Survivable SCADA Via Intrusion-Tolerant Replication

Jonathan Kirsch, Stuart Goose, Yair Amir, *Member, IEEE*, Dong Wei, *Member, IEEE*, and Paul Skare, *Member, IEEE*

***Abstract*—Providers of critical infrastructure services strive to maintain the high availability of their SCADA systems. This paper reports on our experience designing, architecting, and evaluating the first *survivable* SCADA system—one that is able to ensure correct behavior with minimal performance degradation even during cyber attacks that compromise part of the system. We describe the challenges we faced when integrating modern intrusion-tolerant protocols with a conventional SCADA architecture and present the techniques we developed to overcome these challenges. The results illustrate that our survivable SCADA system not only functions correctly in the face of a cyber attack, but that it also processes in excess of 20 000 messages per second with a latency of less than 30 ms, making it suitable for even large-scale deployments managing thousands of remote terminal units.**

***Index Terms*—Cyber attack, fault tolerance, reliability, resilience, SCADA systems, survivability.**

I. Introduction

SUPERVISORY Control and Data Acquisition (SCADA) systems form the backbone of many vital services, such as electricity transmission and distribution, water treatment, and traffic control. As key components of our critical infrastructure, SCADA systems must continue operating correctly and at their expected level of performance at all times. In practice, ensuring such continuous availability requires the capability to tolerate and overcome various types of faults that arise in large distributed systems, including "benign" faults (e.g., hardware crashes, power failures, and network partitions) and more severe faults, including potentially malicious cyber attacks.

Unfortunately, contemporary SCADA systems exhibit an *availability gap* that leaves them vulnerable to downtime. While today's systems are able to withstand effectively many types of benign faults using hardware and software redundancy techniques (e.g., primary/hot standby [1]), their ability to survive in the face of cyber attacks remains limited. Many SCADA systems were designed to operate on isolated, private networks, but this assumption of an "air gap" no longer holds in many modern deployments: interoperability goals and the need to provide access to more grid stakeholders mean that the SCADA system is often connected to enterprise IT infrastructure, inheriting the associated vulnerabilities. SCADA has also become an increasing target for cyber attacks [2], resulting in an arms race between attackers and SCADA vendors and operators. Furthermore, although today's systems employ a defense-in-depth approach to security that focuses on *preventing* attacks, it is impossible to prevent all attacks; insider attacks, in particular, pose a growing threat to critical infrastructure [3].

This paper reports on our experiences to date designing and implementing the first *survivable* SCADA system.[1] By survivable, we mean that the SCADA system continues to *operate correctly* and with *minimal performance degradation* even if malicious attacks compromise part of the system. These twin properties are essential for maintaining high availability in the face of cyber attacks.

To achieve survivability, our system employs *intrusion-tolerant replication* [5], [6]. It runs, in parallel, several copies of the SCADA Master application; the copies collectively behave as a single *logical SCADA Master* that provides correct, timely service as long as less than a threshold fraction of the copies is compromised. Intuitively, intrusion tolerance allows an application to act as its own firewall, providing protection even if the system's security perimeter is breached. A distinguishing feature of intrusion-tolerant systems is that they do not require prior knowledge of attack signatures and behaviors to provide their guarantees.

Intrusion-tolerant replication protocols have been well-studied in the distributed systems community over the last decade (e.g., [6]–[11]), and in this paper we build on this previous research. Specifically, we use the Prime replication protocol [5], [6] as a fundamental building block in our survivable SCADA system. However, we were confronted with two significant challenges when attempting to integrate Prime with a SCADA system. First, existing intrusion-tolerant replication systems, including Prime, implicitly assume that the application being replicated is client driven (i.e., the server application takes action only in response to unsolicited requests submitted by clients). By contrast, in a SCADA system, the SCADA Master application also processes solicited requests, which are pulled from field devices by a *server-driven* polling

Manuscript received December 05, 2012; revised March 18, 2013, May 10, 2013; accepted June 10, 2013. Date of publication August 07, 2013; date of current version December 24, 2013. The work of Y. Amir was supported in part by DARPA Grant N660001-1-2-4014. The content of this paper is solely the responsibility of the authors and does not represent the official view of DARPA or the Department of Defense. Paper no. TSG-00841-2012.

J. Kirsch and S. Goose are with Siemens Technology-To-Business Center, Berkeley, CA 94704 USA (e-mail: jonathan.kirsch@siemens.com; stuart.goose@siemens.com).

Y. Amir is with Johns Hopkins University, Baltimore, MD 21218 USA (email: yairamir@cs.jhu.edu).

D. Wei is with Siemens Corporation, Corporate Technology, Princeton, NJ 08540 USA (email: dong.w@siemens.com).

P. Skare is with Pacific Northwest National Laboratory, Richland, WA 99352 USA (email: paul.skare@pnnl.gov).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSG.2013.2269541

[1] A preliminary version of this paper appeared in the Proceedings of the Annual Cyber Security and Information Intelligence Research Workshop, 2011 [4].

operation. This need to support polling creates an architectural mismatch between Prime and SCADA. The second challenge we confronted relates to performance: the replication engine must be able to provide low enough latency to preserve the real-time control and monitoring of the SCADA system, while being able to support a high enough throughput so that the system can scale to large deployments.

Our efforts to date have resulted in a prototype implementation of a survivable SCADA system for electricity transmission and distribution, where our Prime-based intrusion-tolerant replication engine is integrated with a real Siemens SCADA product. We developed the prototype system on this product because the results of a compromise of a critical infrastructure system such as the power grid would be particularly disruptive and would have significant adverse effects on today's society, and SCADA plays a vital role in the power grid. However, Prime and the protocols described in this paper are generic and could also be applied to other mission-critical systems (e.g., distributed control systems).

The novel contributions of this paper are as follows. i) We present the design of the first SCADA system in which the SCADA Master application is able to survive a partial compromise. ii) To address the architectural mismatch between SCADA systems and traditional intrusion-tolerant replication systems, we present the first scalable and intrusion-tolerant *logical timeout* protocol (to support the scheduling of polling events) and a *logical channel* protocol (to enable reliable and intrusion-tolerant SCADA communication in a replicated environment). iii) We present performance results demonstrating the suitability of our intrusion-tolerant replication engine for use even in large-scale SCADA deployments.

The remainder of this paper is organized as follows. Section II presents background on SCADA and intrusion-tolerant replication needed to understand the rest of the paper. Section III presents the design of our survivable SCADA architecture, as well as our attack model and assumptions. Section IV discusses the integration challenges we faced and presents the new protocols we invented to overcome them. Section V presents our performance results, and Section VI places our solution in the context of related work. Finally, Section VII concludes the paper.

