COCA: A Secure Distributed On-line Certification Authority

LIDONG ZHOU, FRED B. SCHNEIDER, and ROBBERT VAN RENESSE
Cornell University

COCA is a fault-tolerant and secure on-line certification authority that has been built and deployed both in a local area network and in the Internet. Extremely weak assumptions characterize environments in which COCA's protocols execute correctly: no assumption is made about execution speed and message delivery delays; channels are expected to exhibit only intermittent reliability; and with $3t + 1$ COCA servers up to t may be faulty or compromised. COCA is the first system to integrate a Byzantine quorum system (used to achieve availability) with proactive recovery (used to defend against mobile adversaries which attack, compromise, and control one replica for a limited period of time before moving on to another). In addition to tackling problems associated with combining fault-tolerance and security, new proactive recovery protocols had to be developed. Experimental results give a quantitative evaluation for the cost and effectiveness of the protocols.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*client/server*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*authentication*; *cryptographic controls*; E.3 [**Data**]: Data Encryption—*public key cryptosystems*

General Terms: Security, Reliability, Design, Performance, Measurement

Additional Key Words and Phrases: Certification Authority, Public Key Infrastructure, Byzantine Quorum Systems, Threshold Cryptography, Proactive Secret Sharing, Denial of Service

1. INTRODUCTION

In a public key infrastructure, a *certificate* [Kornfelder 1978] specifies a binding between a name and a public key or other attributes. Over time, public keys and attributes can change—a private key might be compromised, leading to selection of a new public key, for example. The old binding and any certificate that specifies that

This work is supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.
Authors' address: Department of Computer Science, Cornell University, Ithaca, NY 14853-7501.

binding then become *invalid.* A *certification authority* (CA) attests to the validity of bindings by issuing digitally signed certificates that specify these bindings and by providing a means for clients to check the validity of certificates. With an *on-line* CA, principals can check the validity of certificates just before using them. COCA (Cornell On-line Certification Authority), the subject of this paper, is such an on-line CA.

COCA employs replication to achieve availability and employs proactive recovery with threshold cryptography for digitally signing certificates in a way that defends against *mobile adversaries* [Ostrovsky and Yung 1991] which attack, compromise, and control one replica for a limited period of time before moving on to another. In that, the system is not novel. What distinguishes COCA is its qualitatively weaker assumptions about communication links and execution timing. Many denial of service attacks succeed by invalidating stronger communication and execution-timing assumptions; in making weaker assumptions, COCA is less vulnerable to these attacks.

New proactive recovery protocols had to be developed for execution in this relatively unconstrained and more realistic environment. Moreover, because implementing agreement is problematic in the absence of execution-timing assumptions [Fischer et al. 1985], COCA employs a Byzantine quorum system [Malkhi and Reiter 1998a] (rather than the state machine approach [Lamport 1978]) for managing replicated state. In so doing, COCA is the first to tackle the problems associated with integrating threshold cryptography and Byzantine quorum systems. Thus, beyond its intrinsic utility for public key infrastructures, COCA has pedagogical value as a vehicle for understanding how to combine mechanisms for supporting fault-tolerance and security properties.

Besides its weak assumptions, a variety of traditional means for combating denial of service attacks are used by COCA: (i) processing only those requests that satisfy authorization checks, (ii) grouping requests into classes and multiplexing resources so that demands from one class cannot impact processing of requests from another, as well as (iii) caching results of expensive cryptographic operations. And while resource-clogging denial of service attacks certainly remain possible, experiments demonstrate that launching a successful attack against COCA is harder with these mechanisms in place. In fact, simulated denial of service attacks have allowed us to measure the effectiveness of the various means COCA employs to resist denial of service attacks, so the work reported herein contributes much-needed experimental data on the performance of traditional denial of service defenses.

The paper is organized as follows. Section 2 discusses assumptions about the environment in which COCA operates and describes the services COCA provides. Protocols to coordinate COCA servers are the subject of Section 3. Section 4 elaborates on the mechanisms COCA incorporates to defend against denial of service attacks. Performance data for COCA deployments both in a local area network and in the Internet are summarized in Section 5, followed by a discussion of related work in Section 6. Section 7 contains concluding remarks.

2. SYSTEM MODEL AND SERVICES SUPPORTED

COCA is implemented by a set of n servers, each running on a separate processor in a network. We intend COCA for use in an environment like the Internet. Thus, COCA tolerates failures and defends against malicious attacks, subject to the following assumptions:

Servers: Servers are either *correct* or *compromised*, where a compromised server might stop executing, deviate arbitrarily from its specified protocols (i.e., Byzantine failure), and/or disclose information stored locally. System execution is viewed in terms of protocol-defined periods called *windows of vulnerability*; terms "correct" and "compromised" are relative to those periods. Specifically, a server is deemed correct in a window of vulnerability if and only if that server is not compromised throughout that period. We assume:

- —At most t of the n COCA servers are ever compromised during each window of vulnerability, where $3t + 1 \leq n$ holds.
- —Clients and servers can digitally sign messages using a scheme that is existentially unforgeable under adaptively chosen message attacks.
- —Various cryptographic algorithms (e.g., public key cryptography and threshold cryptography) COCA employs are secure.

Fair Links: A *fair link* is a communication channel that does not necessarily deliver all messages sent, but if a process sends infinitely many messages to a single destination then infinitely many of those messages are correctly delivered. In addition, messages in transit may be disclosed to or altered by adversaries.

The communications network is assumed to provide (only) fair links. (Without some comparable assumption about the network, an adversary could prevent servers from communicating with each other or with clients.)

Asynchrony: There is no bound on message delivery delay or server execution speed.

These assumptions endow adversaries with considerable power. Adversaries can

- —attack servers, provided fewer than $1/3$ of the servers are compromised within a given window of vulnerability,
- —launch eavesdropping, message insertion, corruption, deletion, reordering, and replay attacks, provided Fair Links is not violated, and
- —conduct denial of service attacks that delay messages or slow servers by arbitrary finite amounts.

2.1 Operations Implemented by COCA

COCA supports one operation (Update) to create, update, and invalidate certificates that specify bindings; a second operation (Query) retrieves certificates specifying those bindings. A client invokes an operation by issuing a *request* and then awaiting a *response*. COCA expects each request to contain a nonce. Responses from COCA are digitally signed using a COCA service key and include the client's request, hence the nonce[1], thereby enabling a client to check whether a given response was produced by COCA for that client's request.

A request is considered *accepted* by COCA once any correct COCA server receives the request or participates in processing the request; and a request is considered *completed* once some correct server has constructed the response. It might, at first, seem more natural to deem a request "completed" only when the client receives a response. However, such a definition would make a client action (receipt of a response) necessary for a request to be considered completed. In the absence of assumptions about clients, it then becomes problematic for COCA to implement

Request Completion: Every request accepted is eventually completed.

However, as will become clear, a correct client making a request will eventually receive a response from COCA.

Each COCA certificate ζ is a digitally signed attestation that specifies a binding between some name cid and some public key or other attributes $pubK$. In addition, each certificate ζ also contains a unique serial number $\sigma(\zeta)$ assigned by COCA. The following semantics of COCA's Update and Query give meaning to the natural ordering on these serial numbers—namely, that a certificate for cid invalidates certificates for cid having lower serial numbers.

Update: Given a certificate ζ for a name cid and given a new binding $pubK'$ for cid, an Update request returns an acknowledgment after COCA has created a certificate ζ' for cid such that ζ' binds $pubK'$ to cid and $\sigma(\zeta) < \sigma(\zeta')$ holds.

Query: Given a name cid, a Query request $\mathcal{Q}$ returns a certificate ζ for cid such that:
(i) ζ was created by some Update request that was accepted before $\mathcal{Q}$ completed.
(ii) For any certificate ζ' for name cid created by an Update request that completed before $\mathcal{Q}$ was accepted, $\sigma(\zeta') \leq \sigma(\zeta)$ holds.

By assuming an initial default binding for every possible name, the operation to create a first binding for a given name can be implemented by Query (to retrieve the certificate for the default binding) followed by Update. And an operation to revoke a certificate for cid is easily built from an Update specifying a new binding for cid.

Update creates and invalidates certificates, so its invocation should probably be restricted to certain clients. Consequently, COCA allows an authorization policy to be defined for Update. In principle, a CA could always process a Query, because Query does not affect any binding. In practice, that policy would create a vulnerability to denial of service attacks, so COCA adopts a more conservative approach discussed in Section 4.

The semantics of Update associates larger serial numbers with newer certificates and, in the absence of concurrent execution, a Query for cid returns the certificate whose serial number is the largest of all certificates for cid. Certificate serial

[1] In the current implementation, requests contain sequence numbers which, along with the client's name, form unique numbers. Therefore, the text of the request itself can serve as the nonce.

numbers are actually consistent only with a *service-centric* causality relation: the transitive closure of relation $\rightarrow$, where $\zeta \rightarrow \zeta'$ holds if and only if ζ' is created by an Update having ζ as input. Two Update requests $\mathcal{U}$ and $\mathcal{U}'$ submitted, for example, by the same client, serially, and where both input the same certificate, are not ordered by the $\rightarrow$ relation. So, our semantics for Update allows $\mathcal{U}$ to create a certificate ζ, $\mathcal{U}'$ to create a certificate ζ', and $\sigma(\zeta') < \sigma(\zeta)$ to hold—consistent with the service-centric causality relation but the opposite of what is required for serial numbers consistent with Lamport's more-useful potential causality relation [Lamport 1978] (because execution of $\mathcal{U}$ is potentially causal for execution of $\mathcal{U}'$).

COCA is forced to employ the service-centric causality relation because COCA has no way to obtain information it can trust about causality involving operations it does not itself implement. Clients would have to provide COCA with that information, and compromised clients might provide bogus information.

Update and Query are not indivisible and (as will become apparent in Section 3) are not easily made so: COCA's Update involves separate actions for the invalidation and for the creation of certificates. In implementing Update, we contemplated either possible ordering for these actions: Execute invalidation first, and there is a period when no certificate is valid; execute invalidation last, and there is a period when multiple certificates are valid.

We wanted Query always to return a certificate, so avoiding periods with no valid certificate for a given name would have meant synchronizing Query with concurrent Update requests. We rejected this because the synchronization creates an execution-time cost and introduces a vulnerability to denial of service attacks—repeated requests by an attacker for one operation could now block requests for another operation. Our solution is to have Update create the new certificate before invalidating the old one, but it too is not without unpleasant consequences. Both of the following cannot now hold.

(i) A certificate for *cid* is valid if and only if it is the certificate for *cid* with largest serial number.

(ii) Query always returns a valid certificate.

COCA clients therefore must accommodate our more-complicated semantics for Query and program their own synchronization.

2.2 Bounding the Window of Vulnerability

The duration of COCA's window of vulnerability cannot be characterized in terms of real time due to our Asynchrony assumption, so its duration is defined in terms of events marking the completion of *proactive recovery protocols* that are executed periodically to:

—reload the code (thereby eliminating Trojan horses),

—reconstitute the state of each COCA server (which might have been corrupted during the previous window of vulnerability), and

—obsolete any confidential information an attacker might have obtained by compromising servers.

Each window of vulnerability at a COCA server begins when that server starts executing the proactive recovery protocols and terminates when that server has again started and finished those protocols. Thus, every execution of the proactive recovery protocols is part of two successive windows of vulnerability. COCA is agnostic about when the proactive recovery protocols start. Currently, each COCA server attempts to run these protocols after a specified interval has elapsed on its local clock but (to avoid denial of service attacks) a server will refuse to participate in the protocols unless enough time has passed on its clock since they last executed.

In theory, using protocol events to delimit the window of vulnerability affords attackers leverage. Denial of service attacks that slow servers and/or increase message delivery delays expand the real-time duration for the window of vulnerability, creating a longer period during which attackers can try to compromise more than t servers. But in practice, we expect assumptions about timing can be made for those portions of the system that have not been compromised.[2] Given such information about correct server execution speeds and message-delivery delays, real-time bounds on the window of vulnerability can be computed.

Limiting the Utility of Compromised Keys.

Server Keys. Each COCA server maintains a private/public key pair, and the public key is known by all COCA servers. These public keys allow servers to authenticate the senders of messages they exchange with other servers.

In the absence of tamper-proof co-processors, server keys must be refreshed as part of proactive recovery. One simple approach has trusted administrators for each server invent and propagate new public keys through secure channels implemented by having an *administrative* public/private key pair. The administrative public key is known to other administrators (and all servers); the administrative private key, kept off-line most of the time as a defense against on-line attacks, is used to sign notification message for the new public server public key. Other rekeying schemes are discussed in [Canetti and Herzberg 1994].

Public keys of COCA servers are not given to COCA clients so that clients need not be informed of changed server keys—attractive in a system with a large number of clients and where server keys are periodically refreshed.

Service Key. There is one service private/public key pair. It is used for signing responses and certificates. All clients and servers know the service public key.

The service private key is held by no COCA server. Instead, different shares of the key are stored on each of the servers, and threshold cryptography [Desmedt and Frankel 1990; 1992; Desmedt 1994; 1998; Frankel and Yung 1998] is used to construct signatures on responses and certificates. To sign a message:

(1) each COCA server generates a *partial signature* from the message and that server's share of the service private key;
(2) some COCA server combines these partial signatures and obtains the signed message.[3]

[2]A server that violates these stronger execution timing assumptions might be considered compromised, for example.
[3]Having a client combine the partial signatures instead of having COCA do it introduces a vulnerability to denial of service attacks. Clients, lacking COCA server public keys, do not have a way to authenticate the origins of messages conveying the partial signatures. Therefore, a client

With $(n, t+1)$ threshold cryptography, $t+1$ or more partial signatures are needed in order to generate a signature. An adversary must therefore compromise $t+1$ servers in order to forge COCA signatures.

