COCA: A Secure Distributed On-line Certification Authority[1]

Lidong Zhou, Fred B. Schneider, and Robbert van Renesse

Department of Computer Science
Cornell University
Ithaca, New York 14853

December 7, 2000

[1]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.

Abstract

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. Replication is used to achieve availability; proactive recovery with threshold cryptography is used for digitally signing certificates in a way that defends against mobile adversaries which attack, compromise, and control one replica for a limited period of time before moving on to another. Relatively weak assumptions characterize environments in which COCA's protocols will 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. The result is a system with inherent defenses to certain denial of service attacks because, by their very nature, weak assumptions are difficult for attackers to invalidate. In addition, traditional techniques, including request authorization, resource management based on segregation and scheduling different classes of requests, as well as caching results of expensive cryptographic operations further reduce COCA's vulnerability to denial of service attacks. Results from experiments in a local area network and the Internet allow a quantitative evaluation of the various means COCA employs to resist denial of service attacks.

1 Introduction

In a public key infrastructure, a *certificate* [45] specifies a binding between a name and a public key or other attributes. Over time, public keys and attributes might change—a private key might be compromised, leading to selection of a new public key, for example. The old binding and the certificate that specifies that binding then become *invalid*. A *certification authority* (CA) attests to the validity of bindings in certificates by digitally signing the certificates it issues 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.[1] 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* [60] which attack, compromise, and control one replica for a limited period of time before moving on to another. 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.

COCA also employs traditional means for combatting denial of service attacks: (i) processing only those requests that satisfy authorization checks, (ii) grouping requests into classes and multiplexing resources so that demand 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, our performance experiments demonstrate that launching a successful attack against COCA is harder with these mechanisms in place. In fact, simulated denial of service attacks allowed us to measure the cost and effectiveness of the various means COCA employs to resist denial of service attacks.

The paper is organized as follows. Section 2 discusses our assumptions about any environment in which COCA is deployed 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

[1] The vulnerability is not completely eliminated, because some time elapses from when the CA sends its response until the client receives that response, and additional time might also elapse until the certificate is actually used.

to defend against denial of service attacks. Performance data for COCA deployments both in a local area network and in the Internet (with servers at University of Tromsø Norway, Cornell, Dartmouth, and U.C. San Diego) are summarized in Section 5, followed by a discussion of related work in Section 6. Section 7 contains some concluding remarks.

2 System Model and Services Supported

COCA is implemented by a set of servers, each running on a separate processor in a network. We intend COCA for use in an environment like the Internet. Thus, the system must tolerate failures and defend against malicious attacks that target clients, servers, and network communications links, as follows.

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 might disclose information stored locally. System execution comprises a sequence of protocol-defined windows of vulnerability; terms "correct" and "compromised" refer 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 protocol-defined window of vulnerability, where $3t + 1 \leq n$ holds.
- Clients and servers can digitally sign messages using some scheme that is non-existentially forgeable, even with adaptive chosen message attacks.
- Various cryptographic schemes (e.g., public key cryptography and threshold cryptography) that COCA employs are secure.

Fair Links: A *fair communication link* does not necessarily deliver all messages sent, but if a process, using such a link, sends infinitely many messages to a single destination then infinitely many of those messages are correctly delivered. 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. Assumptions about those bounds could be invalidated by denial of service attacks. By eschewing such assumptions, a class of vulnerabilities is thus eliminated.

These three classes of assumptions endow adversaries with considerable power. Attackers can:

- attack servers, provided fewer than 1/3 of the servers are compromised within a given interval,
- 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 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 with a COCA service key and include the client's request, hence the nonce[2], 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" once the client receives a response. But such a definition would make a client action (receipt of a response) necessary for a request to be considered completed, and implementing COCA's

Request Completion: Every request accepted is eventually completed.

[2]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.

guarantee then becomes problematic in the absence of assumptions about clients. But a correct client that makes a request will eventually receive a response from COCA.

Certificates stored by COCA are X.509 [7] compliant. It will be convenient here to regard each certificate ζ simply as 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, and 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 new 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 Update by specifying a new binding for *cid*.