II. Background

A. Conventional SCADA Systems

SCADA systems are large and complex. In this section we describe the components of a SCADA system most relevant to the work in this paper. These components include:

- One or more **Remote Terminal Units (RTUs)**, which communicate with, and aggregate data from, local sensors in the field (e.g., within an electricity distribution substation). Some larger systems can have several thousand RTUs.
- A **SCADA Master**, which periodically polls the RTUs by sending messages over a wide-area network. The SCADA Master maintains a real-time database containing the current state of each RTU. It can also send supervisory control commands to the RTUs. For fault tolerance, many SCADA systems use a Primary/Hot Standby (HSB) configuration, in which two similar but slightly different copies of the SCADA Master application run in parallel. Although both the Primary and the HSB receive incoming events, the Primary is responsible for controlling the system, and the output of the HSB is suppressed. The HSB monitors the Primary and performs a "take-over" operation to assume control if it believes the Primary has succumbed to a benign fault.
- One or more **Human Machine Interface (HMI)** workstations, which periodically query the SCADA Master so that the state of the system (e.g., the power grid) can be graphically displayed for a human operator.

B. Intrusion-Tolerant State Machine Replication

An intrusion-tolerant protocol [12] assumes that some of the participants may be *Byzantine* [13] and act in an arbitrary manner (e.g., because they are compromised and under the control of a malicious attacker). The protocol is designed to operate correctly regardless of how the Byzantine participants behave,[2] as long as no more than a threshold fraction of the participants (typically f out of $3f + 1$) is Byzantine [13]. Intrusion-tolerant protocols often use *proactive recovery* techniques [7], where participants are periodically rejuvenated to a clean state. This allows the system to survive more than f Byzantine failures over the life of the system, as long as no more than f are Byzantine at the same time.

In this paper we make use of an intrusion-tolerant replication system, where replication is achieved via the *state machine approach* [14], [15]. In this approach, the several application *replicas* begin in the same initial state, and they cooperate to agree on the order in which to execute any event (i.e., message or timeout) that might change the state of the application. The state transition caused by executing an event is assumed to be deterministic. Therefore, by executing events according to the agreed upon order, the replicas proceed through the same sequence of states.

It is important to note that although the replicas in a state machine replication system must be functionally equivalent, they are permitted to have different implementations, provided they all adhere to the same abstract protocol specification [16]. Indeed, the effectiveness of the state machine approach to replication depends upon using replicas that are unlikely to suffer correlated vulnerabilities, which can be achieved by using replicas with diverse implementations. Diversity can be introduced at various levels, including at the operating system (OS) level [17] and at the application level.

At the operating system level, one may introduce diversity by running each replica on a different OS (or on as many as are available). Garcia, *et al.* [17] studied over 15 years of known OS vulnerabilities and found that there exist sets of operating systems that exhibit sufficient diversity to avoid suffering common vulnerabilities. While this approach may be a viable option in some deployments, in others it may result in excessive management complexity or may simply not be possible (i.e., if an application is tied to a specific OS). Within a single OS deployment, address space layout randomization (ASLR) [18] can be used

[2]Usually subject to certain cryptographic assumptions.

to generate diversity. ASLR is used by many modern operating systems (e.g., OpenBSD, Linux, Solaris, Microsoft Windows, Mac OS X). It places the sections of a process' address space at random offsets. This mitigates certain types of attacks that rely on being able to predict addresses, because the addresses are likely to differ at each replica.

Traditionally, achieving diversity at the application level has required expensive techniques such as N-version programming [19]. However, newer approaches can automatically create software diversity during compilation [20] or (if source code is unavailable) through binary re-writing [21]. Such approaches require no additional development effort and have been demonstrated to have minimal performance impact. Encouragingly, the compiler-based approach of [20] has been used to diversify an entire Linux operating system. Our system also introduces diversity by deploying each replica with its own private key. An attacker that compromises a replica can cause it to send messages that appear legitimate but have invalid content only if the attacker can compromise the replica's private key. For this reason, in a real deployment it may also be prudent to protect the cryptographic keys using tamper-proof hardware.

As noted in Section I, we selected the Prime intrusion-tolerant replication protocol [5], [6] for our survivable SCADA system. Prime requires $3f + 1$ replicas to tolerate f Byzantine faults and was the first protocol to guarantee both correct operation and good performance in executions in which up to f of the replicas actually exhibit Byzantine behavior. Prime uses an elected leader to coordinate the ordering of events. We selected Prime because its service properties make it a particularly good fit for real-time applications, such as a SCADA Master. Specifically, Prime bounds the amount of delay that can be introduced by Byzantine replicas: assuming enough correct (i.e., non-Byzantine) replicas can communicate with one another in a timely manner, Prime ensures that any event submitted to the system will be ordered by the correct replicas within a bounded delay δ, where δ is a function of the network round-trip times (and their variance) among the correct replicas. To achieve this property, Prime runs a distributed monitoring protocol, whereby the replicas constantly monitor the performance of the current leader and quickly elect a new leader if they detect that the current one is performing too slowly.

III. Survivable SCADA: System Architecture

A. Motivation and High-Level Design

We believe the highest value asset in a SCADA system is the SCADA Master, because its compromise can have serious consequences for the control and monitoring of the entire system. Therefore, our focus in this paper is on improving the robustness of the SCADA Master by making it survivable.

In our survivable SCADA architecture, instead of running a Primary and a Hot Standby, we run $3f + 1$ peer replicas of the (primary) SCADA Master application, where f is the maximum number of replicas that may be Byzantine. Fig. 1 depicts a minimal configuration of the system, which runs four SCADA Master replicas (i.e., $f = 1$). Each replica links with a local

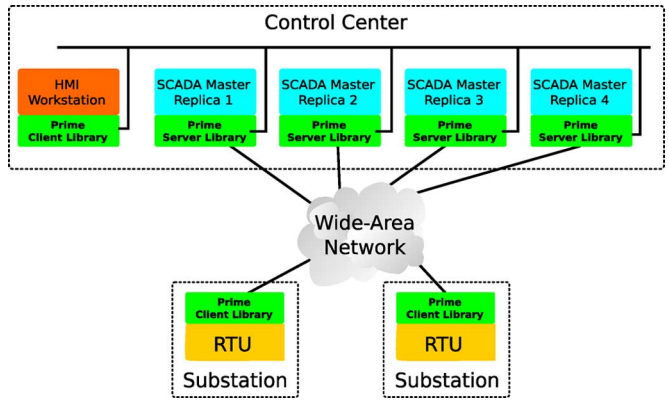

Fig. 1. A survivable SCADA system capable of tolerating the compromise of one SCADA Master replica.

copy of the *Prime Server Library*, the intrusion-tolerant replication engine that delivers to each replica the same events in the same order.

The collection of SCADA Master replicas forms a *logical SCADA Master* that behaves correctly even if f replicas are Byzantine; that is, upon processing an event, the logical SCADA Master makes the same state transition as an unreplicated, uncompromised SCADA Master application would make given the same event. Moreover, Prime's service properties bound the degree to which the Byzantine replicas can slow down the performance of the logical SCADA Master compared to a correct, unreplicated SCADA Master.

In order to interact with the replicated SCADA Master, the HMI and the RTUs each link with the *Prime Client Library*. This library provides functions for i) sending messages to multiple replicas and ii) voting on the messages that arrive from the replicas to determine the correct content.[3] In some deployments one may not have access to the RTU source code in order to link it with the Prime Client Library. We addressed this by implementing an RTU proxy, which communicates with the RTU using its native protocol and uses the Prime Client Library to interact with the SCADA Master replicas. This "bump-in-the-wire" solution may also be suitable for legacy RTUs with limited computational power or memory.