Proactive Secret Sharing. A mobile adversary might compromise $t+1$ servers over a period of time and, in so doing, collect the $t+1$ shares of the service private key. Consequently, COCA employs a proactive secret sharing protocol to refresh these shares, periodically generating a new set of shares for the service private key and deleting the old set. New shares cannot be combined with old shares to construct signatures. So periodic execution of this proactive secret sharing protocol ensures that a mobile adversary can forge COCA signatures only by compromising $t+1$ servers in the interval between protocol executions.

The proactive secret sharing protocol that COCA employs makes no synchrony assumptions (which would be incompatible with the Asynchrony assumption of Section 2), unlike prior work (e.g., [Jarecki 1995; Herzberg et al. 1995; Herzberg et al. 1997; Frankel et al. 1997; Frankel et al. 1997]); details are discussed in [Zhou et al. 2002; Zhou 2001]. For the discussion in this paper, it suffices to regard the protocols simply as services that COCA invokes.

Server Code and State Recovery. Part of proactive recovery should include refreshing the states and reloading the code at COCA servers. The state of a COCA server involves a set of certificates. In theory, this state could be refreshed by performing a Query request for each name that could appear in a certificate, but the cost of such an enumeration would be prohibitive. So instead, during proactive recovery, a list with the name and serial number for every valid certificate stored by each COCA server is sent to every other server. Upon receiving this list, a server retrieves any certificates that appear to be missing. Certificates stored by COCA servers are signed (by COCA), so each certificate can be checked to make sure it is not bogus. The certificate serial numbers enable servers to determine which of their certificates have been invalidated (because a certificate for that same name but with a higher serial number exists).

Server code should be reloaded from some read-only media or other trusted source by proactive recovery in order to eliminate Trojan horses installed by attackers during the previous window of vulnerability. This functionality is not currently implemented in our prototype, however, since defending against such attacks is not the focus of our research; see Castro [Castro 2000] for an in-depth discussion of the issues.

There is one non-obvious point of interaction between proactive recovery and request processing. To satisfy Request Completion, an accepted request that has not been completed when a window of vulnerability ends must become an accepted request in the next window of vulnerability. Therefore, such a request must be propagated to other servers as part of proactive recovery. So each correct server, when executing the proactive recovery protocol, resubmits to all servers any request that is then in progress and awaits acknowledgments from at least $t+1$ servers. Some server that is correct in this next window of vulnerability necessarily receives

---

could be bombarded with bogus partial signatures, and only by actually trying to combine these fragments—an expensive enterprise—could the bona fide partial signatures be identified.

that request, and that means this accepted request in the previous window of vulnerability also becomes an accepted request in the new window of vulnerability.

To avoid a spate of new requests from delaying termination of proactive recovery (a potential denial of service attack), COCA servers could ignore such new requests.[4] In those rare cases where a re-started request has not finished before a new proactive recovery is started, COCA could delay proactive recovery until after the processing of re-started requests has been completed. In practice, windows of vulnerability will tend to be long (*viz.* days) relative to the time (5 seconds or less) required for processing a Query or Update request. It is thus extremely unlikely that a request restarted in a subsequent window of vulnerability would not be completed before proactive recovery is again commenced.

3. PROTOCOLS

In COCA, every client request is processed by multiple servers and every certificate is replicated on multiple servers. The replication is managed as a dissemination Byzantine quorum system [Malkhi and Reiter 1998a], which is feasible because we have assumed $3t + 1 \leq n$ holds. So servers are organized by COCA into sets, called *quorums*, satisfying:[5]

Quorum Intersection: The intersection of any two quorums contains at least one correct server.

Quorum Availability: A quorum comprising only correct servers always exists.

And every client request is processed by all correct servers in some quorum.

Detailed protocols for Query and Update appear as an Appendix; in this section, we explain the main ideas behind the design of these protocols. Technical challenges the protocols must address include:

—Because requests are processed by a quorum of servers but not necessarily by all correct COCA servers, different correct servers might process different Update requests. Consequently, different certificates for a given name *cid* are stored by correct servers. Certificate serial numbers provide a solution to the problem of determining which of those certificates is the one to use.

—Because clients do not know COCA server public keys, a client making a request cannot authenticate messages from a COCA server and, therefore, cannot determine whether a quorum of servers has processed that request. The solution is for some COCA servers to become *delegates* for each request. A delegate presides over the processing of a client request and, being a COCA server, can authenticate server messages and assemble the needed partial signatures from other COCA servers. A client request is handled by $t + 1$ delegates to ensure that at least one of these delegates is correct.

—Because communication is done using fair links, retransmission of messages may be necessary.

[4] The time to execute proactive recovery tends to be short, and ignoring (a finite number of) messages is permitted by the Fair Links assumption.

[5] Provided there are $3t+1$ servers and at most t of those servers may be compromised, the quorum system $\{Q : |Q| = 2t + 1\}$ constitutes a dissemination Byzantine quorum system. For simplicity, we assume $n = 3t + 1$ holds; the protocols are easily extended to cases where $n > 3t + 1$ holds.

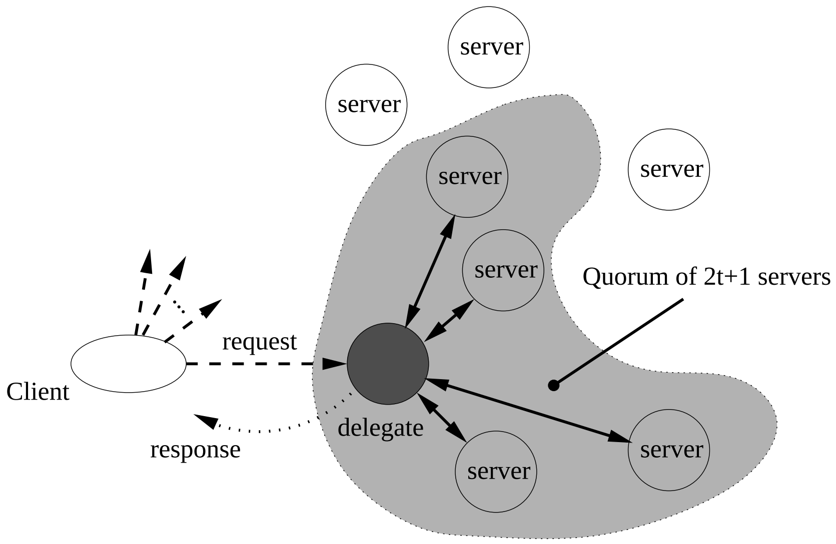

Fig. 1. Overview of client request processing.

Figure 1 gives a high-level view of how COCA operates by depicting one of the $t+1$ delegates and the quorum of servers working with that delegate to handle a client request. The figure shows a client making its request by sending a signed message to $t+1$ COCA servers. Each server that receives this message assumes the role delegate for the request. A delegate engages a quorum of servers to handle the request (by sending that request to all COCA servers) and constructs a response to the request based on the responses received from that quorum. The delegate then causes this response to be signed by the service—this involves running a threshold signature protocol in cooperation with t other servers. Once signed, the response is sent by the delegate to the client. Upon receipt, the client checks that the response is correctly signed by the service and contains the client's original request; if it isn't or if no response has been received within a specified period of time, then the client simply again sends the original request to $t+1$ servers.

Protocol Details

Certificate Serial Numbers. The serial number $\sigma(\zeta)$ for a COCA certificate ζ is implemented as a pair $\langle v(\zeta), h(R_\zeta)\rangle$, where $v(\zeta)$ is a *version number* and $h(R_\zeta)$ is a collision-resistant hash of the Update request $\mathcal{R}_\zeta$ that led to creation of ζ. Version numbers encode the service-centric causality relation as follows.

—The first certificate created to specify a binding for a name *cid* is assigned version number 0.

—A certificate ζ' produced by an Update given certificate ζ is assigned version number $v(\zeta') = v(\zeta) + 1$.

Because different requests have different collision-resistant hashes, certificates created by different requests have different serial numbers. The usual lexicographic ordering on serial numbers yields the total ordering on serial numbers we seek—an ordering consistent with the transitive closure of the $\rightarrow$ relation.

Note that, even with serial numbers on certificates, the same new certificate will be created by COCA if an Update request is re-submitted, and Update requests are thus idempotent. This is because the serial number of a certificate is entirely determined by the arguments in the request that creates the certificate.

Determining a Response for Query. It suffices to consider an abstract description of COCA's Update and Query protocols in order to characterize responses satisfying parts (i) and (ii) in the specification for Query. The actual protocols refine this abstract description.

COCA Update requests are processed by correct servers in some quorum and not necessarily by all correct COCA servers. Consequently, a correct COCA server p can be ignorant of certificates having larger serial numbers than p stores for a name *cid*. Part (ii) in the specification for Query implies that all completed Update requests (hence, all certificates) must be taken into account in determining the response to a Query request $\mathcal{Q}$. Therefore, a quorum of servers must be engaged in processing $\mathcal{Q}$. Responses from a quorum Q of servers is guaranteed if all COCA servers are contacted. Provided each server in Q responds with the certificate (signed by COCA) it stores having the largest serial number among all certificates (for *cid*) known to the server, then the certificate ζ having the largest serial number among the correctly signed certificates received in the responses from Q can serve as the response to $\mathcal{Q}$. That is, ζ will satisfy part (i) and part (ii) in the specification for Query, as we now show.

We first show that any certificate ζ obtained by refining the protocol outlined above satisfies part (i). Part (i) stipulates that a certificate returned for Query is created by an accepted Update; it is satisfied by ζ if each certificate is signed by COCA only after the Update request creating that certificate has been accepted. This is because the $(n, t+1)$ threshold cryptography being employed for digital signatures requires cooperation (collusion) by more than t servers in order to sign a certificate. Given our assumption of at most t compromised servers, we conclude that there are not enough compromised servers to create bogus signed certificates. Therefore, when a certificate is signed, a correct server must have participated in processing the request that created the certificate; the request creating the certificate had to have been accepted.

Part (ii) of the specification for Query requires that, for any Update request $\mathcal{U}$ naming *cid* and completed before $\mathcal{Q}$ is accepted, $\sigma(\zeta') \leq \sigma(\zeta)$ must hold where ζ' is the certificate created by $\mathcal{U}$. This holds for implementations that refine the abstract description given above due to Quorum Intersection, because some correct server p in Q must also be in the quorum that processed $\mathcal{U}$. Let certificate ζ_p be p's response for $\mathcal{Q}$. Server p always chooses the certificate for *cid* with the largest serial number, so $\sigma(\zeta') \leq \sigma(\zeta_p)$ holds. And because ζ is the certificate that has the largest serial number among those from all servers in Q, $\sigma(\zeta_p) \leq \sigma(\zeta)$ holds. Therefore, $\sigma(\zeta') \leq \sigma(\zeta)$ holds.

The Role of Delegates. Every request is processed by all correct servers in some quorum; the client must be notified once that has occurred. Direct notification by servers in the quorum is not possible because clients do not know the public keys for COCA servers and, therefore, have no way to authenticate messages from those servers. So, instead, a COCA server is employed to detect the completion of request processing and then to notify the client, as follows.

A delegate for a request $\mathcal{R}$ is a COCA server that causes $\mathcal{R}$ to be processed by correct COCA servers in some quorum and then sends a response (signed by COCA) back to the initiating client. The processing needed to construct the response depends on the type of request being processed.

—To process a Query request $\mathcal{Q}$ for name *cid*, the delegate obtains certificates from a quorum of servers, picks the certificate ζ having the largest serial number, and uses the threshold signature protocol to produce a signed response containing ζ:
  (1) Delegate forwards $\mathcal{Q}$ to all COCA servers.
  (2) Delegate awaits certificates for *cid* from a quorum of COCA servers.
  (3) Delegate picks the certificate ζ having the largest serial number of those received in step 2.
  (4) Delegate invokes COCA's threshold signature protocol to sign a response containing ζ; that response is sent to the client.

—To process an Update request $\mathcal{U}$ for name *cid*, the delegate constructs the certificate ζ for the given new binding (using the threshold signature protocol to have COCA digitally sign it) and then sends ζ to all COCA servers. A server p replaces the certificate ζ_p^{cid} for *cid* that it stores by ζ if and only if the serial number in ζ is larger than the serial number in ζ_p^{cid}:
  (1) Delegate constructs a new certificate ζ for *cid*, using the threshold signature protocol to sign the certificate.
  (2) Delegate sends ζ to every COCA server.
  (3) Every server, upon receipt, replaces the certificate for *cid* it had been storing if the serial number in ζ is larger. The server then sends an acknowledgment to the delegate.
  (4) Delegate awaits these acknowledgments from a quorum of COCA servers.
  (5) Delegate invokes COCA's threshold signature protocol to sign a response; that response is sent to the client.

Quorum Availability ensures that a quorum of servers are always available, so step 2 in Query and step 4 in Update are guaranteed to terminate. Since at least $t+1$ of COCA's $3t+1$ servers are correct, compromised servers cannot prevent a delegate from using $(n, t+1)$ threshold cryptography in constructing the COCA signature for a certificate or a response. Thus, step 4 in Query and steps 1 and 5 in Update, which involve contacting all COCA servers, cannot be disrupted by compromised servers.