Update creates and invalidates certificates, so it 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 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. Thus, the semantics of Update allows $\mathcal{U}$ to create a certificate $\zeta, \mathcal{U}'$ 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 [47] (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. By using service-centric causality, COCA and its clients are not hostage to information about causality furnished by compromised clients.

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.

Since we wanted Query to return a certificate, having 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.

And COCA clients therefore live with a semantics for Query that is more complicated than one might have hoped for.

2.2 Bounding the Window of Vulnerability

COCA is designed to operate provided no more than t servers are compromised within a protocol-defined *window of vulnerability*. The duration of this 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 protocols (described below) that are executed periodically to refresh keys and server states. Together, these *proactive recovery protocols* reconstitute the state of each COCA server (which might have been corrupted during the previous window of vulnerability) and obsolete keys 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.[3] Given such information about 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, with the public key given to all COCA servers. These public keys allow servers to authenticate the senders of messages they exchange with other servers.

Public keys of COCA servers are not given to COCA clients so that clients

[3] A server that violates these stronger execution timing assumptions might be considered compromised, for example.

need not be informed of changed server keys—attractive in a system with a large number of clients and where a proactive recovery protocol periodically refreshes server keys. But without knowledge of server keys, clients cannot easily determine the COCA server that sent a message. This, in turn, precludes voting or other schemes in which a client synthesizes or counts responses from individual COCA servers to obtain COCA's response.

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, for obvious reasons. Instead, different shares of the key are stored on each of the servers, and threshold cryptography [16, 17, 14, 15, 24] 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.[4]

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 Recovery. 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. New shares cannot be combined with old shares to construct signatures. And 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.

[4] One might think partial signatures could be combined by clients (instead of COCA servers) to obtain a signed messages, but that introduces a vulnerability to denial of service attacks. Lacking COCA server public keys, clients do not have a way to authenticate the origins of messages conveying the partial signatures. Therefore, a client 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.

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., [38, 36, 35, 23, 22]). Protocol details are discussed in [78, 77]. For the discussion in this paper, we can regard the protocols simply as services that COCA invokes.

There is one non-obvious point of interaction involving the proactive recovery protocols used to refresh server keys and service key shares. To satisfy Request Completion (of Section 2.1), 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, correct servers about to execute the proactive recovery protocol resubmit to all servers any requests that are then in progress; these requests are marked so that they will be processed during the correct (i.e., next) window of vulnerability. Some server that is correct in this next window of vulnerability will receive the request. Thus, by definition, in-progress accepted requests in the previous window of vulnerability remain accepted in the next one.

In practice, windows of vulnerability 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.

Server State Recovery

In addition to generating new server keys and new shares of the service key, COCA also periodically refreshes the states of its servers. This is done as part of proactive recovery. The state of a COCA server consists of 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 that becomes prohibitive when many certificates are being stored by COCA. So instead, during proactive recovery, a list with the name and serial number for every valid certificate stored by each server is sent to every other. Upon receiving this list, a server retrieves any certificates that appear to be missing. Certificates stored by COCA servers are signed (by COCA)—a certificate retrieved from another server can thus 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).

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 [49], which is feasible because we have assumed $3t + 1 \leq n$ holds. So servers are organized 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. The technical challenges are:

- 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 is the correct certificate.

- 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.

[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.

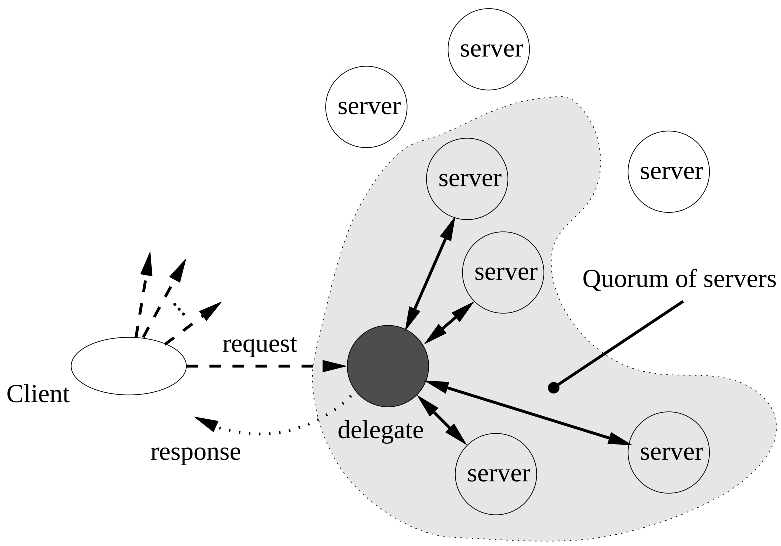

Figure 1: Overview of client request processing.