Although our solution significantly improves the robustness of the SCADA Master and its communication with RTUs, we emphasize that achieving system-wide survivability requires taking a systematic approach to security at all levels and in all parts of the system. Intrusion-tolerant replication is complementary to more traditional host, network, and perimeter security technologies and should be deployed alongside them rather than being seen as a replacement for them.

In previous work we discussed how virtualization can be used to reduce the hardware cost of replication to that of a conventional SCADA system: four SCADA Master replicas can be run on only two physical machines (the number required for a Primary/Hot Standby deployment), while still maintaining the ability to survive the crash of either machine. We refer the interested reader to [4] for details.

[3]Since at most f replicas may be Byzantine, a client can act on a message when it receives $f + 1$ copies of the message from different replicas, indicating that at least one correct replica sent a message with the given content.

B. Attack Model and Assumptions

As described above, the logical SCADA Master is implemented by a set of $3f+1$ replicas, f of which may be Byzantine. Byzantine replicas may send invalid or conflicting messages to other replicas or clients, drop or delay messages, or otherwise attempt to disrupt the system. Digital signatures provide message integrity, authentication, and non-repudiation for all Prime messages.

We assume that Byzantine replicas cannot disrupt the communication between correct replicas; this can be achieved through network isolation techniques [11] or by using tamper-proof network cards capable of rate-limiting outgoing traffic (e.g., [22]). Such solutions insulate the correct replicas from denial-of-service attacks launched from within the control center. Mitigating external denial-of-service attacks is a difficult problem for which production systems typically rely on commercial solutions. Note that denial-of-service attacks may, for their duration, affect the performance and monitoring capability of the system but do not affect the consistency of the replicas and, therefore, the correctness of the supervisory control commands issued by the logical SCADA Master.

As in a conventional SCADA system, the ability of the logical SCADA Master to monitor effectively the physical assets of the system relies on the RTUs to report legitimate values. In the current version of our system, a compromised RTU may report incorrect values that will be replicated consistently by the SCADA Master replicas. This impacts the SCADA Master's ability to monitor the substation containing the compromised RTU but is unlikely to affect the monitoring of other substations. Note that although our focus is on making the SCADA Master survivable, intrusion-tolerant replication could also be used at the substation level to form survivable RTUs. In a real deployment, a utility would need to evaluate the costs and benefits of such an approach to determine the feasibility of using replication at this level.

IV. Survivable SCADA: Challenges and Solutions

Conventional SCADA systems are *server driven*: the SCADA Master (the server) periodically sends poll requests to the RTUs (the clients), and the RTUs respond with their current status.[4] The periodic sending of poll requests is triggered by the expiration of *timeout events* at the SCADA Master. In contrast, existing intrusion-tolerant replication systems implicitly assume that the state machine of the server application being replicated is *client driven*: the application processes a client request as input, sends a reply to the requesting client as output, and then processes the next client request.

In this section we describe how this seemingly small difference has large implications on the functionality required from the replication engine in our survivable SCADA system. Indeed, addressing this architectural mismatch required the invention of several new protocols.

A. Scalable Intrusion-Tolerant Synchronization

1) Motivation: When polling an RTU, a SCADA Master takes action based on the passage of time: the expiration of a local timeout triggers the SCADA Master to send a poll request to the RTU. However, absent perfectly synchronized clocks, the passage of time will be observed in a non-deterministic way at the different SCADA Master replicas in our survivable SCADA system. As a result, if the replicas were to make state transitions based on their local clock values, they could become inconsistent with one another. Therefore, in our survivable SCADA system, the SCADA Master replicas use a *logical timeout protocol* to agree on the *logical time* at which a time-based action (such as generating a poll request) should be taken. This protocol must be intrusion-tolerant so that Byzantine replicas cannot disrupt the agreement or trigger the expiration of spurious timeouts. Moreover, since large SCADA systems may contain thousands of RTUs, each of which may be polled individually, the protocol must scale with the number of different logical timeouts being agreed upon.

Existing techniques for time-based synchronization in a Byzantine environment [5] are intrusion tolerant but not scalable, requiring a number of messages proportional to the number of logical timeouts set by the application. In [5], each replica sends a "vote" message each time its local clock indicates a logical timeout should expire (i.e., once per timeout), and the replicas act on a given logical timeout when they agree that $f+1$ replicas have sent corresponding votes.

To provide a solution that scales to large SCADA deployments, we developed a new protocol that only requires a constant number of messages to be exchanged, independent of the number of logical timeouts requested by the application. Our protocol makes no assumptions about the relative speeds of the replicas' local clocks and prevents Byzantine replicas from arbitrarily advancing or delaying the logical time at which a logical timeout expires. As explained below, our protocol achieves scalability at the cost of a (potentially) slightly lower timer resolution compared to [5].

2) Protocol Details: The Prime Server Library provides an API call enabling an application to *set* a logical timeout, (t, d), where d is the number of seconds that should pass before t expires. For example, to poll an RTU a SCADA Master replica might set a logical timeout, t, with a duration, d, of 1 second. Approximately 1 second later, the replica will receive from the Prime Server Library a notification that t has expired. Upon receiving this notification (which is delivered to all replicas at the same logical time), each replica generates an identical poll request and sends it to the RTU.

Observe that since the replicas set each logical timeout at the same logical time, they can consistently map each logical timeout to a unique sequence number, where the ith timeout set is mapped to sequence number i. Although logical timeouts are set in sequence number order, they may *expire* out of order, because they can be set with arbitrary (non-negative) durations.

Each replica r periodically broadcasts to the other replicas a SYNC message of the form $\langle$SYNC, *localClock, lastTimeoutSet*, $r\rangle$, where *localClock* is r's current local clock value and *lastTimeoutSet* is the sequence number of the last logical timeout that r has set. The replicas use Prime to order the SYNC messages. Thus, each replica executes an (identical) ordered stream, S, of SYNC messages. Prime guarantees that if replica r sends SYNC message S_1 before sending SYNC message S_2, then S_1

[4]Some SCADA communication follows the traditional client-server pattern, such as the request/reply protocol between an HMI and the SCADA Master.

appears before S_2 in S; SYNC messages from different replicas may be interleaved in S. Note that SYNC messages are sent periodically and are the only types of messages sent during the protocol. Thus, the protocol has a constant message complexity.

As explained below, the replicas agree upon the logical time at which each logical timeout should expire by using a deterministic, online algorithm that examines the ordered stream of SYNC messages, $S = S_1, S_2, \ldots$, one message at a time, as each new message S_i is ordered by Prime. The examination of the SYNC message most recently ordered results in the expiration of a (potentially empty) set of logical timeouts. The algorithm proceeds in three steps.

Step 1: Identifying candidates for expiration. Let $M = \langle \text{SYNC}, c_r, i, r \rangle$ be the current SYNC message being examined by some replica s. From this message, s learns that M was originated by replica r at time c_r (i.e., r's local clock value) and that the last timeout set by r had sequence number i. Since logical timeouts are set in sequence number order, M also implies that r has set all timeouts 1 through i.