A compromised delegate might not follow the protocol just outlined for processing Query and Update requests. COCA ensures that such behavior does not disrupt the service by enlisting $t+1$ delegates (instead of just one) for each request. At least one of the $t+1$ delegates must be correct, and this delegate can be expected to follow the Query and Update protocols. So, we stipulate that a (correct) client making a request to COCA submits that request to $t+1$ COCA servers; each server then serves as a delegate for processing that request.[6]

With $t+1$ delegates, a client might receive multiple responses to each request and each request might be processed repeatedly by some COCA servers. The duplicate responses are not difficult for clients to deal with—a response is discarded if it is received by a client not waiting for a request to be processed. That each request might be processed repeatedly by some COCA servers is not a problem either, because COCA's Query and Update implementations are idempotent.

But a compromised client might not follow the protocol and thus might not submit its request to $t+1$ delegates. A problem then occurs if the delegates receiving a request $\mathcal{R}$ execute the first step of Query or Update processing and then halt. Correct COCA servers now participated in the processing of $\mathcal{R}$, so (by definition) $\mathcal{R}$ is accepted. Yet no (correct) delegate is responsible for $\mathcal{R}$. Request $\mathcal{R}$ is never completed, and Request Completion is violated.

The defense is straightforward:

—Messages related to the processing of a client request $\mathcal{R}$ contain $\mathcal{R}$.

—Whenever a COCA server receives a message related to processing a client request $\mathcal{R}$, that server becomes a delegate for $\mathcal{R}$ if it is not already serving as one.

The existence of a correct delegate is now guaranteed for every request that is accepted.

Self-Verifying Messages. Compromised delegates might also attempt to produce an incorrect (but correctly signed) response to a client by sending erroneous messages to COCA servers. For example, in processing a Query request, a compromised delegate might construct a response containing a bogus or invalidated certificate and try to get other servers to sign that; in processing an Update request, a compromised delegate might create a fictitious binding and try to get other servers to sign that; or when processing an Update request, a compromised delegate might not disseminate the updated certificate to a quorum (causing the response to a later Query to contain an invalidated certificate).

COCA's defense against erroneous messages from compromised servers is a form of monitoring and detection that we call *self-verifying messages*.[7] A self-verifying message comprises:

—information the sender intends to convey and

—evidence enabling the receiver to verify—without trusting the sender—that the information being conveyed by the message is consistent with some given protocol and also is not a replay.

In COCA, every message a delegate sends on behalf of a request contains a transcript of relevant messages previously sent and received in processing that request (including the original client request). Because messages contained in the transcript are signed by their senders, a compromised delegate cannot forge the transcript.

[6]An optimization discussed in Section 5 makes it possible for clients, in normal circumstances, to submit requests to only a single delegate.

[7]Similar schemes can be found in [Kihlstrom et al. 1997; Castro and Liskov 1999; Baldoni et al. 2000; Doudou et al. 2000].

And, because the members of the quorum participating in the protocol are known to all, the receiver of such a self-verifying message can independently establish whether messages sent by a delegate are consistent with the protocol and the messages received.[8]

Communicating using Fair Links. The Fair Links assumption means that not all messages sent are delivered. To implement reliable communication in this environment, it suffices for a sender to resend each message until a signed acknowledgment is received from the intended recipient. In turn, the recipient returns a signed acknowledgment for every message it receives (including duplicates, since the previous acknowledgments could have been lost). If both the sender and the receiver are correct then (due to Fair Links) this protocol ensures that the receiver eventually receives the message, the sender eventually receives an acknowledgment from the receiver, and the sender exits the protocol.

Each protocol in COCA is structured as a series of multicasts, with information piggybacked on the acknowledgments. A client starts by doing a multicast to $t+1$ delegates; the signed response from a single delegate can be considered the acknowledgment part of that multicast. A delegate then interacts with COCA servers by performing multicasts and awaiting responses from servers. For the threshold signature protocol, $t+1$ correct responses suffice; for retrieving and for updating certificates, responses from a quorum of servers are needed. Thus, with at least $2t+1$ correct servers, COCA's multicasts always terminate due to Quorum Availability since a delegate is now guaranteed to receive enough acknowledgments at every step and, therefore, eventually that delegate will stop retransmitting messages.

4. DEFENSE AGAINST DENIAL OF SERVICE ATTACKS

A large class of successful denial of service attacks work by exploiting an imbalance between the resources an attacker must expend to submit a request and the resources the service must expend to satisfy that request, as has been noted, for example, in [Juels and Brainard 1999; Meadows 1999; 2001]. If making a request is cheap but processing one is not, then attackers have a cost-effective way to disrupt a service—submit bogus requests to saturate server resources. A service, like COCA, where request processing involves expensive cryptographic operations and multiple rounds of communication is especially susceptible to such resource-clogging attacks.

COCA implements three classic defenses to blunt resource-clogging denial of service attacks:

(i) An authorization mechanism identifies requests on which resources should not be expended.

(ii) Requests are grouped into classes, and resources are scheduled in a manner that prevents demands by one class from affecting requests in another class.

(iii) The results of expensive cryptographic operations are cached, and attackers cannot destroy the locality that makes this cache effective.

[8]In [Gong and Syverson 1998], Gong and Syverson introduce the notion of a *fail-stop protocol*, which is a protocol that halts in response to certain attacks. One class of attacks is thus transformed into another, more benign, class. Our self-verifying messages can be seen as an instance of this approach, transforming certain Byzantine failures to more-benign failures.

The details for COCA's realizations of these defenses constitute the bulk of this section.

Note that our Fair Links and Asynchrony assumptions are an important defense against denial of service attacks, too. An attacker stealing network bandwidth or cycles from processors that run COCA servers is not violating assumptions needed for COCA's algorithms to work. Such a "weak assumptions" defense is not without a price, however. Implementing real-time service guarantees on request processing requires a system model with stronger assumptions than we are making. Consequently, COCA can guarantee only that requests it receives are processed eventually. Those who equate availability with real-time guarantees (e.g., [Gligor 1984; Yu and Gligor 1990; Millen 1992; 1995]) would not be satisfied by an eventuality guarantee. But a system whose correctness depends only on "weak assumptions" is not precluded from satisfying real-time guarantees when the environment satisfies stronger assumptions, and COCA does just that.

Finally, COCA employs connectionless protocols for communication with clients and servers, so COCA is not susceptible to connection-depletion attacks such as the well-known TCP SYN flooding attack [Schuba et al. 1997]. But the proactive secret sharing protocol in the current COCA implementation does use SSL (Secure Socket Layer) [Freier et al. 1996] and is, therefore, subject to certain denial of service attacks. This vulnerability could be eliminated by restricting the rate of SSL connection requests, reprogramming the proactive secret sharing protocol, or adopting the mechanisms described in [Juels and Brainard 1999].

4.1 Request-Processing Authorization

Each message received by a COCA server must be signed by the sender. The server rejects messages that

—do not pass certain sanity checks,

—are not correctly signed, or

—are sent by clients or servers that, from messages received in the past, were deemed by this server to be compromised.

An invalid self-verifying message, for example, causes the receiver r to judge the sender s compromised, and the request-processing authorization mechanism at r thereafter will reject messages signed by s (until instructed otherwise, perhaps because s has been repaired).

Verifying a signature is considerably cheaper than executing an Update or Query request (which involve threshold cryptography and multiple rounds of message exchange). But verifying a signature is not free, and an attacker might still attempt to flood COCA with requests that are not correctly signed. Should this vulnerability ever become a concern, we would add a still-cheaper authorization check that requests must pass before signature verification is attempted. Cookies [Karn and Simpson 1997; Oppliger 1999], hash chains [Kent et al. 1996], and puzzles [Juels and Brainard 1999] are examples of such checks.

Of course, any server-based mechanism for authorization will consume some server resources and thus could itself become the target of a resource-clogging attack, albeit an attack that is more expensive to launch by virtue of the additional authorization mechanism. An ultimate solution is authorization mechanisms that also establish the origin of the request being checked, since fear of discovery and reprisal is an effective deterrent.

4.2 Resource Management

Because requests are signed, COCA servers are able to identify the client and/or server associated with each message received. This enables each COCA server to limit the impact that any compromised client or server can have. In particular, each COCA server stores messages it receives in one of a set of *input queues* and employs a scheduler to service those queues. The queues and scheduler limit the fraction of a server's cycles that can be co-opted by a given source of requests.[9] Others have also advocated similar approaches [Gligor 1984; Yu and Gligor 1990; Millen 1992; 1995].

Our COCA prototype has a configurable number of input queues at each server. A round-robin scheduler services these queues. Client requests are stored on one or more queues, and messages from each COCA server are stored on a separate queue associated with that server. Duplicates of an element already present on a queue are never added to that queue. Each server queue has sufficient capacity so replays of messages associated with a request currently being processed cannot cause the queue to overflow (since that would constitute a denial of service vulnerability).

In a production setting, we would expect to employ a more sophisticated scheduler and a rich method for partitioning client requests across multiple queues. Clients might be grouped into classes, with requests from all clients in the same administrative domain stored together on a single queue.

4.3 Caching

Replays of legitimate requests are not rejected by COCA's authorization mechanism. Nor should they be, since Fair Links forces clients to resend each request until enough acknowledgments are received. But attackers now have an inexpensive way to generate requests that will pass COCA's authorization mechanism, and COCA must somehow defend against such replay-based denial of service attacks.

There are actually two ways to redress an imbalance between the cost of making requests and the cost of satisfying them. One is to increase the cost of making a request, and that is what the signature checking in COCA's authorization mechanism does. A second is to decrease the cost of processing a request. COCA also embraces

[9] Clearly, this offers no defense against distributed denial of service attacks [Richtel and Robinson 2000] in which an attacker, masquerading as many different clients, launches attacks from different locations. If the clients involved in such an attack can be detected, then their requests could be isolated using COCA's queues and scheduler, but solving the difficult problem—determining which clients are involved in such an attack—is not helped by this COCA mechanism.

No host-based defense can combat an attack that saturates incoming links. Still, COCA does enable two forms of what might be termed a distributed defense. First, distributed denial of service attacks directed at some region of a network (rather than targeting the COCA service *per se*) can be tolerated when COCA servers have been deployed so widely that a significant number reside outside the region under attack. Second, the proactive recovery protocols in COCA could enable the service to migrate from one set of hosts to another, which then could allow the service to outrun a distributed denial of service attack (provided sufficient bandwidth remains available to execute proactive recovery).

this latter alternative. Each COCA server caches responses to client requests and caches the results of expensive cryptographic operations for requests that are in progress, as also is suggested in [Oppliger 1999; Castro 2000]. Servers use these cached responses instead of recalculating them when processing replays.

The cache for client responses is managed differently than the cache for in-progress cryptographic results. We first discuss the client-response cache. With finite-capacity caches, responses to clients cannot be cached indefinitely. If the server cache is to be effective against replays submitted by clients, we must minimize the chance of such replays causing cache misses (and concomitant costly computation by the server). The solution is to ensure that client replays are forced to exhibit a temporal locality consistent with the information being cached. In particular, by caching COCA's response for each client's most recent request,[10] by restricting clients to making one request at a time, and by having clients associate ascending sequence numbers with their requests, older requests not stored in the cache can be rejected as bogus by COCA's authorization mechanism.

Because requests are processed by only a quorum of COCA servers, a given server's cache of client responses might not be current. A replay request signed by client c to some server s can have a sequence number that is larger than the sequence number for the last response cached at s for c. The larger sequence-numbered request would not be rejected by s and could not be satisfied from the cache—the request would have to be processed. But with quorums comprising $2t+1$ of the $3t+1$ COCA servers, at most t such replays can lead to computation by COCA servers. COCA's implementation further limits susceptibility to these attacks. Whenever a COCA server sends a response to a client, that response is also sent to all other COCA servers. Each server is thus quite likely to have cached the most recent response for every client request.

An attacker not only can replay client requests for denial of service attacks, but can also replay messages that servers exchange. COCA's defense here, too, is a cache. Servers cache results from all expensive operations, such as computing one-way hashes[11] from shares for proactive secret sharing and computing partial signatures for in-progress requests. The cache at each server is sufficiently large to handle the maximum number of requests that all COCA servers could have in-progress at any time. A total of 60K bytes suffices for a cache to support one client request, when COCA certificates do not exceed 1024 bytes (which seems reasonable given observed usage).

COCA limits the number of requests that can be in-progress at any time by having each delegate limit the number of requests it initiates. Of course, a compromised delegate would not respect such a bound. But recall that COCA servers are notified when responses are sent, so a server can estimate the number of concurrent requests that each server (delegate) has in progress. COCA servers can thus ignore messages from servers that initiate too many concurrent requests.

[10]In a system with a million clients, this client cache would be roughly 5 gigabytes because approximately 5K bytes is needed to store a client's last request and COCA's response.

[11]The one-way hash function involves expensive modular exponentiation and is needed to implement verifiable secret sharing [Feldman 1987].

5. PERFORMANCE OF COCA

Our COCA prototype is approximately 35K lines of new C source; it employs threshold and proactive threshold RSA schemes (with 1024-bit RSA keys), constructed using the protocol described in [Zhou 2001] from building blocks given in [Rabin 1998].[12] We implemented the protocols in OpenSSL [OpenSSL Project nd]. COCA certificates have the same syntax as X.509 [CCITT 1988] certificates, with a COCA certificate's serial number embedded in the X.509 serial number.[13]

Much of the cost and complexity of COCA's protocols is concerned with tolerating failures and defending against attacks, even though failures and attacks are infrequent today. We normally expect:

N1: Servers will satisfy stronger assumptions about execution speed.