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

Figure 1 summarizes this 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.

Protocol Details

Certificate Serial Numbers. The serial number $\sigma(\zeta)$ for a COCA certificate ζ is 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. This is because the serial number of a certificate is entirely determined by the arguments in the request that creates the certificate. So, Update requests are idempotent, which proves useful for tolerating compromised COCA servers.

Determining a Response for Query. 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) are taken into account in determining the response to a Query request $\mathcal{Q}$. To satisfy this, a quorum of servers must be engaged in processing $\mathcal{Q}$. All servers are contacted and responses from a quorum of servers are expected. Each server in a quorum Q responds with the certificate (signed by COCA) having the largest serial number among all certificates (for *cid*) known to the server. The certificate ζ that has the largest serial number among the correctly signed certificates received in the responses from Q is the response to $\mathcal{Q}$.

This choice of ζ satisfies parts (i) and (ii) in the specification for Query. Part (i) stipulates that a certificate returned for Query is created by an accepted Update. This condition will be satisfied by ζ because a certificate is signed by COCA only after the Update request creating that certificate has been accepted. 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. The signature on certificates also prevents a compromised server from submitting a bogus certificate with an arbitrarily high serial number during the processing of a Query request without being detected.

Part (ii) of the Query specification 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 the implementation outlined 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 $\mathcal{Q}$. Because p always chooses the certificate for *cid* with the largest serial number, $\sigma(\zeta') \leq \sigma(\zeta_p)$ holds. 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. After making a request $\mathcal{R}$, a client awaits notification that $\mathcal{R}$ has been processed. 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 quorums contain $2t + 1$ servers, 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 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

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

COCA servers is not a problem either, because COCA's Query and Update implementations are idempotent.

But a compromised client might not submit its request to $t+1$ delegates, as is now required. We must ensure that Request Completion is not violated. The problem occurs if the delegates receiving that request $\mathcal{R}$ execute the first step of Query or Update processing and then halt. Correct COCA servers now participate 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.

We must ensure that some correct COCA server becomes a delegate for each request that has been received by any correct COCA server. The solution 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 could 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 binding 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

[7] Similar schemes can be found in [43, 5, 1, 18].

- 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. 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]

Returning to the erroneous message examples given above, here is how the self-verifying messages used in COCA prevent subversion of the service:

- Compromised delegates cannot cause COCA to sign a Query response containing a bogus or invalidated certificate, because messages instructing servers to sign such a response must contain signed messages from a quorum of servers, where these signed messages contain the certificates submitted by servers for this Query.

- Compromised delegates are prevented from creating a certificate that specifies a fictitious binding, because every message pertaining to an Update request must include the original client's signed request. COCA servers check that message before signing a new certificate.

- Compromised delegates that do not disseminate some new certificate to a quorum are foiled, because every subsequent message the delegate sends in processing this request must contain the signed responses from a quorum of servers attesting that they received the new certificate.

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

[8]In [33], 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.

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.

The protocols in COCA are 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 [39, 52, 53]. 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.

It is the details for COCA's realizations of these defenses that constitutes the bulk of this section.

Note, however, that our Fair Links and Asynchrony system-model 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., [30, 76, 54, 55]) would not be satisfied by an eventuality guarantee.

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 [70]. But the proactive secret sharing protocol in the current COCA implementation does use SSL (Secure Socket Layer) [25] 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 [39].

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 have been 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 [40, 59], hash chains [42], and puzzles [39] 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 [59].