At replica s, we say that a logical timeout with sequence number j becomes a *candidate for expiration with respect to* r the first time that s examines a SYNC message implying that r has set a timeout with sequence number j. In general, the examination of SYNC message M by s may cause several logical timeouts to become candidates for expiration with respect to r; this stems from the fact that r generates SYNC messages periodically rather than each time it sets a logical timeout. As a concrete example, if s is currently examining a message M_i from r with $lastTimeoutSet = 5$, while the last message that s examined from r had $lastTimeoutSet = 3$, then the logical timeouts with sequence numbers 4 and 5 would become candidates for expiration when s examines M_i. Each replica maintains N *candidate lists* (N being the number of replicas), numbered 1 through N, where list r contains an entry for each logical timeout that has become a candidate for expiration with respect to r.

Step 2: Computing when each candidate should expire. Each candidate timeout t is stored along with its *local expiration time with respect to r*. This value is computed as $c_r + d$, where c_r is the local clock value contained in the SYNC message that caused t to become a candidate and d is the duration of t as requested by the application. Intuitively, c_r represents the "best guess" of replica s for when r set timeout t, so s believes t should expire when r's local clock reads $c_r + d$ (i.e., d seconds later). Note that since r only sends SYNC messages periodically, c_r may be up to one period later than the actual time at which r set t. This error is the price paid by our protocol to achieve constant message complexity.

Step 3: Triggering the expiration of candidate timeouts. At replica s, we say that a candidate timeout t is *triggered with respect to replica r* when s examines a SYNC message from r indicating that r's local clock has reached the local expiration time associated with t in candidate list r. Observe that a logical timeout that became a candidate upon examinination of a SYNC message from r will typically not be triggered until a later SYNC message from r is examined; this is due to the fact that examining new SYNC messages from r is the only way in which s updates its estimate of r's current local clock value. When a logical timeout has been triggered with respect to $f + 1$ different replicas, the timeout is said to *expire* and is delivered to the application.

3) Discussion: We make several observations about our logical timeout protocol. First, because the examination of ordered SYNC messages is deterministic, each logical timeout (t, d) expires at the same logical time at all correct replicas. Second, Byzantine replicas cannot cause t to expire before d seconds have elapsed on the local clock of at least one correct replica; thus, Byzantine replicas cannot cause t to expire "too soon." This property holds because t does not expire until it is triggered with respect to $f + 1$ replicas, at least one of which is correct. Third, Byzantine replicas cannot delay the expiration of t: t becomes a candidate for expiration and becomes triggered with respect to a correct replica r based solely on the SYNC messages ordered from r, which cannot be influenced or delayed by the Byzantine replicas.

The "clock resolution" of our logical timeout protocol is determined by two configurable parameters: i) the rate at which SYNC messages are generated (since this dictates how quickly a replica's local clock value can be observed to advance), and ii) the rate at which SYNC messages can be ordered by Prime. The latter is determined by the network delay between replicas (which, on a LAN, should be small) and the rates at which certain periodic messages are sent in Prime. The resolution of the timeout protocol is therefore limited by the slower of rates i) and ii). As explained in Section V, our current implementation achieves a resolution of approximately 17 ms, with the limiting factor being the rate at which Prime orders SYNC messages.

B. Intrusion-Tolerant Reliable Channels

Many SCADA systems make use of a reliable transport protocol, such as TCP, to pass messages between the SCADA Master and the RTUs. By contrast, existing intrusion-tolerant replication systems tend to use UDP and implement their own reliability.[5] In such systems, the replication engine passes application messages between clients and the server replicas. Unfortunately, since existing intrusion-tolerant replication systems implicitly assume that the application is client driven, they provide only limited support for reliable communication between a client and the server replicas. Most use a transaction-based protocol, where the client retransmits its request if it does not receive a response within a timeout period. This approach makes additional implicit assumptions, namely that the server application will always generate a response that can be used by the client as an acknowledgement, and that this acknowledgement message will be sent to the requesting client.

In our survivable SCADA system, messages may be originated by both the SCADA Master replicas and by clients. Moreover, the execution of a client message (e.g., a poll response) by the replicas may cause them to send a reply to an entirely different entity (e.g., the HMI workstation) or not to send a reply at all. Therefore, the system needs a more flexible approach to achieving reliability than the simple client-driven scheme just described.

[5]As noted in [7], TCP is poorly suited to systems with potentially Byzantine receivers because, by failing to send acknowledgements, the Byzantine receivers can require correct replicas to buffer an unbounded number of messages.

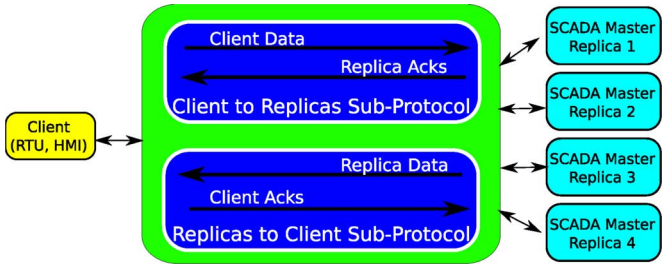

Fig. 2. Intrusion-tolerant reliable channel abstraction.

To address this issue, we developed two protocols that together implement the abstraction of an *intrusion-tolerant reliable channel* between clients and the SCADA Master replicas. Each protocol handles a different communication direction (see Fig. 2). The two endpoints of a channel are asymmetric: one is a client (RTU or HMI) and the other is the set of SCADA Master replicas. Using two unidirectional protocols allows us to take advantage of this asymmetry to optimize performance in each direction. The SCADA Master replicas may communicate with many clients, using a separate channel for each client. Each client is an endpoint of exactly one channel.

Applications interact with a channel using an API similar to that of a TCP socket. The API provides calls for establishing a connection, sending and receiving a message into and from the channel, and closing a connection. Despite the similarity of our API to the socket API, the fact that one end of our channels is replicated has several practical implications. First, when a client application sends a message m into a channel, the channel implementation actually sends m to multiple replicas $(f + 1)$ to ensure its delivery (since some replicas may be Byzantine). Second, although it is a useful abstraction to imagine that a logical SCADA Master application sends a single message m to a client, the sending of m by the logical SCADA Master actually requires several physical messages to be sent, since multiple replicas $(2f + 1)$) introduce m into the channel at the same logical time. Byzantine replicas may also introduce arbitrary messages into the channel at any time.

We refer to messages introduced into a channel by correct replicas or correct clients as *legitimate*; all other messages are *spurious*. The correctness requirements of our channel abstraction, which we now state, define the delivery properties of legitimate and spurious messages.

Let C be a communication channel established between the SCADA Master replicas and a correct (non-Byzantine) client. We say that C is an "intrusion-tolerant reliable channel" if, even when up to f of the SCADA Master replicas are Byzantine, it i) delivers legitimate messages without modification or unnecessary delay; ii) does not deliver spurious messages; and iii) prevents either endpoint from causing the other to consume excessive computational or networking resources. Achieving intrusion tolerance is challenging, because Byzantine replicas may attempt to insert spurious messages into the channel, delay or prevent the delivery of certain legitimate messages, attempt to deliver messages out of order, or otherwise attempt to disrupt the protocol.