N2: Messages sent will be delivered in a timely way.

Our COCA prototype is optimized for these normal circumstances. Wherever possible, redundant processing is delayed until there is evidence that assumptions N1 and N2 no longer hold.

In particular, our prototype delays when COCA servers start serving as the additional delegates for client requests already in progress. This reduces the number of delegates when N1 and N2 hold, hence it reduces the cost of request processing in normal circumstances. The refinements to the protocols of Section 3 are:

—A client sends its request only to a single delegate at first. If this delegate does not respond within some timeout period, then the client sends its request to another t delegates, as required by the protocols in Section 3.

—A server that receives a message in connection with processing some client request $\mathcal{R}$ and that is not already serving as a delegate for $\mathcal{R}$ does not become a delegate until some timeout period has elapsed.

—A delegate p sends a response to all COCA servers, in addition to sending the response to the client initiating the request, after the request has been processed. After receiving such a response, a server that is not yet a delegate for this request will not become one in the future; a server that is serving as a delegate aborts that activity.

—A cached response is forwarded to a server q whenever q instructs p to participate in the processing of a request that has already been processed. Upon receiving the forwarded response, q immediately terminates serving as a delegate for that request.

Also, the threshold signature protocol COCA uses is designed to give better performance when N1 and N2 hold.

[12]The protocols [Zhou 2001] we use employ replication of shares and subshares in achieving fault tolerance rather than the backup scheme used in [Rabin 1998].

[13]Although syntactically compatible with X.509 certificates, COCA certificates are not interchangeable with the X.509 certificates in use by public key infrastructures today. First, COCA imposes an interpretation on the serial numbers embedded in certificates—a COCA certificate with a higher serial number invalidates one with a lower serial number for the same client. Second, COCA, because it supports Query, has no need to and therefore does not provide the CRLs (Certification Revocation Lists) usually associated with public key infrastructures that support X.509 certificates.

Table I. Execution Time in a LAN when N1 and N2 hold.

COCA Operation	Mean (*msec*)	Std dev. (*msec*)
Query	629	16.7
Update	1109	9.0
PSS	1990	54.6

Table II. Breakdown of costs for Query, Update, and proactive secret sharing (PSS) in local area network deployment.

	Query	Update	PSS
Partial Signature	64%	73%	
Message Signing	24%	19%	22%
One-Way Function			51%
SSL			10%
Idle	7%	2%	15%
Other	5%	6%	2%

5.1 Local Area Network Deployment

The experiments reported in this subsection all involved a COCA prototype comprising four servers (i.e., $n = 4$ and $t = 1$) communicating using a 100Mbps Ethernet. The servers were Sun E420R Sparc systems running Solaris 2.6, each with four 450 MHz processors. The round-trip delay for a UDP packet between any two servers on this Ethernet is usually under 300 microseconds.

Table I gives times for COCA operations executing in isolation when assumptions N1 and N2 hold. We report the delay for Query, for Update, and for a round of proactive secret sharing.[14] The reported sample means and sample standard deviations are based on 100 executions. All samples were within 5% of the mean.

To better understand the origin of these delays, we report in Table II the (percentage) contribution that can be attributed to certain CPU-intensive cryptographic operations. For Query and Update, we measured the time spent generating partial signatures and signing messages. For proactive secret sharing, we measured the delay associated with the one-way function, with message signing, and with computation involved in establishing an SSL (Secure Socket Layer) connection to transmit confidential information between servers. Notice that improved hardware for performing cryptographic operations could have a considerable impact. Idle time, because servers sometimes wait for one another, is also listed in Table II. Only 2% to 6% of the total execution time is unaccounted. That time is being used for signature verification, message marshaling and un-marshaling, and task management.

To evaluate the effectiveness of the optimizations outlined above for when assumptions N1 and N2 hold, Figure 2 compares performance with and without the optimizations. The results summarize 100 executions; very small sample standard deviations are observed here. The optimizations thus can be seen to be effective.

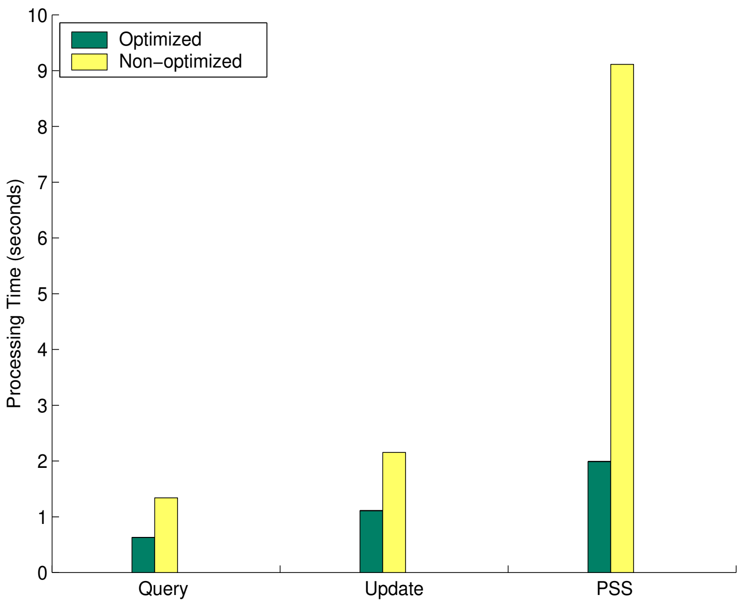

Fig. 2. Effectiveness of optimization in Query, Update, and proactive secret sharing (PSS) when assumptions N1 and N2 hold.

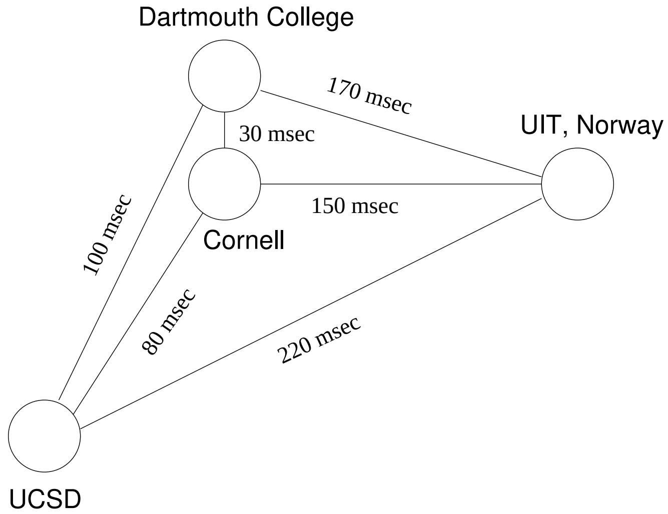

Fig. 3. Deployment of COCA over the Internet with message delays between servers.

5.2 Internet Deployment

Communications delays in the Internet are higher than in a local area network; the variance of these delays is also higher. To understand the extent, if any, this affects

[14]Time spent checking certificates and performing other state recovery at each server is not included in these delays.

Table III. Performance of COCA over the Internet. The averages and sample standard deviations are from 100 repeated executions during a 3-day period.

COCA Operation	**Mean** (*msec*)	**Std dev.** (*msec*)
Query	2270	340
Update	3710	440
PSS	5200	620

Table IV. Breakdown of costs for Query, Update, and proactive secret sharing (PSS) in Internet deployment.

	Query	Update	PSS
Partial Signature	8.0%	8.7%	
Message Signing	3.2%	2.5%	2.6%
One-Way Function			7.8%
SSL			1.6%
Idle	88%	87.7%	87.4%
Other	0.8%	1.1%	0.6%

performance, we deployed four COCA servers as follows.

—University of Tromsø (UIT), Tromsø, Norway. (300 MHz, Pentium II)

—University of California at San Diego (UCSD), San Diego, CA. (266 MHz, Pentium II)

—Cornell University, Ithaca, NY. (550 MHz, Pentium III)

—Dartmouth College, Hanover, NH. (450 MHz, Pentium II)

All ran Linux.[15] Figure 3 depicts the average message delivery delay (measured using ping) between these servers. Delivery delays on the Internet vary considerably [Labovitz et al. 1997] but the values observed during the experiments we report did not differ significantly from those in Figure 3.

Table III gives measurements for the Cornell host in our 4-site Internet deployment. In comparing Table I and Table III, we see the impact of the Internet's longer communication delays (which also lead to longer server idle times). The sample standard deviation is also higher for the Internet deployment, due to higher load variations on servers and due to the higher variance of delivery-delays on the Internet; all samples are located within 25% of the mean. See Table IV for a breakdown of delays (analogous to Table II) for our Internet deployment of COCA.

5.3 COCA Performance and Denial of Service Attacks

Any denial of service attack will ultimately involve attackers (i.e., some combination of compromised clients and/or servers) (i) sending new messages, (ii) replaying old messages, and (iii) delaying message delivery or processing. To evaluate how effective COCA's defenses are against these, we simulated attacks and measured their

[15]Beggars can't be choosers. For making measurements, we would have preferred having the same hardware at every site, though we have no reason to believe that our conclusions are affected by the modest differences in processor speeds. For a production COCA deployment, we would recommend having different hardware and different operating systems at each site so that common-mode vulnerabilities are reduced.

impact. The results of those experiments for our local area network deployment of COCA are discussed in this subsection.

Message-Creation Defense. New messages sent by servers are not nearly as potent in denial of service attacks against COCA as new messages sent by clients. This is because messages from servers are rejected unless they self-verify. So such messages must contain a correctly signed client request as well as correctly signed messages from all servers involved in previous protocol steps—the collusion and compromise of more than t COCA servers is thus required to get past COCA's request-processing authorization mechanism. Moreover, once any message from a given server is found by a COCA server p to be invalid, subsequent messages from that server will be ignored by p, considerably blunting their effectiveness in a denial of service attack to saturate p.

In contrast, a barrage of requests from compromised clients, if correctly signed, cannot be rejected by COCA's request-processing authorization mechanism (unless the identities of these compromised clients is already known by the receiver). The impact of such a barrage should be mitigated by COCA's resource management mechanism, which ensures that messages from a small set of senders do not monopolize server resources. How effective as a defense this mechanism is depends on the exact configuration of COCA's resource management mechanism: the number of input queues, on which input queues various clients are grouped, and the scheduler used in servicing these input queues.

To measure this effectiveness, it suffices to investigate the simple case of two clients. A *compromised client* sends a barrage of new requests to the service at rates we control;[16] a *correct client* sends a request and then awaits a response or a timeout[17]. Of interest is by how much the correct client's requests become delayed due to requests the compromised client sends, since this information can then be used in predicting COCA's behavior when there are more than two clients.

Once a client's request $\mathcal{R}$ is appended to some input queue on a (correct) COCA server, two factors contribute to delay processing $\mathcal{R}$. The first source of delay arises from multiplexing the server as it concurrently processes a number of requests. This number of requests is referred to as the *level of concurrency*. Assuming a modest load from correct clients, the delay due to sharing the processor with other, concurrent requests is not affected by actions an attacker might take and thus is not of interest here; our experiments therefore assume servers process requests to completion serially (*viz.* the level of concurrency is 1). The second source of delay is affected by the compromised client's barrage of new messages—requests in input queues whose processing will precede $\mathcal{R}$. A mechanism to defend against a barrage of client requests must control this source of delay, and it is this delay that we measure.

Our first experiment adjusted the rate of requests from the compromised client while measuring the performance of requests from the correct client. To start, each server was configured to store all client requests on a single input queue. The

[16] Because the compromised client does not await responses before sending additional requests, these experimental results apply directly to the case where a group of compromised clients all share the same input queue on each server.
[17] The timeout is 1 second for Query and 2 seconds for Update.

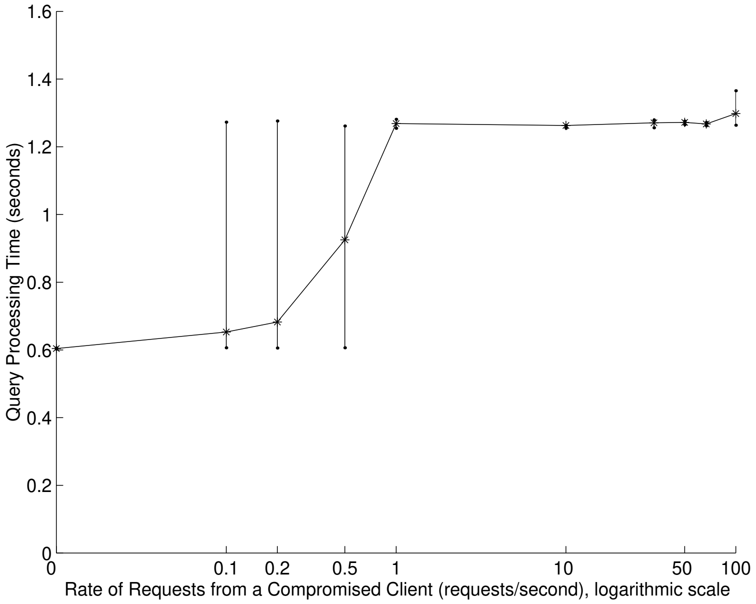

Fig. 4. Performance of Query for a correct client when a compromised client makes requests at varying rates.

capacity of this queue was 10 requests. We found that the correct client would get no service whenever the compromised client sent requests at a rate in excess of 10 requests per second. At 10 requests per second, requests from the compromised client fill the (fixed capacity) input queue virtually all the time—a Query request from the correct client has a 9 in 10 chance of being discarded because it arrives when there is no room in the input queue, and an Update request has half that (due to the 1 and 2 seconds timeout respectively). The denial of service attack is a success.