4.2 Resource Management

Because requests are signed, COCA servers are able to identify the client and/or server associated with each message received. And 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 some scheduler to service those queues. The queues and scheduler limit the fraction of a server's cycles that can be co-opted by an attacker.[9] Others have also advocated similar approaches [30, 76, 54, 55].

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

[9]Clearly, this offers no defense against distributed denial of service attacks [67] 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.

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 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 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 suggested in [59, 4]. 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. Each COCA server cache has finite capacity, so all 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] 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.

[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.

Because requests are processed by a quorum of COCA servers—and not necessarily by all COCA servers—a given server's cache of client responses might not be current. Thus, a replay request signed by client c to some server s might 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.

Clients are not the only source of replay-based denial of service attacks. Compromised servers also could attempt such attacks. COCA's defense here too is a cache. Servers cache results from all expensive operations, such as computing new 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 X.509 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.

5 Performance of COCA

Our COCA prototype is approximately 35K lines of new C source; it employs a threshold RSA scheme and a proactive threshold RSA scheme [62] (using 1024-bit RSA keys) that we built using OpenSSL [58]. Certificates stored on COCA servers are in accordance with X.509 [7], with the COCA's serial number embedded in the X.509 serial number.

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 COCA prototype sequences when servers start serving as 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 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 will be 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.

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

Table 1: Execution Time in a LAN when N1 and N2 hold.

5.1 Local Area Network Deployment

These experiments involved a COCA prototype comprising four servers (i.e., $n = 4$ and $t = 1$) communicating through a 100Mb 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 the Ethernet is usually under 300 microseconds.

Table 1 gives times for COCA functions 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. The reported sample means and sample standard deviations are based on 100 executions. All samples are located within 5% of the mean.

To better understand the origin of these delays, we report in Table 2 the (percentage) contribution that can be attributed to certain CPU-intensive cryptographic operations. For Query and Update, we measured the time spent in generating partial signatures and in signing messages. For proactive secret sharing, we measured the delay associated with the one-way function[11], 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 must sometimes wait for one another, is also listed in Table 2. 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 were observed here. The optimizations thus can

[11] The one-way function involves expensive modular exponentiation and is needed to implement verifiable secret sharing [20].

	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%

Table 2: Breakdown of costs for Query, Update, and proactive secret sharing (PSS) in local area network deployment.

be seen to be effective.

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 performance, we deployed four COCA servers as follows.

- University of Tromsø, Tromsø, Norway. (300 MHz, Pentium II)
- University of California, 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.[12] Figure 3 depicts the average message delivery delay (measured using ping) between these servers. Delivery delays on the Internet vary considerably [46] but the values observed during the experiments we report did not differ significantly from those in Figure 3.

Table 3 gives measurements for the Cornell host in our 4-site Internet deployment. In comparing Table 1 and Table 3, we see the impact of the Internet's longer communication delays (which also lead to longer server idle

[12]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 real COCA deployment, we would recommend having different hardware and different operating systems at each site so that common-mode vulnerabilities are reduced.

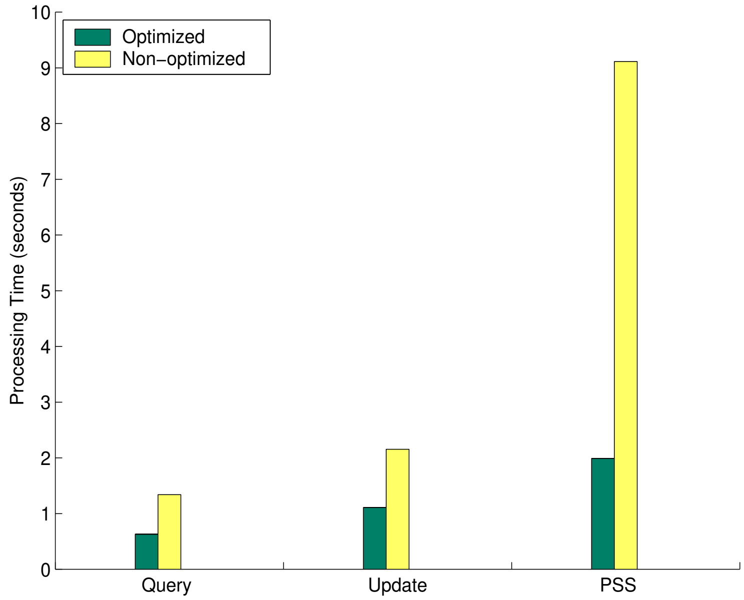

Figure 2: Effectiveness of optimization in Query, Update and proactive secret sharing (PSS) when assumptions N1 and N2 hold.

time). 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 4 for a breakdown of delays (analogous to Table 2) for our Internet deployment of COCA.