Let C' be a communication channel established between the SCADA Master replicas and a Byzantine client. C' makes no delivery or timing guarantees for messages sent from the replicas to the client. Messages sent from the client (which, by definition, are spurious) to the replicas may or may not be delivered, but if any correct replica delivers a message, then they all do. The channel also prevents the Byzantine client from consuming excessive resources at the replicas.

The following sub-sections provide a high-level overview of the operation of each reliable channel sub-protocol.

Client-to-Replicas Sub-Protocol: To introduce a data message into the channel, a client assigns it a sequence number and sends it to a set of $f + 1$ replicas. Since at most f replicas are Byzantine, this ensures that the message is received by at least one correct replica. Each client message carries a digital signature, which prevents Byzantine replicas from modifying its content. Upon receiving a data message, a replica places it into its *local window*. A replica introduces a message for ordering (via Prime) when all messages with lower sequence numbers have been placed in its window. Note that the same client message may be (legitimately) introduced for ordering multiple times (by different replicas), and that Byzantine replicas may introduce messages for ordering out of sequence number order. The replica's channel implementation overcomes these issues and ensures that the client's messages are delivered exactly once, in sequence number order.

Each replica sends cumulative acknowledgements containing the sequence number of the last client data message it has executed. Since the replicas execute data messages at the same logical time, all correct replicas construct identical acknowledgements. The client can slide its window forward (and thus send more messages) when at least $f + 1$ distinct replicas have acknowledged executing the data message at the front of the window. This prevents Byzantine replicas from causing the client to prematurely slide its window. Note that although the replicas send identical acknowledgements, they send negative acknowledgements individually (i.e., based on which data messages they have locally received). This separation enables faster packet recovery in the face of loss, because a replica need not wait until an out-of-order message is executed before requesting a retransmission.

Replicas-to-Client Sub-Protocol: Since the SCADA Master replicas construct data messages at the same logical time, each correct replica introduces identical data messages into the channel, in the same order. Outgoing messages are assigned a sequence number and sent to the client. A client's channel implementation delivers a data message to the client application when the client receives $(f + 1)$ copies of the message (from distinct replicas) and when the channel has delivered all messages with lower sequence numbers.

Clients send cumulative acknowledgements containing the sequence number of the last data message they have delivered. A client also sends negative acknowledgements for those messages it knows to be missing. For efficiency, the client indicates, for each missing sequence number, from which replicas it has already received a copy of the message. Such replicas do not need to retransmit their copies.

The SCADA Master replicas explicitly avoid introducing for ordering (via Prime) client acknowledgements that they receive from the network. This significantly reduces the computational

overhead of the protocol, because it avoids the several rounds of message exchange and the corresponding cryptographic operations associated with ordering. However, an important implication of this design decision is that different replicas may process client acknowledgements at different logical times. As a result, replicas may disagree on which messages the client has received so far, and they may slide their windows asynchronously with respect to one another. To ensure that the replicas still proceed through the same sequence of application states despite the asynchrony that may occur within their channel implementations, the replicas *do* use Prime to order the limited number of key events that might cause the behavior of the application to change. These events include i) connection establishment messages, ii) connection termination messages, and iii) messages indicating that a given replica believes a connection should be closed due to a timeout or because the client is not reading fast enough. Therefore, the replicas always agree on whether the logical state of a channel is open or closed, so they still behave like deterministic state machines at the application level.

V. Performance Evaluation

We have integrated our Prime-based intrusion-tolerant replication engine with a real SCADA Master product for electricity distribution, and we have developed a proxy to integrate this survivable SCADA Master with an RTU. Before integrating our engine with this product, we benchmarked the engine in both fault-free and under-attack scenarios to verify that its throughput and latency could meet the performance requirements of a SCADA system. We also evaluated the performance of our logical timeout protocol to determine whether it could scale to large deployments. Benchmarking the engine in this way enabled us to assess its performance in isolation from the effects of any particular SCADA product or deployment. This section presents the results of our benchmarks.

A. Testbed and Network Setup

We used a cluster of four Dell Precision T3500 servers. The machines had 64-bit, 6-core, 3.47 GHz Intel Xeon processors, with 12 GB RAM and hyper-threading enabled. The machines were connected on a local-area network via a Netgear ProSafe 8-port Gigabit switch. All machines ran 64-bit Fedora 12 Linux. 1024-bit RSA signatures [23] provided authentication and non-repudiation. Each machine can compute an RSA signature in 0.389 ms (2570/sec) and can verify an RSA signature in 0.02 ms (49,683/sec).

For our benchmarks, we implemented a simplified SCADA Master and a simplified RTU. The SCADA Master links with the Prime Server Library and the RTU links with the Prime Client Library (see Fig. 1). The SCADA Master can be configured to be server driven (i.e., to poll one or more RTUs, driven by the expiration of logical timeouts) or client driven (i.e., to receive RTU state updates and send replies). In the experiments described below, we ran two SCADA Master replicas on each of two machines (for a total of four replicas), and the remaining machines ran RTU processes.

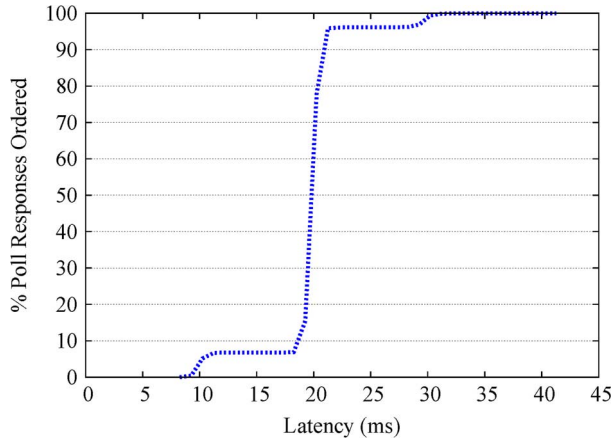

Fig. 3. Poll operation latency, cumulative distribution function.

B. Polling Scenario

Since the main operation in a SCADA system is the polling of RTUs by the SCADA Master, we first evaluated the performance of our engine in a **polling scenario**. We ran four replicas of the SCADA Master, and we ran 1000 RTU processes. The replicas polled each of the 1000 RTUs individually, at a rate of once per second. The time at which each RTU was initially polled was selected uniformly at random over a 1 second interval. Poll requests were 100 bytes long. Upon delivering a poll request, an RTU responded by sending a 100-byte poll reply. When the SCADA Master replicas delivered an ordered poll reply, they re-scheduled a logical timeout for 1 second in the future to poll the associated RTU again. At steady state, replicas set (approximately) 1000 logical timeouts per second, sent 1000 poll requests per second, and received 1000 poll replies per second.

Polling Latency: First, we measured the latency of each poll operation, as measured by SCADA Master replica 1 during an 8-minute run. The latency of a poll operation is computed as the time between the replica sending the poll request to a given RTU and executing the ordered poll reply from that RTU. Since the test was performed on a LAN with sub-millisecond link delay, the measured latency was dominated by the time required for the Prime Server Library to order (and subsequently deliver) the incoming poll reply. This enabled us to measure the amount of latency added by Prime to a typical polling roundtrip. In a real deployment, the SCADA Master replicas would be separated from the RTUs by a wide-area network, so the actual polling latency reported here would be scaled up by the network roundtrip time.