For the next experiment, each server was configured to have separate queues for the correct client and the compromised client. A round-robin scheduler serviced the two queues. Figures 4 and 5 show performance of Query and Update requests from the correct client for various rates of requests from the compromised client. Every reported data point is the average processing time over 100 experiments; the error bars depict the range for 95% of the samples.

The curves for Query and Update in Figures 4 and 5 comprise two segments. In the first segment, increases in the rate of requests that the compromised client sends cause increased delays for requests from the correct client. This is because as the rate of requests from the compromised client increases, so does the probability that COCA—with its round-robin servicing of input queues—will have to process one of those requests $\mathcal{R}$ before processing a request from the correct client. The processing of $\mathcal{R}$ thus increases the processing time for a request from the correct client. We see in this first segment almost identical wide ranges of samples for each rate measured. The worst case occurs when the request from the correct server

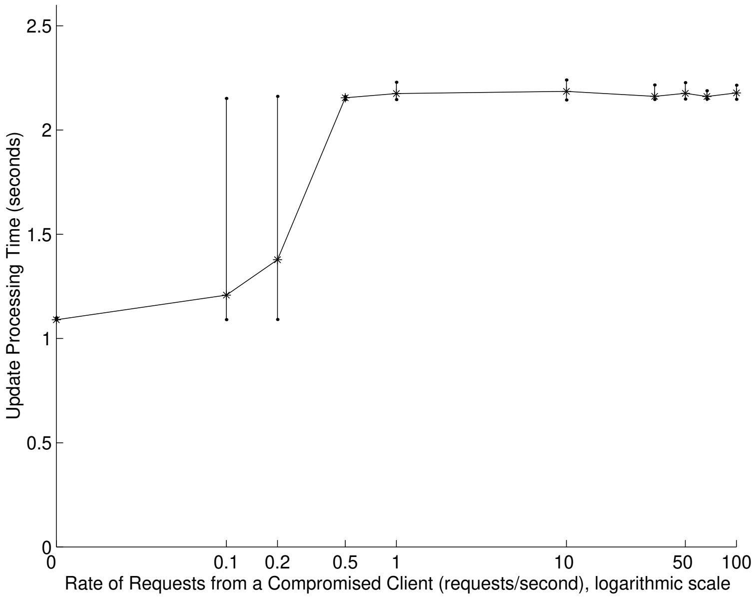

Fig. 5. Performance of Update for a correct client when a compromised client makes requests at varying rates.

arrives just after a request from the compromised client starts to get processed, while the best case occurs when the request from the correct server arrives when no request from the compromised client is being processed. Even though we see the same worst and best case, the means of samples increases as the rate of requests from the compromised client increases, reflecting an increasing probability that the request from the correct client has to wait for the processing of a request from the compromised client.

The second segment of the curves begins once the compromised client is sending requests at approximately the same rate as the normal client (i.e., approximately 1 request per second for Query and 0.5 requests per second for Update). Throughout this second segment, further increases in the request rate from the compromised client do not further degrade the processing of requests from the correct client. This is because requests from the two clients are being processed in alternation, and the delay for requests from the correct client remains at about double what is measured when there is no compromised client. Note that, as the rate of requests from the compromised client increases, more and more of those requests are discarded by servers—the fixed-capacity input queue for the compromised client is full when those requests arrive.

COCA's request-processing authorization mechanism starts saturating at 100 requests per second, due to the time required for a server to perform signature verification on each incoming message. With 100 requests per second, a server has diminished processing capacity to execute protocols for Query and Update. There was thus little point in exploring higher request rates in performance experiments, and we didn't.

In an actual deployment, clients will be partitioned over a set of input queues. But the worst-case performance for this case is easy to bound in light of the above experiments for two clients. Suppose b queues are serving only compromised clients, c queues are serving only correct clients, and d queues are serving both kinds of clients. Requests from compromised clients will starve requests from correct clients that share the same input queue, because the first experiment above established that if the rate of requests to a single input queue from compromised clients exceeds 10 requests per second then requests from correct clients to that input queue are unlikely to succeed. And the second experiment established that COCA's resource-management mechanisms will guarantee that $c/(b+c+d)$ of each server's processing time and other resources are devoted to processing requests on the queues that serve only correct clients.

Message-Replay Defense. COCA employs caching to defend against denial of service attacks involving message replays. We need not consider replays of client requests in our experiments, because their impact is considerably smaller than the impact of processing new requests from a compromised client. Specifically, for new requests, COCA must expend resources in executing the protocol for the operation being requested, but for replays of client requests, processing (by design) involves considerably fewer resources—the request is one that can be rejected because its sequence number is too small, one that can be be satisfied from the server's cache, or one that can be ignored because it is already being processed. Assuming that the requests being replayed are from the same (compromised) client that launches the denial of service attacks in the experiments of Figures 4 and 5, those curves give the bounds we seek on the worst-case performance of COCA when client-request replays form the basis for a denial of service attack.

Replays of messages from servers in COCA are not immobilizing, because relatively expensive cryptographic computations are cached. To validate this, we simulated an attacker replaying server messages at varying rates to all other COCA servers. The message being replayed was designed to cause a defenses-disabled COCA server to compute partial signatures, which takes approximately 200 milliseconds on a 450 MHz Sun E420 Sparc server—a relatively expensive operation and thus particularly effective in a denial of service attack.

We measured the average delay for Query, Update, and proactive secret sharing as a function of the rate of message replay sent by the compromised server. We compared the performance in the case where caching is enabled to that in the case where caching is disabled. This information appears in Figures 6 through 8.

For the case where caching is enabled, the average delay for each operation is largely unaffected as the rate of message replay increases, because caches satisfy most of the computational needs in handling those messages. We witnessed a slight increase in the average delay when the rate of message replay reaches 100 messages per second. This is the point where the request-processing authorization mechanism becomes saturated by incoming messages.

For the case where caching is disabled, each curve consists of two segments. The first segment (which ends at approximately 3 replays per second for Query and Update, and 10 replays per second for PSS) resembles the first segment in the

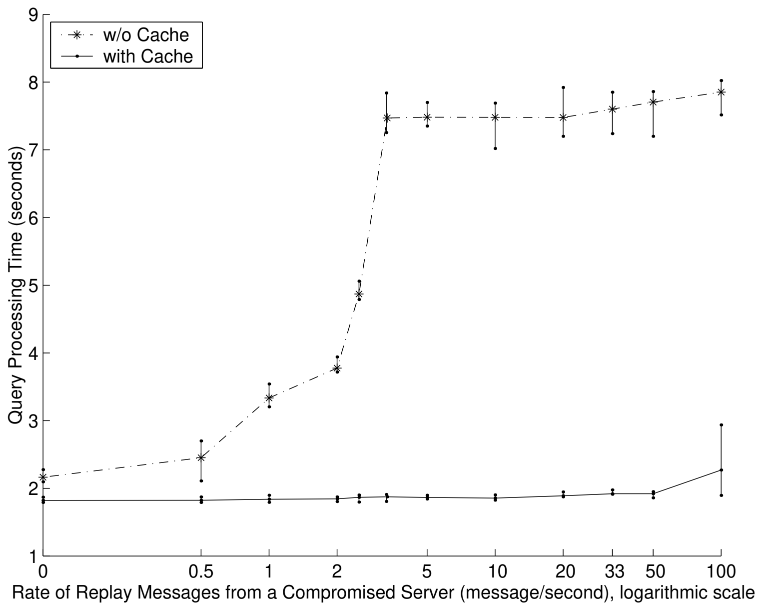

Fig. 6. Performance of Query processing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

curves of Figures 4 and 5, and it reflects the increased use of processing resources by replays to recompute values that were not cached as the replay rate increases. The second segment only gradually increases. Over this range, additional computation is not required (so additional delay is not incurred) since the resource management mechanism bounds the number of attacker messages that are processed.

Even without the compromised server launching the attack (i.e., when the rate of replay messages is 0), the average delay for each operation in the case where caching is enabled is lower than that in the case where caching is disabled. This is because, with one fewer server participating, repeated executions of certain expensive operations is necessary since assumption N1 no longer holds, so correct servers are unable to finish processing in an optimized execution. The switch back to the fault-tolerant version causes repeated executions of certain expensive cryptographic operations, which can be avoided when caching is enabled.

Delivery-Delay Defense. To measure the impact of message transmission and processing delays on the performance of COCA, we instrumented each server so that messages delivered to a client or server could be delayed a specified amount before becoming available for receipt. We investigated both the case where messages sent to one specific server are delayed and the case where messages sent to all servers are delayed.

Figure 9 gives the average time and the interval containing 95% of the samples for COCA to process three operations of interest—Query, Update, and a round of proactive secret sharing—when messages from a single server are delayed. The case where this server is unavailable is also noted as inf on the abscissa.

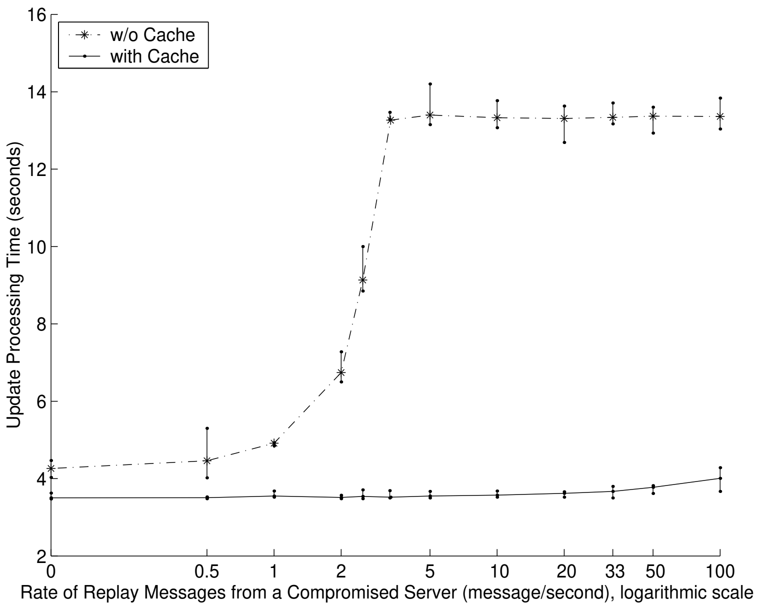

Fig. 7. Performance of Update processing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

As delay increases, the processing time is seen to move through three phases. During the first phase, as server p (say) increases its delay in processing messages, so does the delay for the operation of interest. This occurs because COCA protocols initially assume N1 and N2 hold, and the optimized protocols require participation by p. A delay in messages from p thus slows the protocols.

The second phase is entered after the delay for p causes servers to suspect that assumptions N1 and N2 do not hold. These servers initiate redundant processing, creating additional delegates for in-process operations, for example. Participation by p is no longer required for the operation to terminate; increasing the delay at p does not delay completion of the operation. But p will continue to send messages requiring servers to compute replies. The time that servers devote to generating these replies decreases as the delay for p increases, simply because p sends fewer such messages when the delay is greater. Servers thus have more cycles to devote to generating replies for servers other than p; these are the replies needed in order for the protocols to terminate. So, the increasing delay for p frees server resources to speed the termination of the protocol, and average processing time decreases in this second phase.[18]

The third phase—a plateau in response time—is reached when the delay for p is

[18]We see that the decrease in processing time is more significant in the case of proactive secret sharing than in the cases of Query and Update. In the case of proactive secret sharing, processing messages from server p involves some new (therefore not cached) expensive cryptographic operations, while, in the other two cases, expensive cryptographic operations can be avoided due to caching.

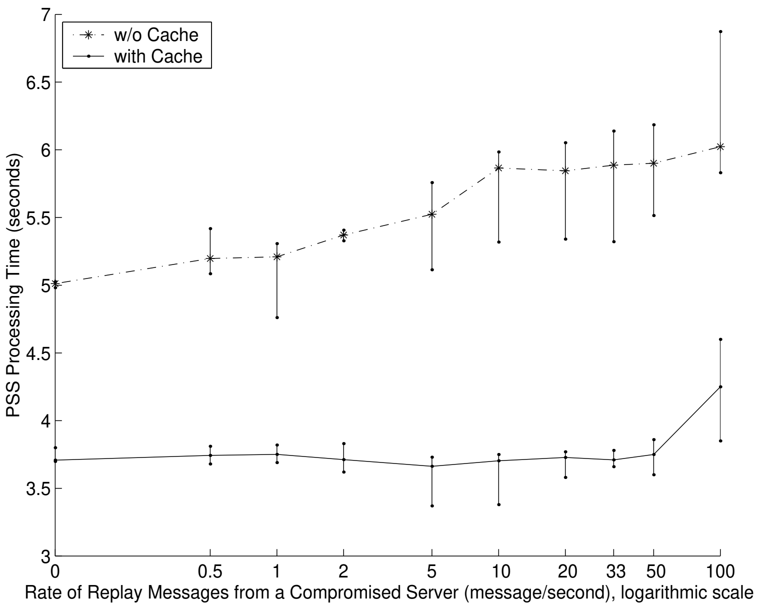

Fig. 8. Performance of proactive secret sharing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

sufficiently high so that it imposes little load on other servers.