5.3 COCA Performance and Denial of Service Attacks

Any denial of service attack will ultimately involve some combination of compromised clients and/or servers (i) sending new messages, (ii) replaying old messages, and (iii) delaying message delivery or processing. COCA defends against these attack manifestations with a combination of request-processing authorization, resource management, and caching. To evaluate how effective these classical defense are, we simulated certain attacks. The results of those experiments for our local area network deployment of COCA are discussed in this subsection.

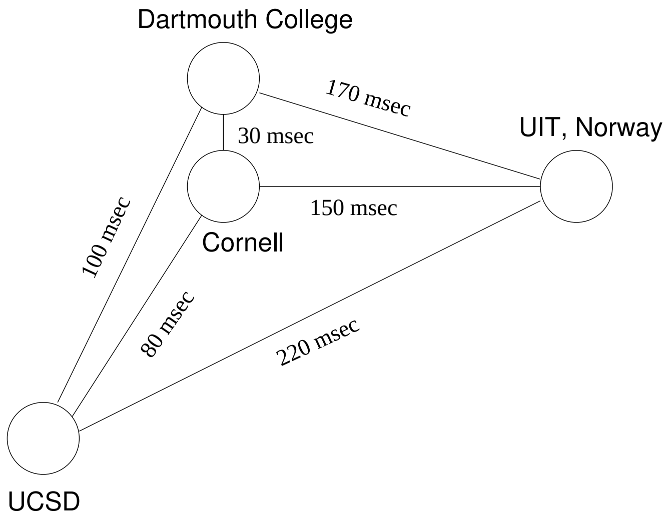

Figure 3: Deployment of COCA over the Internet with message delays between servers.

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

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

Message-Creation Defense. New messages sent by servers are not nearly as effective 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.

	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%	88.7%	87.4%
Other	0.8%	1.1%	0.6%

Table 4: Breakdown of costs for Query, Update, and proactive secret sharing (PSS) in Internet deployment.

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 the effectiveness of COCA's resource management mechanism, 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;[13] a *correct client* sends a request, awaits a response or a timeout[14]. 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 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

[13]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.

[14]The timeout is 1 second for Query and 2 seconds for Update.

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 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). Needless to say, 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, an increase in the rate of requests that the compromised client sends causes an increased delay for requests from the correct client. 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 segment almost identical wide ranges of samples for each rate measured. The worst case occurs when the request from the correct server 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

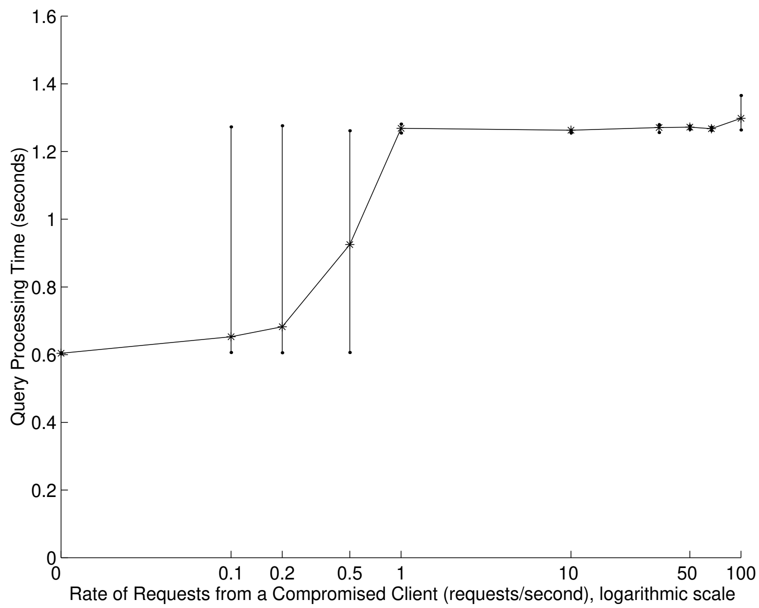

Figure 4: Performance of Query for a correct client when a compromised client makes requests at varying rates.