Fig. 3 shows a cumulative distribution function of the polling latencies measured during the run. The y-value of a point represents the percentage of poll operations whose latency was less than or equal to the x-value of the point. For example, the figure shows that about 96 percent of poll operations had a latency less than or equal to approximately 22 ms, and all poll operations had a latency of less than 43 ms. Our discussions with SCADA system architects suggest that, given the supervisory nature of SCADA, this latency is sufficiently low to be suitable for real deployments (in fact, even a latency added by Prime that was twice as high would likely be low enough).

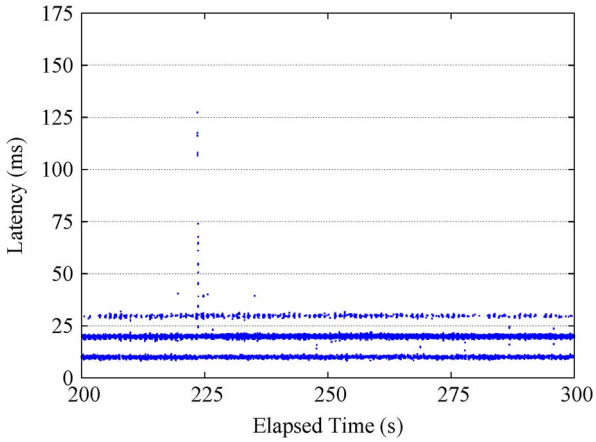

Fig. 4. Polling operation latency, under-attack scenario. At time 220 s, server replica 1 (the coordinator of the replication protocol) began delaying its outgoing messages by 100 ms. The other replicas quickly detected the attack so that subsequent operations were not affected.

Polling Under Attack: To demonstrate that Prime can mitigate performance attacks effectively, we re-ran the polling experiment, but this time we instrumented the coordinator of Prime so that it would begin delaying its outgoing packets by 100 ms after the system had been running for 220 seconds. Fig. 4 shows the latency of the poll operations initiated between time 200 and 300 seconds, as measured at SCADA Master replica 2. Each point represents the latency of a single poll operation. Before the attack, all poll operations had a latency of less than 30 ms, with most falling into two bands at 10 ms and 20 ms. The bands reflect the batching period of 10 ms used for certain messages within Prime. At time 220 s, there is a momentary spike in latency when the attack is triggered; poll operations initiated at this time were delayed by as much as 130 ms. The spike contains at most one operation for each RTU. That the spike is so "skinny" implies that the other replicas quickly detected the attack and reconfigured Prime to mitigate the attack (by electing a new coordinator) so that subsequent operations were not affected.

C. Scalability Scenario

To better understand the scalability of our replication engine, we ran a **scalability scenario** in which we measured Prime's request ordering latency at different throughputs. To generate load, we configured the SCADA Master to be client driven: the RTU submitted to the SCADA Master replicas a 100-byte request to be ordered, and then the replicas responded with a 100-byte reply. Upon receiving a reply, an RTU submitted a new request. We ran between 1 and 30 RTUs, each of which had up to 40 outstanding requests at a time.

In our first, baseline scalability test, each RTU submitted each of its requests to $(f+1)$ SCADA Master replicas. Since at most f replicas may be Byzantine, this ensures that the RTU's message is received by a correct replica the first time it is sent. The trade-off is that in fault-free executions (like the one tested), each message is actually ordered $f+1$ times (twice, in this case), reducing the maximum throughput that can be achieved. The throughput numbers that we report apply to the number of *unique* requests ordered per second.

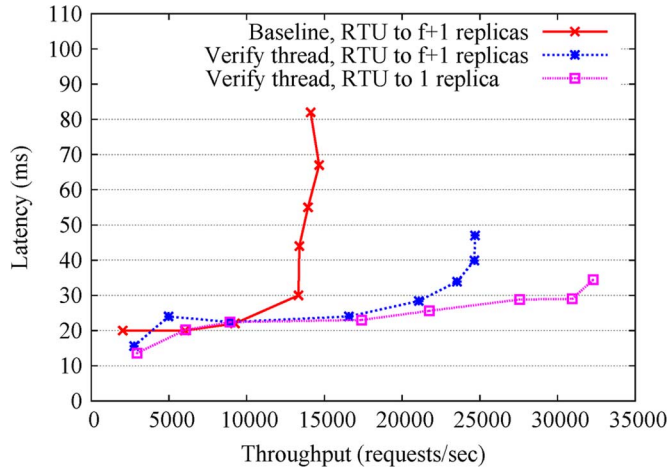

Fig. 5. Request latency vs. replication engine throughput.

Fig. 5 shows the average latency of a request vs. the throughput achieved by our replication engine. Latency was measured at the RTU and computed as the time between submitting a request for ordering and receiving the corresponding reply, averaged across all requests during the run. Throughput was measured at the SCADA Master replicas as the number of requests ordered per second. The performance of our baseline test is shown in the line labeled "Baseline, RTU to $f+1$ replicas" in Fig. 5. Requests experienced a latency of about 25 ms when 12 000 requests per second were ordered, and they experienced a latency of about 30 ms when 13 500 requests per second were ordered. For loads beyond this point, latency dramatically increased because the system was saturated and at its peak throughput.

In running our baseline scalability test, we observed that the throughput of the system was CPU bound, and that the performance bottleneck was the computation required to verify the RSA signatures contained in all Prime messages. Therefore, we re-engineered our engine to use a separate thread for the verification of digital signatures, allowing the implementation to better exploit multiple CPU cores. As seen in the middle plot in Fig. 5, using a verification thread significantly increased the peak throughput of our engine, resulting in requests experiencing a latency of about 27 ms when 20 000 requests per second were ordered and about 35 ms when 24 000 requests per second were ordered.

In the two tests just described, the RTU submitted its requests to $(f+1)$ replicas. A different strategy would be for the RTU to initially submit its request to only one replica, and if it does not receive a reply within a timeout period, it submits to a different replica. This strategy can result in higher peak throughput in fault-free executions, but in under-attack scenarios it can cause latency to be increased by the duration of the RTU's timeout. To assess the fault-free performance impact of this approach, we configured each RTU to submit its requests to a randomly-selected replica. As seen in Fig. 5, this modification increased the scalability of our engine even further, enabling it to order 30 000 requests per second with a latency just under 30 ms. We comment that although this optimistic strategy may be suitable for some SCADA systems, others may be more sensitive to delay, and thus which strategy to prefer is deployment-specific.

TABLE I
LOGICAL TIMEOUT ACCURACY, SINGLE TIMEOUT

Target Delay (ms)	Minimum Delay (ms)	Maximum Delay (ms)	Average Delay (ms)	Average Δ (ms)	Standard Dev. (ms)
0	8.8	26.5	16.8	16.8	4.7
10	19.3	33.1	30.0	20.0	1.4
100	109.7	122.4	119.5	19.5	3.3
500	512.1	524.5	514.7	14.7	3.1
980	993.6	1000.4	997.0	17.0	0.8
1000	1015.6	1020.7	1017.1	17.1	0.8

The above results suggest that the performance of our intrusion-tolerant replication engine will be sufficient for most SCADA systems, even large-scale systems with several thousand RTUs being polled approximately once per second.