Figure 10 gives average measured delay and the interval containing 95% of the samples when message delay increases at all servers. Observe that the execution time increases linearly with the increase of message delay. The curves are consistent with how the protocols operate: processing a Query involves 6 message delays, processing an Update involves 8 message delays, and a round of proactive secret sharing involves 6 message delays.

6. RELATED WORK

Systems. A fault-tolerant authentication substrate [Reiter et al. 1994] for supporting secure groups in the Horus system appears to be the first use of threshold cryptography along with replication for implementing a CA. That led to the design and implementation of Ω [Reiter et al. 1996], a stand-alone general-purpose CA having more ambitious functionality, performance, and robustness goals. Unlike COCA, none of this early work was intended to resist denial of service attacks or mobile adversaries. Ω does provide clients with key escrow operations, something that COCA does not currently support.[19]

Ω was built using middleware (called Rampart [Reiter 1995; 1996]) that implements process groups in an asynchronous distributed system where compromised processors can exhibit arbitrary behavior. Rampart manages groups of replicas

[19]The same threshold decryption and blinding [Chaum 1983; 1985; 1988] that Ω uses for supporting this additional functionality would allow COCA to support these features too.

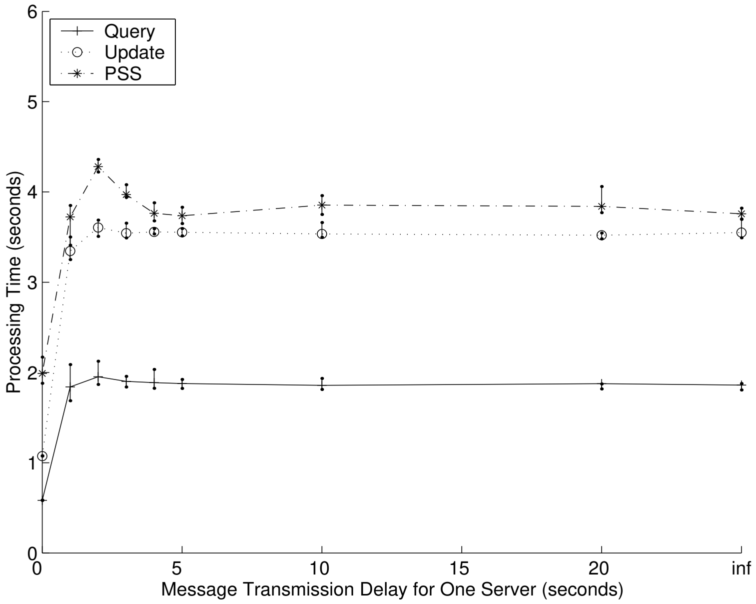

Fig. 9. Performance of COCA vs. message delay for one server. Message delay of **inf** indicates the case where this one server is unavailable.

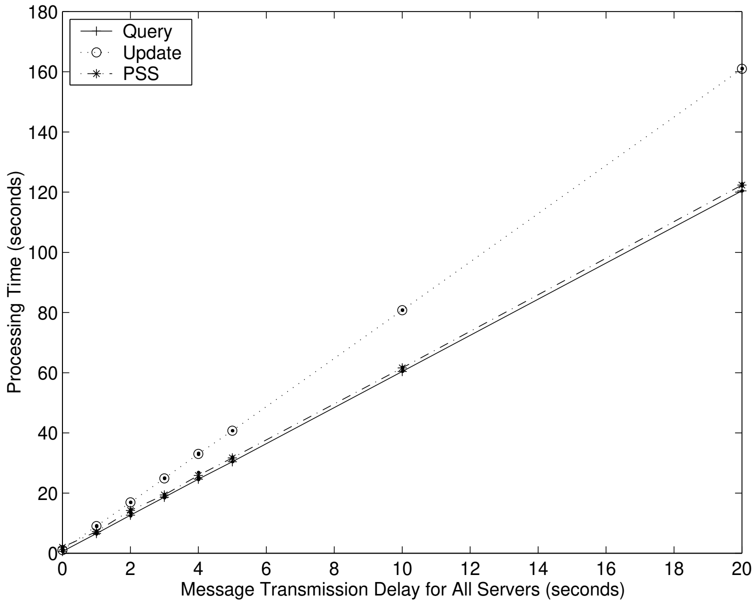

Fig. 10. Performance of COCA vs. message delay for all servers.

and removes non-responsive members from process groups to ensure the system does not stall due to compromised replicas. However, it is impossible to distinguish between slow and halted processors in an asynchronous system, so Rampart uses timeouts for identifying processors that might be compromised. A correct but slow server might thus be removed from a process group, and this constitutes a denial of service vulnerability. In addition, because making group membership changes involves expensive protocols, an adversary can launch denial of service attacks against Rampart by instigating membership changes. Furthermore, neither Rampart nor Ω employs proactive recovery, so these systems are vulnerable to mobile adversaries.

An approach related to Rampart is embodied in the Byzantine Fault Tolerance work (BFT) discussed in [Castro and Liskov 1999]. BFT employs an ordering mechanism that not only defines a total ordering on requests but also enables a server to know, given some request $\mathcal{R}$ received for processing, whether processing $\mathcal{R}$ should be delayed because some request whose ordering precedes $\mathcal{R}$ exists. But like COCA, BFT is unable to guarantee that requests are processed in an order consistent with Lamport's causality relation—that would require trusting all clients. BFT's stronger ordering mechanism is not needed for implementing COCA's Query and Update; it would be needed if the specification of Update were changed so that a copy of the certificate being updated were no longer passed as an argument. BFT is extremely fast because, wherever possible, it uses MACs (message authentication codes) instead of public key cryptography. Employing MACs would also boost COCA's performance, although public key cryptographic operations are needed by COCA for signing certificates and responses to clients.

As with COCA, BFT employs proactive recovery [Castro and Liskov 2000]. Even though BFT replicas do not store shares of a service private key, these replicas do refresh shared secret keys to combat mobile adversaries. BFT takes denial of service attacks into account and employs defenses similar to the mechanisms discussed for COCA in Section 4 [Castro 2000].

An approach to implementing secure and fault-tolerant services based on replication in asynchronous systems with potentially malicious adversaries has also been proposed in [Cachin 2001], and this seems to be a basis for the Hydra asynchronous group communication primitives [Cachin and Poritz 2001]. State machine replication [Lamport 1978] is intended here, with randomized Byzantine agreement to circumvent the impossibility result concerning agreement in asynchronous systems [Fischer et al. 1985]. The Hydra work does not, at present, address mobile adversary or denial of service attacks; COCA's solutions would apply.

The PASIS (Perpetually Available and Secure Information Systems) architecture [Wylie et al. 2000] is intended to support a variety of approaches—decentralized storage system technologies, data redundancy and encoding, and dynamic self-maintenance—that have been used in constructing survivable information storage systems. Once PASIS has been implemented, it should be possible to use it and program COCA's Query and Update in any number of ways. What is not clear is whether PASIS will support COCA's optimizations or defense against denial of service attacks, since doing so would depend on PASIS selecting a weak model of computation and supporting access to low-level details of the PASIS building-block protocols.

Replication and secret sharing are the basis for a fault-tolerant and secure key distribution center (KDC) described in [Gong 1993]. In this design, each client/KDC-server pair shares a separate secret key. The KDC allows two clients to establish their own shared secret key, and does so using protocols in which no single KDC-server ever knows that shared secret key. In fact, an attack must compromise a significant fraction of the KDC's servers before any keys the KDC establishes to link clients would be revealed.

Also related to COCA are various distributed systems that implement data repositories with operations analogous to Query and Update. Phalanx [Malkhi and Reiter 1998b] is particularly relevant, because it is intended for a setting quite similar to COCA's (*viz.* asynchronous systems in which compromised servers exhibit arbitrary behavior) and can be used to implement shared variables having similar semantics to COCA's certificates. (COCA's certificates can be regarded as shared variables that are being queried and updated.)

Phalanx supports two different implementations of read (Query) and write (Update) for shared variables. One implementation is optimized for *honest writers*, clients that follow specified protocols or exhibit benign failures (crash, omission, or timing failures); a second implementation tolerates *dishonest writers*, clients that can exhibit arbitrary behavior when faulty. Phalanx employs a masking Byzantine quorum system [Malkhi and Reiter 1998a] for dishonest writers and employs a dissemination quorum system for honest writers.[20]

In Phalanx's honest writer protocol, writers must be trusted to sign the objects being stored. Although, as with this honest writer protocol, COCA also uses a dissemination quorum system, COCA's protocols do not require clients to be trusted—COCA servers store objects (certificates) that are signed by COCA's service key, and that prevents compromised COCA servers from undetectably corrupting objects they store. Another point of difference between COCA and Phalanx is the manner in which clients verify responses from the service. In Phalanx, every client must know the public key of every server, whereas in COCA each client need know only the single public key for the service.

The e-vault data repository [Iyengar et al. 1998; Garay et al. 2000] at IBM T.J. Watson Research Center implements M. Rabin's information dispersal algorithm [Rabin 1989] for storing and retrieving files. Information is stored in e-vault with optimal space efficiency. But the e-vault protocols assume a synchronous model of computation and, thus, involve stronger assumptions about execution timing and delivery delays than we make for COCA. An attacker that is able to overload processors or clog the network can invalidate these assumptions and cause the e-vault protocols to fail. Like with COCA, clients of e-vault communicate with the system through a single server (there called a gateway).

Cryptographic Building Blocks and Public Key Infrastructure. COCA employs threshold cryptography [Desmedt and Frankel 1990; 1992; Desmedt 1994; 1998; Frankel and Yung 1998] and proactive secret sharing [Jarecki 1995; Herzberg et al.

[20]In a masking Byzantine quorum system, Quorum Intersection is strengthened to stipulate that the intersection of any two quorums always contains more correct replicas than compromised replicas. A masking Byzantine quorum system can tolerate compromise of as many as one quarter of its servers. Recall, a dissemination quorum system tolerates one third compromised servers.

1995; Herzberg et al. 1997; Frankel et al. 1997; Frankel et al. 1997] as building blocks. Because existing protocols were not intended for systems in which (only) our Fair Links and Asynchrony assumptions hold, it was necessary to design new protocols for COCA [Zhou et al. 2002; Zhou 2001]. Implementations of threshold cryptography and proactive secret sharing schemes for stronger system models are reported in [Barak et al. 1999; Wu et al. 1999; Draelos et al. 1998; CertCo, Inc. nd].

Most previous work on public key infrastructure (e.g., [Gasser et al. 1989; Tardo and Algappan 1991; Lampson et al. 1992; Kaufman 1993]) advocates off-line CAs, which issue certificates and certificate revocation lists (CRLs). Trade-offs associated with CRLs and related mechanisms are discussed in [Rivest 1998; Myers 1998; Kocher 1998; Fox and LaMacchia 1998; McDaniel and Rubin 2000]. Stubblebine [Stubblebine 1995] compares different mechanisms to deal with revoked certificates and argues that a single on-line service is impractical for both performance and security reasons, advocating a solution with an off-line identification authority and an on-line revocation authority. COCA could be used to implement the on-line part of such a solution.

In [Byrd et al. 2001], a security infrastructure consisting of a distributed CA and a certificate revocation notification service is discussed, although the implementation does not yet appear to be complete. As with COCA, the distributed CA employs threshold cryptography. However, the proposed CA does not support Query or Update, instead promptly notifying clients about invalidated certificates.

Alternatives to using an off-line CA include on-line certificate status checking (OCSP) [Myers et al. 1999; Myers 1998; Kocher 1998] and on-demand revocation lists [McDaniel and Rubin 2000]. The DVCS data validation and certification server [Adams et al. 2001] extends OCSP for checking arbitrary digitally signed documents. All of these services rely on some sort of trusted on-line service (a responder, a validation authority, and so on), so our experience implementing and deploying COCA is directly applicable.

Some believe that scalability in a global public key infrastructure would dictate deploying a hierarchy of certification authorities. Previous work (e.g., [Maurer 1996; Reiter and Stubblebine 1997; Burmester et al. 1998]) has applied the notion of "web of trust", first adopted in PGP [Zimmerman 1995], and exploited independent hosts or paths to establish trust in such an infrastructure. Services like those provided by COCA might still be desired in such an infrastructure, since that would allow clients to verify, on demand, certificate validity.

7. CONCLUDING REMARKS

Off-line operation of a CA—an air gap—is clearly an effective defense against network-borne attacks. For that reason, the traditional wisdom has been to keep a CA off-line as much as possible. This approach, however, trades one set of vulnerabilities for another. A CA that is off-line cannot be attacked using the network but it also cannot update or validate certificates on demand. Vulnerability to network-borne attacks is decreased at the expense of increased client vulnerability to attacks that exploit recently invalidated certificates.

By staying on-line, COCA makes the trade-off between vulnerabilities differently.

COCA's vulnerability to network-borne attacks is greater, but its clients' vulnerability to attacks based on invalidated certificates is reduced. Marrying COCA with an off-line CA would exhibit the advantages of both [Lampson et al. 1992; Stubblebine 1995; Myers et al. 1999]. The off-line CA issues certificates for clients, and COCA validates (on demand) these certificates. Revocation of a certificate is thus achieved by notifying COCA; issuance of a new certificate requires interacting with the off-line CA. We are now trading performance for security. In particular, while it becomes harder for an adversary to create a new, valid certificate (because that requires compromising the offline CA), it also now takes longer for a client to have a new certificate issued (because that requires interacting with the offline CA).

