at a time reboots from a fresh copy of the application code and a clean copy of the state is obtained from other replicas through an internal service network. This approach is used to implement two prototypes: a distributed firewall and a distributed storage service, both with small state size (few hundreds kilobytes) and arbitrary rejuvenation rate (on the order of minutes). In our work instead we augment the built-in operating system tools with a compiler-based diversification engine and support large-state applications.

Finally, it is worth mentioning that our solution is built on Prime [8], which guarantees performance under attacks. Recently other BFT protocols with performance guarantees have been presented. Aardvark [24] guarantees that over sufficiently long periods the system throughput remains within a constant factor of what it would be if only correct replicas were participating in the protocol. BFT-Mencius [25], like Prime, ensures *bounded-delay* in a period of network stability. In addition, the protocol reduces the cost of the pre-ordering phase by using a multi-leader approach, in which replicas propose an order through an atomic broadcast primitive.

B. State transfer

State transfer for BFT protocols has been previously presented in [1]. The proposed approach is designed for a small state (few hundreds of kilobytes), and hence it is not suitable for the kind of systems that we target. The state is organized as a tree of digests and is always retrieved from the latest checkpoint. If the system makes progress to the next checkpoint before a rejuvenated replica completes state transfer, that replica must restart state transfer. In a system with large state this would generate a cascade of state transfer executions that would not end until another f replicas fail/rejuvenate. At that time the recovering replica can finally catch up, at the cost of a period of service unavailability.

Authors in [6] present two state transfer strategies that are specifically tailored for a virtualized environment, exploiting the fact that more virtual replicas can share the same physical hardware. The work in [21] speeds up proactive recovery by rejuvenating several replicas at the same time and restarting the execution from a previously saved correct state.

A collaborative state transfer protocol based on sequential checkpointing has been presented in [26]. Replicas checkpoint their state at different points of execution, in groups of at most f replicas each, in order to minimize the performance impact of taking a snapshot. The state of a replica consists of a checkpoint and a log file with all operations executed between two consecutive checkpoints at that replica. The log file is composed of many segments. During collaborative state transfer, the recovering replica retrieves the checkpoint and all log segments from the replica with the $f + 1^{th}$ most recent checkpoint, while the first f replicas provide a digest of segments and checkpoint. The proposed approach is effective against the performance degradation generated by the typical checkpointing mechanisms. While the solution in [26] is very efficient for transferring a large log, in this paper we are more interested in transferring a large checkpoint.

IX. Conclusion

The increasing reliance on critical systems necessitates the construction of systems that are able to tolerate intrusions. In this paper we described the construction of a practical intrusion tolerant replication system, which offers defense across space (*diversity*) and time (*proactive recovery*). The innovative aspects of our work include: (i) a theoretical model that computes how resilient the system is over its lifetime; (ii) the description of an architecture for survivable SCADA systems; and (iii) the support for large state applications, with a theoretical evaluation of different state transfer strategies based on the application constraints. We ran state transfer for different state sizes and measured the maximum rejuvenation rate and the minimum strength of a replica required to achieve high confidence in the system (e.g. 95%) over 30 years.

References

[1] 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.

[2] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012.

[3] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[4] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 2, p. 4, 2010.

[5] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 4, pp. 452–465, 2010.

[6] T. Distler, R. Kapitza, and H. P. Reiser, "Efficient state transfer for hypervisor-based proactive recovery," in *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems–WRAITS*. ACM, 2008, p. 4.

[7] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 15–28, 2001.

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Dependable Systems and Networks with FTCS and DCC. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 197–206.

[9] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 2013, pp. 1–11.

[10] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare, "Survivable SCADA via intrusion-tolerant replication," *IEEE Transaction on Smart Grid*, vol. 5, no. 1, pp. 60–70, 2014.

[11] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, 1978, pp. 3–9.

[12] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering, IEEE Transactions on*, no. 1, pp. 96–109, 1986.

[13] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[14] "Prime 2.0," http://www.dsn.jhu.edu/byzrep/prime.html.

[15] "Spines 4.0," http://www.spines.org/.

[16] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 383–394.

[17] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE, 1995, pp. 381–390.

[18] T. Dohi, K. Goševa-Popstojanova, and K. Trivedi, "Estimating software rejuvenation schedules in high-assurance systems," *The Computer Journal*, vol. 44, no. 6, pp. 473–485, 2001.

[19] P. Sousa, N. F. Neves, and P. Verissimo, "How resilient are distributed f fault/intrusion-tolerant systems?" in *Dependable Systems and Networks. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 98–107.

[20] P. Sousa, A. N. Bessani, and R. R. Obelheiro, "The forever service for fault/intrusion removal," in *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems–WRAITS*. ACM, 2008, p. 5.

[21] F. Zhao, M. Li, W. Qiang, H. Jin, D. Zou, and Q. Zhang, "Proactive recovery approach for intrusion tolerance with dynamic configuration of physical and virtual replicas," *Security and Communication Networks*, vol. 5, no. 10, pp. 1169–1180, 2012.

[22] L. Teixeira Aguiar Norton Brandão and A. Neves Bessani, “On the reliability and availability of replicated and rejuvenating systems under stealth attacks and intrusions,” *Journal of the Brazilian Computer Society*, vol. 18, no. 1, pp. 61–80, 2012.

[23] A. Bessani, A. Daidone, I. Gashi, R. Obelheiro, P. Sousa, and V. Stankovic, “Enhancing fault/intrusion tolerance through design and configuration diversity,” in *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems–WRAITS*, vol. 9, 2009.

[24] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, “Making byzantine fault tolerant systems tolerate byzantine faults.” in *NSDI*, vol. 9, 2009, pp. 153–168.

[25] Z. Milosevic, M. Biely, and A. Schiper, “Bounded delay in byzantine tolerant state machine replication,” in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. IEEE, 2013, pp. 61–70.

[26] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, “On the efficiency of durable state machine replication,” in *USENIX ATC*, 2013.