D. Logical Timeout Performance

Accuracy: To measure the accuracy of our logical timeout protocol, we configured the SCADA Master application so that it set logical timeouts at various periods; upon delivering a logical timeout, each application replica re-scheduled a new one with the same duration. As shown in Table I, we ran the application in this scenario with several different target periods (Table I, column 1). In each run, the replicas set timeouts at the given period for a five minute duration. For each logical timeout, we measured the time between replica 1 setting the timeout and delivering the event indicating that the timeout expired. Columns 2, 3, and 4 of Table I report the minimum, maximum, and average delays measured by replica 1, respectively. Column 5 reports the average Δ, defined as the actual delay experienced minus the target delay, and Column 6 reports the standard deviation.

Table I shows that the average Δ for all measured target delays is between 15 and 20 ms, indicating that the application consistently experienced a delay 15–20 ms higher than it requested. As discussed in Section IV-A, this "error" reflects the clock resolution of the logical timeout protocol and is caused by the time required for Prime to reach agreement on SYNC messages. Since the Δ values are fairly consistent, the results suggest that an application should take the error into account when specifying its polling period. For example, Table I shows that an application wishing to poll at one second intervals should specify a polling period of 980 ms to compensate for the error. The 0 ms row in Table I shows the clock resolution directly when logical timeouts are set sequentially (i.e., the next one is started only after the current one expires). The data show that Prime can deliver roughly 60 sequential timeout events per second, reflecting an agreement time of roughly 17 ms.

Scalability: To evaluate the scalability of our logical timeout protocol, we measured the average Δ observed by the application when increasing numbers of periodic logical timeouts are set. We repeated this experiment for four different target periods: 10 ms, 100 ms, 500 ms, and 1 s.

Fig. 6 shows the intuitive result that the number of timeouts that can be set before the system reaches saturation (and the Δ values spike) increases with the target period. For example, the replicas can reach agreement on roughly 20 000 500 ms timeouts with a Δ of about 27 ms, and they can reach agreement on roughly 30 000 1 s timeouts with a Δ of 28 ms. Recall from Table I that the Δ values for 500 ms and 1 s timeouts when only one timeout was being set at a time were 14.7 and 17.1 ms, respectively. Thus, in the 1 s case, the number of timeouts being set increased by a factor of 30 000 while the average Δ increased by less than a factor of 2, reflecting the protocol's scalability.

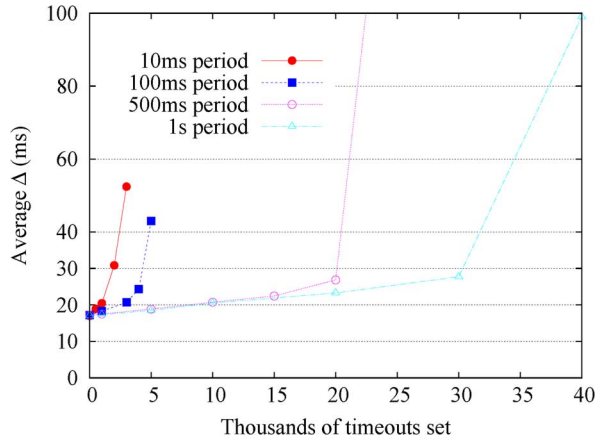

Fig. 6. Logical timeout scalability. Δ is the difference between the actual delay observed by the application and the target delay (timeout period).

VI. RELATED WORK

The number of reported cyber attacks, especially insider attacks, continues to rise; McAfee reported more than 90 million unique pieces of malware in its database in its Q2 2012 threat report [24], up from 70 million just one year earlier. Therefore, SCADA systems have begun deploying standard IT security technologies to harden their defenses. Some representative technologies include firewalls [25] to police incoming traffic; intrusion-detection systems [26] to monitor network and system events to log and report suspicious behavior; and application whitelisting [27] to ensure that only known, trusted executables can run. Although these technologies significantly enhance the security of today's SCADA systems, they focus on preventing attacks and do not protect the SCADA application if an attack compromises part of the system.

The field of *intrusion tolerance* [12] represents a different way of thinking about security and availability. An intrusion-tolerant protocol assumes that some of the protocol participants may be *Byzantine* [13] and act in an arbitrary manner. Over the last decade, using intrusion-tolerant protocols to achieve consistent global state (e.g., [6]–[11]) has been shown to be an effective technique for building highly available systems able to withstand partial compromises. Such protocols are known as *intrusion-tolerant replication protocols*. While earlier protocols guaranteed correctness (i.e., replica consistency) in the face of partial compromise, more recent protocols [6], [11], [28] also guarantee minimal performance degradation whilst under attack and hence meet our definition of survivability. Importantly, these recent protocols can also scale to support thousands of clients.

A different approach to applying intrusion tolerance techniques to critical infrastructure systems was presented by Bessani *et al.* [29]. Rather than integrating intrusion-tolerant replication within the SCADA system itself, one creates an intrusion-tolerant "firewall" (called a CRUTIAL Information Switch) that sits on the perimeter of the network and ensures

that only messages which adhere to policy are admitted into the system, even if some of the replicas implementing the firewall are compromised. Such an approach has the benefit that it does not require any changes to, or integration with, the SCADA Master. However, its effectiveness requires the policy to be specified and implemented correctly, and (unlike our approach) it does not protect the SCADA Master from attacks launched from within the system's security perimeter. Another important distinction is that CRUTIAL relies on trusted hardware components, which are assumed to be unable to be compromised. In contrast, our survivable SCADA system assumes that any machine may be compromised.

VII. Conclusion

In addition to the conventional challenges to availability, such as hardware crashes, power failures, and network partitions, SCADA providers must also anticipate the consequences of cyber attacks. Whereas conventional enterprise security technologies have sought to build increasingly sophisticated perimeter defenses, in this research we sought to answer whether it is possible to build a SCADA system that is able to operate correctly and with good performance even if a cyber attack was successful at evading these conventional defenses.

As the compromise of the highest value asset, the SCADA Master, can have potentially disastrous consequences, our work has focused on protecting this entity via intrusion-tolerant replication. In effect, intrusion tolerance allows the SCADA Master application to act as its own firewall, thus providing protection in the event of a security breach.

This paper reports on our experience designing and evaluating the first survivable SCADA system. We described the unique requirements imposed by the SCADA architecture and gave an overview of several new techniques facilitating the integration of intrusion-tolerant replication and SCADA. Our experimental results illustrate that our replication engine performs sufficiently well to meet the needs of even large-scale SCADA systems containing thousands of RTUs.

References

[1] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Primary-backup protocols: Lower bounds and optimal implementations," in *Proc. 3rd IFIP Conf. Dependable Comput. Critical Appl.*, 1992, pp. 187–198.

[2] E. Byres, "Next generation cyber attacks target oil and gas SCADA," *Pipeline Gas J.*, vol. 239, no. 2, 2012.

[3] D. S. Wall, "Organization security and the insider threat: Malicious, negligent, and well-meaning insiders," 2011, Symantec.

[4] J. Kirsch, S. Goose, Y. Amir, and P. Skare, "Toward survivable SCADA," in *Proc. Annu. Cyber Security Inf. Intell. Res. Workshop (CSIIRW11)*, Oct. 2011.