Looking to the Future

The development of COCA has led to more than a prototype on-line CA, more than specific protocols and denial of service defenses, and more than a set of experimental data documenting the performance of a system under certain attacks. In composing mechanisms for fault-tolerance and security, COCA implements a secure multi-party computation [Yao 1982; Goldreich et al. 1987; Ben-Or et al. 1988; Chaum et al. 1988]. Just as agreement protocols and their kin have become part of the vocabulary of system builders concerned with fault-tolerance, so too must protocols for secure multi-party computation if we aspire to build trustworthy systems. Query and Update have relatively simple semantics. For building richer services that are fault-tolerant and secure, we must become facile with implementing richer forms of secure multi-party computation—protocols that enable n mutually distrusted parties to compute a publicly known function on a secret input they share without disclosing the input or what input shares are held by the parties.

If one lesson from COCA is a call to investigate practical, secure, multi-party computation, a second is the value of weak assumptions—rather than specific mechanisms—for a principled approach to defending against attacks. Defenses based on weak assumptions are, by construction, accompanied by a characterization of vulnerabilities—the assumptions themselves. And, by their very nature, weak assumptions are difficult to violate. So, for example, careful attention paid to the assumptions that characterize COCA's environment led to a system with inherent defenses to denial of service attacks. New assumptions, however, invariably require the development of new protocols and perhaps also involve new kinds of guarantees which we must then learn to build on.

8. ACKNOWLEDGMENTS

We are grateful to Dag Johansen, David Kotz, and Keith Marzullo for loaning hardware that enabled us to deploy COCA on the Internet. Mike Reiter provided extremely helpful comments about related work. Feedback from Andrew Myers, Miguel Castro, Stuart Stubblebine, Christian Cachin, and Yacov Yacobi enabled us to sharpen and clarify our ideas. Li Gong, Steve Kent, and Cathy Meadows also provided helpful feedback on a draft of this paper. Yaron Minsky, Yvo Desmedt, and Zygmunt Haas were influential during the early stages of our investigations. And the ACM TOCS reviewers' remarks were extremely constructive.

APPENDIX

A. DETAILED DESCRIPTION OF PROTOCOLS

This appendix gives details for the protocols described in Section 3.[21] We describe the protocol initiated by a delegate p. In practice, more than one delegate could initiate the protocol for the same given request because a server p starts acting as a delegate when p first receives the request or when p receives any message related to the processing of the request. The optimizations outlined in Sections 4 and 5 are not included in this presentation.

The following notational conventions are used throughout the appendix:

p, q: COCA servers
c: COCA client
$\langle m \rangle_k$: message m signed by COCA using its service private key k
$\langle m \rangle_p$: message m signed by a server p using p's private key
$\langle m \rangle_c$: message m signed by a client c using c's private key
$PS(m, s_p)$: a partial signature for a message m generated by a server p using p's share s_p
$[h_1 \longrightarrow h_2 : m]$: message m is sent from host (a server or a client) h_1 to host h_2
$[\forall q.\ p \longrightarrow q : m_q]$: message m_q is sent from server p to server q for every COCA server q

Each message includes a type identifier to indicate the purpose of the message. These type identifiers are presented in the sans serif font.

A.1 Client Protocol

Every client request has the form:

$$\langle \mathit{type}, c, \mathit{seq}, \mathit{parm}, \mathit{cred} \rangle_c$$

where *type* indicates the type of the request, c is the client issuing the request, *seq* is a unique sequence number for the request, *parm* contains parameters related to the request, and *cred* is credentials that authorize the request.

Clients use the following protocol to communicate with COCA.

(1) To invoke Query for the certificate associated with name *cid*, client c composes a request:

$$\mathcal{R} = \langle \mathsf{query}, c, \mathit{seq}, \mathit{cid}, \mathit{cred} \rangle_c$$

To invoke Update to establish a new binding of *key* with name *cid* based on a given certificate ζ' for *cid*, client c composes a request:

$$\mathcal{R} = \langle \mathsf{update}, c, \mathit{seq}, \zeta', \langle \mathit{cid}, \mathit{key} \rangle, \mathit{cred} \rangle_c$$

(2) Client c sends $\mathcal{R}$ to $t + 1$ servers. It periodically re-sends $\mathcal{R}$ until it receives a response to its request. For a Query, the response will have the form $\langle \mathcal{R}, \zeta \rangle_k$, where ζ is a certificate for *cid*. For an Update, the response will have the form $\langle \mathcal{R}, \mathsf{done} \rangle_k$.

[21] See [Zhou et al. 2002] for a description of the proactive secret sharing protocol used by COCA.

A.2 Threshold Signature Protocol

The following describes threshold signature protocol[22] $threshold\_sign(m, \mathcal{E})$, where m is the message to be signed and $\mathcal{E}$ is the evidence used in self-verifying messages to convince receivers to generate partial signatures for m. As detailed in Appendices A.3 and A.4, different evidence is used in the protocols for Query and Update.

(1) Server p sends to each server q a sign_request message with message m to be signed and evidence $\mathcal{E}$.

$$[\forall q.\ p \longrightarrow q :\ \langle \mathsf{sign\_request}, p, m, \mathcal{E}\rangle_p] \tag{i}$$

(2) Each server q, upon receiving a sign_request message (i), verifies evidence $\mathcal{E}$ with respect to m. If $\mathcal{E}$ is valid, then q generates a partial signature using its share s_q and sends the partial signature back to p.

$$[q \longrightarrow p :\ \langle \mathsf{sign\_response}, q, p, m, PS(m, s_q)\rangle_q]$$

(3) Server p periodically repeats step 1 until it receives partial signatures from a quorum of servers[23] (which includes a partial signature from p itself). Server p then selects $t+1$ partial signatures to construct signature $\langle m\rangle_k$. If the resulting signature is invalid (which would happen if compromised servers submit erroneous partial signatures), then p tries another combination of $t+1$ signatures.[24] This process continues until the correct signature $\langle m\rangle_k$ is obtained.

A.3 Query processing protocol

(1) Upon receiving a request $\mathcal{R} = \langle \mathsf{query}, c, seq, cid, cred\rangle_c$ from a client c, server p first checks whether $\mathcal{R}$ is valid based on the credentials $cred$ provided. If $\mathcal{R}$ is valid then p sends a query_request message to all servers:

$$[\forall q.\ p \longrightarrow q :\ \langle \mathsf{query\_request}, p, \mathcal{R}\rangle_p] \tag{ii}$$

(2) Each server q, upon receiving a query_request message (ii), checks the validity of the request. If the request is valid, then q fetches the current signed local certificate associated with name cid: $\zeta_q = \langle cid, \sigma(\zeta_q), key_q\rangle_k$. Server q then sends back to p the following message:

$$[q \longrightarrow p :\ \langle \mathsf{query\_response}, q, p, \mathcal{R}, \zeta_q\rangle_q]$$

[22] While this protocol is appropriate for schemes such as threshold RSA, the protocol might not be applicable to other threshold signature schemes, such as those based on discrete logarithms (e.g., [Cerecedo et al. 1993; Harn 1994]). Those schemes may require an agreed-upon random number in generating partial signatures. Such schemes can be implemented by adding a new first step, in which a delegate decides a random number based on suggestions from $t+1$ servers (to ensure randomness) and notifies others of this random number before servers can generate partial signatures.

[23] In fact, p can try to construct the signature as soon as it has received $t+1$ partial signatures. p has to wait for more partial signatures only if some partial signatures it received are incorrect.

[24] In the worst case, p must try $\binom{2t+1}{t+1}$ combinations. The cost is insignificant when t is small. There are robust threshold cryptography schemes [Gennaro et al. 1996a; 1996b] that can reduce the cost by using error correction codes.

(3) Server p repeats step 1 until it receives $\mathsf{query\_response}$ messages from a quorum of servers (including p itself). p verifies that the certificates in these messages are correctly signed by COCA. Let $\zeta = \langle cid, \sigma, key \rangle_k$ be the certificate with the largest serial number in these $\mathsf{query\_response}$ messages. Server p invokes $\mathit{threshold\_sign}(m, \mathcal{E})$, where m is $(\mathcal{R}, \zeta)$ and $\mathcal{E}$ is the $\mathsf{query\_response}$ messages collected from a quorum of servers, thereby obtaining $\langle \mathcal{R}, \zeta \rangle_k$.

(4) Server p sends the following response to client c:[25]

$$[p \longrightarrow c: \ \langle \mathcal{R}, \zeta \rangle_k].$$

A.4 Update processing protocol

(1) Upon receiving a request $\mathcal{R} = \langle \mathsf{update}, c, \mathit{seq}, \zeta', \langle cid, key \rangle, \mathit{cred} \rangle_c$ from a client c, server p first checks whether $\mathcal{R}$ is valid, based on the credentials cred provided. If $\mathcal{R}$ is valid then p computes serial number $\sigma(\zeta) = (v+1, h(\mathcal{R}))$ for new certificate ζ, where v is the version number of ζ' and h is a public collision-free hash function. Then, p invokes $\mathit{threshold\_sign}(m, \mathcal{E})$, where m is $\langle cid, \sigma(\zeta), key \rangle$ and $\mathcal{E}$ is $\mathcal{R}$, thereby obtaining $\zeta = \langle cid, \sigma(\zeta), key \rangle_k$.

(2) Server p then sends an $\mathsf{update\_request}$ message to every server q.

$$[\forall q.\ p \longrightarrow q: \ \langle \mathsf{update\_request}, p, \mathcal{R}, \zeta \rangle_p] \tag{iii}$$

(3) Each server q, upon receiving an $\mathsf{update\_request}$ message (iii), updates its certificate for cid with ζ if and only if $\sigma(\zeta_q) < \sigma(\zeta)$, where ζ_q is the certificate for cid stored by the server. Server q then sends back to p the following message:

$$[q \longrightarrow p: \ \langle \mathsf{update\_response}, q, p, \mathcal{R}, \mathsf{done} \rangle_q]$$

(4) Server p repeats step 2 until it receives the $\mathsf{update\_response}$ messages from a quorum of servers. p then invokes $\mathit{threshold\_sign}(m, \mathcal{E})$, where m is $(\mathcal{R}, \mathsf{done})$ and $\mathcal{E}$ is the $\mathsf{update\_response}$ messages collected from a quorum of servers, thereby obtaining $\langle \mathcal{R}, \mathsf{done} \rangle_k$.

(5) Server p sends the following response to client c:

$$[p \longrightarrow c: \ \langle \mathcal{R}, \mathsf{done} \rangle_k]$$

REFERENCES

Adams, C., Sylvester, P., Zolorarev, M., and Zuccherato, R. 2001. Data validation and certification server protocols. Request for Comments 3029.

Baldoni, R., Helary, J.-M., and Raynal, M. 2000. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Proceedings of the International Conference on Dependable Systems and Networks.* IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society, New York, NY USA, 273–282.

Barak, B., Herzberg, A., Naor, D., and Shai, E. 1999. The proactive security toolkit and applications. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS'99).* ACM SIGSAC, ACM, Kent Ridge Digital Labs, Singapore, 18–27.

[25] To implement the optimization described in Section 5, p also forwards the response to all other servers. Henceforth, these servers do not need to act as a delegate for this request any more. The same is true for the last step of Update request processing.

Ben-Or, M., Goldwasser, S., and Wigderson, A. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*. ACM, Chicago, IL USA, 1–10.

Burmester, M., Desmedt, Y., and Kabatianski, G. 1998. Trust and security: A new look at the Byzantine generals problem. In *Network Threats, DIMACS series in Discrete Mathematics and Theoretical Computer Science*, R. N. Wright and P. G. Neumann, Eds. Vol. 38. American Mathematical Society, Providence, RI USA, 75–83.

Byrd, G. T., Gong, F., Sargor, C., and Smith, T. J. 2001. Yalta: A secure collaborative space for dynamic coalitions. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*. IEEE, United States Military Academy, West Point, NY USA, 30–37.

Cachin, C. 2001. Distributing trust on the Internet. In *Proceedings of International Conference on Dependable Systems and Networks (DSN-2001)*. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society, Göteborg, Sweden, 183–192.

Cachin, C. and Poritz, J. A. 2001. Hydra: Secure replication on the Internet. Technical Report RZ 3393, IBM Research. December.

Canetti, R. and Herzberg, A. 1994. Maintaining security in the presence of transient faults. In *Advances in Cryptology—Crypto'94, the 14th Annual International Cryptology Conference, Proceedings*, Y. Desmedt, Ed. Lecture Notes in Computer Science, vol. 839. Springer-Verlag, Berlin, Germany, 425–438.

Castro, M. 2000. Practical Byzantine fault tolerance. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA.

Castro, M. and Liskov, B. 1999. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI'99)*. USENIX Association, IEEE TCOS, and ACM SIGOPS, New Orleans, LA USA, 173–186.

Castro, M. and Liskov, B. 2000. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'00)*. USENIX Association, IEEE TCOS, and ACM SIGOPS, San Diego, CA USA, 273–287.

CCITT. 1988. Recommendation X.509: The directory-authentication framework.

Cerecedo, M., Matsumoto, T., and Imai, H. 1993. Efficient and secure multiparty generation of digital signatures based on discrete logarithms. *IEICE Transactions on Fundamentals of Electronics, Information and Communication Engineers E76-A,* 4 (April), 532–545.

CertCo, Inc. n.d. http://www.certco.com.

Chaum, D. 1983. Blind signatures for untraceable payments. In *Advances in Cryptology—Crypto'82, A Workshop on the Theory and Application of Cryptography, Proceedings*, D. Chaum, R. L. Rivest, and A. T. Sherman, Eds. Plenum Press, New York, 199–203.

Chaum, D. 1985. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM 28,* 10 (October), 1030–1044.

Chaum, D. 1988. Blinding for unanticipated signatures. In *Advances in Cryptology—Eurocrypt'87, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, D. Chaum and W. L. Price, Eds. Lecture Notes in Computer Science, vol. 304. Springer-Verlag, Berlin, Germany, 227–233.

Chaum, D., Crépeau, C., and Damgård, I. 1988. Multiparty unconditionally secure protocols. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*. ACM, Chicago, IL USA, 11–19.

Desmedt, Y. 1994. Threshold cryptography. *European Transactions on Telecommunications 5,* 4 (July–August), 449–457.

Desmedt, Y. 1998. Some recent research aspects of threshold cryptography. In *Information Security, The 1st International Workshop, ISW'97, Proceedings*, E. Okamoto, G. Davida, and M. Mambo, Eds. Lecture Notes in Computer Science, vol. 1396. Springer-Verlag, Berlin, Germany, 158–173.

Desmedt, Y. and Frankel, Y. 1990. Threshold cryptosystems. In *Advances in Cryptology—Crypto'89, the 9th Annual International Cryptology Conference, Proceedings*, G. Brassard, Ed. Lecture Notes in Computer Science, vol. 435. Springer-Verlag, Berlin, Germany, 307–315.

Desmedt, Y. and Frankel, Y. 1992. Shared generation of authenticators and signatures (Extended Abstract). In *Advances in Cryptology—Crypto'91, the 11th Annual International Cryptology Conference, Proceedings*, J. Feigenbaum, Ed. Lecture Notes in Computer Science, vol. 576. Springer-Verlag, Berlin, Germany, 457–469.

Doudou, A., Garbinato, B., and Guerraoui, R. 2000. Modular abstractions for devising Byzantine-resilient state machine replication. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*. IEEE Computer Society, Nürnberg, Germany, 144–153.

Draelos, T., Hamilton, V., and Istrail, G. 1998. Proactive DSA application and implementation. Tech. Rep. SAND--97-2939C; CONF-980554--, Sandia National Laboratories, Albuquerque, NM USA. May 3.

Feldman, P. 1987. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*. IEEE, New York, 427–437.

Fischer, M. J., Lynch, N. A., and Peterson, M. S. 1985. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM 32,* 2 (April), 374–382.

Fox, B. and LaMacchia, B. 1998. Certificate revocation: Mechanics and meaning. In *Financial Cryptography, the 2nd International Conference (FC'98), Proceedings*, R. Hirschfeld, Ed. Lecture Notes in Computer Science, vol. 1465. Springer-Verlag, Berlin, Germany, 158–164.

Frankel, Y., Gemmel, P., MacKenzie, P., and Yung, M. 1997. Optimal resilience proactive public-key cryptosystems. In *Proceedings of the 38th Symposium on Foundations of Computer Science*. IEEE, Miami Beach, FL USA, 384–393.

Frankel, Y., Gemmell, P., MacKenzie, P., and Yung, M. 1997. Proactive RSA. In *Advances in Cryptology—Crypto'97, Proceedings*, B. S. Kaliski Jr., Ed. Lecture Notes in Computer Science, vol. 1294. Springer-Verlag, Berlin, Germany, 440–454.

Frankel, Y. and Yung, M. 1998. Distributed public key cryptosystem. In *Public Key Cryptography, the 1st International Workshop on Practice and Theory in Public Key Cryptography, PKC'98, Proceedings*, H. Imai and Y. Zheng, Eds. Lecture Notes in Computer Science, vol. 1560. Springer-Verlag, Berlin, Germany, 1–13.

Freier, A. O., Karlton, P., and Kocher, P. C. 1996. The SSL protocol Version 3.0. Internet Draft.

Garay, J. A., Gennaro, R., Jutla, C., and Rabin, T. 2000. Secure distributed storage and retrieval. *Theoretical Computer Science 243,* 1–2 (July 28), 363–389.

Gasser, M., Goldstein, A., Kaufman, C., and Lampson, B. 1989. The digital distributed systems security architecture. In *Proceedings of the 12th National Computer Security Conference*. National Institute of Standards and Technology (NIST), National Computer Security Center (NCSC), National Institute of Standards and Technology (NIST), Baltimore, MD USA, 305–319.

Gennaro, R., Jarecki, S., Krawczyk, H., and Rabin, T. 1996b. Robust and efficient sharing of RSA functions. In *Advances in Cryptology—Crypto'96, the 16th Annual International Cryptology Conference, Proceedings*, N. Koblitz, Ed. Lecture Notes in Computer Science, vol. 1109. Springer-Verlag, Berlin, Germany, 157–172.

Gennaro, R., Jarecki, S., Krawczyk, H., and Rabin, T. 1996a. Robust threshold DSS signatures. In *Advances in Cryptology—Eurocrypt'96, International Conference on the Theory and Application of Cryptographic Techniques, Proceedings*, U. M. Maurer, Ed. Lecture Notes in Computer Science, vol. 1233. Springer-Verlag, Berlin, Germany, 354–371.

Gligor, V. D. 1984. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering 10,* 3 (May), 320–324.

Goldreich, O., Micali, S., and Wigderson, A. 1987. How to play ANY mental game. In *Proceedings of the 19th Annual Conference on Theory of Computing, STOC'87*. ACM, New York, NY USA, 218–229.

Gong, L. 1993. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications 11,* 5 (June), 657–662.

Gong, L. and Syverson, P. 1998. Fail-stop protocols: An approach to designing secure protocols. In *Dependable Computing for Critical Applications 5*, R. K. Iyer, M. Morganti, W. K. Fuchs, and V. Gligor, Eds. IEEE Computer Society Press, New York, 79–99.

Harn, L. 1994. Group oriented (t, n) digital signature scheme. *IEE Proceedings—Computer and Digital Techniques 141,* 5 (September), 307–313.

Herzberg, A., Jakobsson, M., Jarecki, S., Krawczyk, H., and Yung, M. 1997. Proactive public-key and signature schemes. In *Proceedings of the 4th Annual Conference on Computer Communications Security.* ACM SIGSAC, ACM, Zürich, Switzerland, 100–110.

Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M. 1995. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology—Crypto'95, the 15th Annual International Cryptology Conference, Proceedings*, D. Coppersmith, Ed. Lecture Notes in Computer Science, vol. 963. Springer-Verlag, Berlin, Germany, 457–469.

Iyengar, A., Cahn, R., Jutla, C., and Garay, J. 1998. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC'98).* International Federation for Information Processing, TC11: Security and Protection in Information Processing Systems, Elsevier Science, Vienna, Austria and Budapest, Hungary, 123–135.

Jarecki, S. 1995. Proactive secret sharing and public key cryptosystems. M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA.

Juels, A. and Brainard, J. 1999. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium.* Internet Society, San Diego, CA USA, 151–165.

Karn, P. and Simpson, W. 1997. The Photuris session key management protocol. Internet Draft: draft-simpson-photuris-17.txt.

Kaufman, C. 1993. DASS: Distributed authentication security service. Request for Comments 1507.

Kent, S. T., Ellis, D., Helinek, P., Sirois, K., and Yuan, N. 1996. Internet infrastructure security countermeasures. Tech. Rep. 8173, BBN. January.

Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. 1997. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'97).* Hermes, Chantilly, France, 61–76.

Kocher, P. C. 1998. On certificate revocation and validation. In *Financial Cryptography, the 2nd International Conference (FC'98), Proceedings*, R. Hirschfeld, Ed. Lecture Notes in Computer Science, vol. 1465. Springer-Verlag, Berlin, Germany, 172–177.

Kornfelder, L. M. 1978. Toward a practical public-key cryptosystem. Bachelor's thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA USA.

Labovitz, C., Malan, G. R., and Jahanian, F. 1997. Internet routing instability. In *Proceedings of the Annual Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97).* ACM, Cannes, French Riviera, France, 115–126.

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

Lampson, B., Abadi, M., Burrows, M., and Wobber, E. 1992. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10,* 4, 265–310.

Malkhi, D. and Reiter, M. 1998a. Byzantine quorum systems. *Distributed Computing 11,* 4, 203–213.

Malkhi, D. and Reiter, M. 1998b. Secure and scalable replication in Phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems.* IEEE Computer Society, West Lafayette, IN USA, 51–58.

MAURER, U. 1996. Modeling a public-key infrastructure. In *Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS'96)*, E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Eds. Lecture Notes in Computer Science, vol. 1146. Springer-Verlag, Berlin, Germany, 325–350.

MCDANIEL, P. AND RUBIN, A. 2000. A response to "Can we eliminate revocation lists?". In *Financial Cryptography, the 4th International Conference (FC'00), Proceedings*, Y. Frankel, Ed. Lecture Notes in Computer Science, vol. 1962. Springer-Verlag, Berlin, Germany, 245–258.

MEADOWS, C. 1999. A formal framework and evaluation method for network denial of service. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, Mordano, Italy, 4–13.

MEADOWS, C. 2001. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security 9,* 1/2, 143–164.

MILLEN, J. K. 1992. A resource allocation model for denial of service. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA USA, 137–147.

MILLEN, J. K. 1995. Denial of service: A perspective. In *Dependable Computing for Critical Applications 4*, F. Cristian, G. L. Lann, and T. Lunt, Eds. Springer-Verlag, Berlin, Germany, 93–108.

MYERS, M. 1998. Revocation: Options and challenges. In *Financial Cryptography, the 2nd International Conference (FC'98), Proceedings*, R. Hirschfeld, Ed. Lecture Notes in Computer Science, vol. 1465. Springer-Verlag, Berlin, Germany, 165–171.

MYERS, M., ANKNEY, R., MALPANI, A., GALPERIN, S., AND ADAMS, C. 1999. X.509 Internet public key infrastructure online certificate status protocol (OCSP). Request For Comments 2560.

OPENSSL PROJECT. n.d. http://www.openssl.org.

OPPLIGER, R. 1999. Protecting key exchange and management protocols against resource clogging attacks. In *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*, B. Preneel, Ed. International Federation for Information Processing, Kluwer Academic Publishers, Leuven, Belgium, 163–175.

OSTROVSKY, R. AND YUNG, M. 1991. How to withstand mobile virus attacks. In *Proceedings of the 10th Annual Symposium on Principles of Distributed Computing (PODC'91)*. ACM, Montreal, Quebec, Canada, 51–59.

RABIN, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36,* 2 (April), 335–348.

RABIN, T. 1998. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology—Crypto'98, the 18th Annual International Cryptology Conference, Proceedings*, H. Krawczyk, Ed. Lecture Notes in Computer Science, vol. 1462. Springer-Verlag, Berlin, Germany, 89–104.

REITER, M. K. 1995. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems, International Workshop, Selected Papers*, K. P. Birman, F. Mattern, and A. Schiper, Eds. Lecture Notes in Computer Science, vol. 938. Springer-Verlag, Berlin, Germany, 99–110.

REITER, M. K. 1996. Distributing trust with the Rampart toolkit. *Communications of the ACM 39,* 4 (April), 71–74.

REITER, M. K., BIRMAN, K. P., AND VAN RENESSE, R. 1994. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems 12,* 4 (November), 340–371.

REITER, M. K., FRANKLIN, M. K., LACY, J. B., AND WRIGHT, R. N. 1996. The Ω key management service. *Journal of Computer Security 4,* 4, 267–297.

REITER, M. K. AND STUBBLEBINE, S. G. 1997. Path independence for authentication in large-scale systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*. ACM SIGSAC, ACM, Zürich, Switzerland, 57–66.

RICHTEL, M. AND ROBINSON, S. 2000. Several web sites attacked following assaults on Yahoo. *New York Times*. February 8.

Rivest, R. L. 1998. Can we eliminate revocation lists? In *Financial Cryptography, the 2nd International Conference (FC'98), Proceedings*, R. Hirschfeld, Ed. Lecture Notes in Computer Science, vol. 1465. Springer-Verlag, Berlin, Germany, 178–183.

Schuba, C., Krsul, I., Kuhn, M., Spafford, G., Sundaram, A., and Zamboni, D. 1997. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA USA, 208–223.

Stubblebine, S. G. 1995. Recent-secure authentication: Enforcing revocation in distributed systems. In *Proceedings of the 1995 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, Oakland, CA USA, 224–234.

Tardo, J. J. and Algappan, K. 1991. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA USA, 232–244.

Wu, T., Malkin, M., and Boneh, D. 1999. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*. The USENIX Association, Washington, D.C. USA, 79–91.

Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliççöte, H., and Khosla, P. K. 2000. Survivable information storage system. *IEEE Computer 33,* 8 (August), 61–68.

Yao, A. C. 1982. Protocols for secure computation. In *Proceedings of the 23rd Symposium on Foundations of Computer Science (FOCS'82)*. IEEE, Chicago, IL USA, 160–164.

Yu, C.-F. and Gligor, V. D. 1990. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering 16,* 6 (June), 581–592.

Zhou, L. 2001. Towards building secure and fault-tolerant on-line services. Ph.D. thesis, Computer Science Department, Cornell University, Ithaca, NY USA.

Zhou, L., Schneider, F. B., and van Renesse, R. 2002. Proactive secret sharing for asynchronous systems. In preparation.

Zimmerman, P. R. 1995. *The Official PGP User's Guide*. MIT Press, Cambridge, MA USA.

Received December 2000; revised February 2002; accepted May 2002