[5] J. Kirsch, "Intrusion-tolerant replication under attack," Ph.D. dissertation, Johns Hopkins University, Baltimore, MD, USA, 2010.

[6] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 4, pp. 564–577, 2011.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[8] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault-tolerant services," in *Proc. 19th ACM Symp. Oper. Syst. Principles*, 2003, pp. 253–267.

[9] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Trans. Dependable Secure Comput.*, vol. 3, no. 3, pp. 202–215, 2006.

[10] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 1, pp. 80–93, 2010.

[11] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proc. 6th USENIX Symp. Netw. Syst. Design Implementation*, 2009, pp. 153–168.

[12] P. E. Veríssimo, N. F. Neves, and M. P. Correia, R. Lemos, C. Gacek, and A. Romanovsky, Eds., "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*, 2003, vol. 2677, Lecture Notes in Computer Science.

[13] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[16] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," in *Proc. 18th ACM Symp. Operating Systems Principles (SOSP'01)*, Banff, AB, Canada, 2001, pp. 15–28.

[17] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality," in *Proc. 41st IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN'11)*, 2011, pp. 383–394.

[18] PaX Team, PaX Address Space Layout Randomization [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[19] A. Avizeinis, "The N-Version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1491–1501, Dec. 1985.

[20] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proc. New Security Paradigms Workshop (NSPW 2010)*, Concord, MA, USA, Sep. 2010.

[21] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. 2012 ACM Conf. Comp. Commun. Security (CCS'12)*, 2012, pp. 157–168.

[22] *Secure Computing SnapGear User Manual, Revision 3.1.4* Aug. 2006, Secure Computing.

[23] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[24] Z. Bu, T. Dirro, P. Greve, Y. Lin, D. Marcus, F. Paget, V. Pogulievsky, C. Schmugar, J. Shah, D. Sommer, P. Szor, and A. Wosotowsky, "McAfee threats report: Second quarter 2012," 2012.

[25] R. Oppliger, "Internet security: Firewalls and beyond," *Commun. ACM*, vol. 40, no. 5, pp. 94–102, 1997.

[26] D. E. Denning, "An intrusion detection model," in *Proc. 7th IEEE Symp. Security and Privacy*, May 1986, pp. 119–131.

[27] J. V. Harrison, "Enhancing network security by preventing user-initiated malware execution," in *Proc. Int. Conf. Inf. Technol.: Coding and Computing (ITCC'05)—Volume II*, Washington, DC, USA, pp. 597–602, IEEE Computer Society.

[28] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proc. 28th IEEE Int. Symp. Reliable Distrib. Syst.*, 2009, pp. 135–144.

[29] A. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Verissimo, "The CRUTIAL way of critical infrastructure protection," *IEEE Security Privacy*, vol. 6, no. 6, pp. 44–51, Nov.–Dec. 2008.

Jonathan Kirsch received the B.Sc. degree from Yale University, New Haven, CT, USA, in 2004 and the M.S.E. degree from Johns Hopkins University, Baltimore, MD, USA, in 2007. He received a Ph.D. degree in computer science from Johns Hopkins University in 2010.

He is currently a Research Scientist at the Siemens Technology to Business Center, Berkeley, CA, USA. His research interests include fault-tolerant replication and survivable systems.

Dr. Kirsch has served on the program committee for the International Symposium on Stabilization, Safety, and Security of Distributed Systems, as well as the Cyber Security and Information Intelligence Research Workshop. He is an active reviewer for several journals, including IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Computers, and ACM *Transactions on Computer Systems*.

Stuart Goose received the B.Sc. and Ph.D. degrees in computer science from the University of Southampton, U.K., in 1993 and 1997, respectively.

He held a Postdoctoral position at the University of Southampton. He then joined Siemens Corporate Research Inc., Princeton, NJ, USA, holding various positions in the Multimedia Technology Department where he led a research group exploring and applying various aspects of Internet, mobility, multimedia, speech, and audio technologies. His current position is Director of Venture Technology at Siemens Technology-To-Business Center in Berkeley, CA, USA. He scouts for disruptive technologies from universities and startups, runs projects to validate the technical and business merit of technologies, and, if successful, the technologies are transferred to the relevant product lines within Siemens.

Dr. Goose serves as program committee member and reviewer for IEEE International Conference on Distributed Multimedia Systems and IEEE International Conference Multimedia Expo.

Yair Amir received B.S. and M.S. degrees from the Technion, Israel Institute of Technology, in 1985 and 1990, respectively, and a Ph.D. degree from the Hebrew University of Jerusalem, Israel, in 1995.

He has served as Professor of Computer Science at Johns Hopkins University. Baltimore, MD, USA, since 1995. Prior to his Ph.D., he gained extensive experience building C3I systems. He is a creator of the Spread and Secure Spread group communication toolkits, the Backhand and Wackamole clustering projects, the Spines overlay network messaging system, and the SMesh wireless mesh network.

Dr. Amir has been a member of various program committees including the IEEE International Conference on Distributed Computing Systems, the ACM Conference on Principles of Distributed Computing, and the IEEE/IFIP International Conference on Dependable Systems and Networks. He currently serves as an Associate Editor for the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING. He co-founded Spread Concepts LLC (2000) and LTN Global Communications Inc (2008), and is a member of the ACM and the IEEE Computer Society.

Dong Wei (S'00–M'04) received his B.S. degree in electrical engineering from Tsinghua University, Beijing, China. He received his M.S. and Ph.D. degrees from New Jersey Institute of Technology, Newark, NJ, USA, both in electrical engineering. is a research scientist at Siemens Corporation, Corporate Technology.

He has worked at Siemens for more than 10 years. He has worked on factory automation systems, PLC, motion control, human-machine interface, drive system, and industrial communication networks for more than 10 years. He has more than 20 publications, including book chapters and journal papers.

Dr. Wei works as Principal Investigator for several government-funded research projects. He is also an active reviewer of IEEE TRANSACTIONS ON SMART GRID, IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON SYSTEM, MAN, AND CYBERNETICS, *Computer in Industry*, *Journal of the Network and Systems Management*, etc.

Paul Skare is the Chief Cyber Security Program Manager in Electricity Infrastructure at Pacific Northwest National Laboratory, Richland, WA (PNNL). Programs that he manages include PNNL's work on the Department of Energy's Cybersecurity for Energy Delivery Systems (CEDS) and the Cybersecurity Risk Information Sharing Program (CRISP). Previously he worked for Northern States Power for 4 years, and Siemens Energy for 26 years, including roles in power applications, R&D manager for SCADA, Product Lifecycle Manager for EMS and substation automation products, and most recently he was the Director of Cyber Security. He has a patent published on cybersecurity for control systems. He is the Convenor of Working Group 19 in IEC TC 57 and a member of WG 13 & 15 and former WG 7. He is a member of the IEEE PES and worked in IEEE P1689, P1711, and P2030. He has twice testified to the U.S. Congress on cyber security for control systems. Paul has been active in the NERC CSSWG, has been in numerous NERC working groups including Hydra and the GridEx cyber security exercises, and has been active in the DHS ICSJWG and CyberStorm III programs.