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.

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), the second segment of the curve begins. Throughout this 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 remain 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.

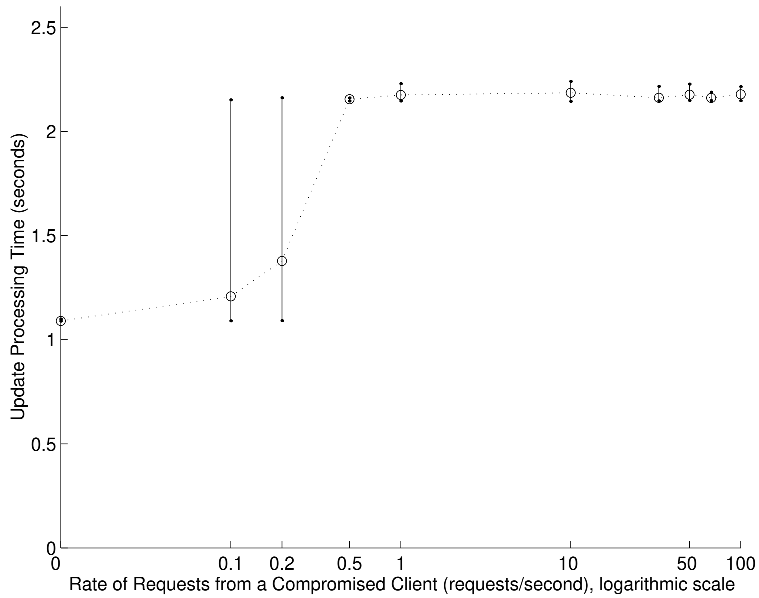

Figure 5: Performance of Update for a correct client when a compromised client makes requests at varying rates.

COCA's request-processing authorization mechanism starts saturating at 100 requests per second and thereafter the server would have diminished processing capacity to execute protocols for Query and Update.

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 do not consider replays of client requests in our experiments, because their impact on COCA 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. The curves of Figures 4 and 5 thus 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 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

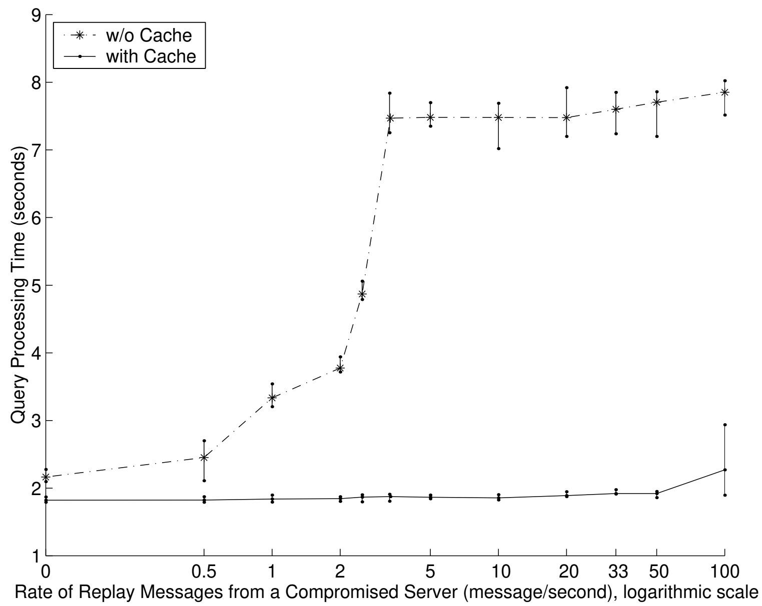

Figure 6: Performance of Query processing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

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 normal circumstances 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 added code to each server so that messages delivered to a client or server could be delayed a

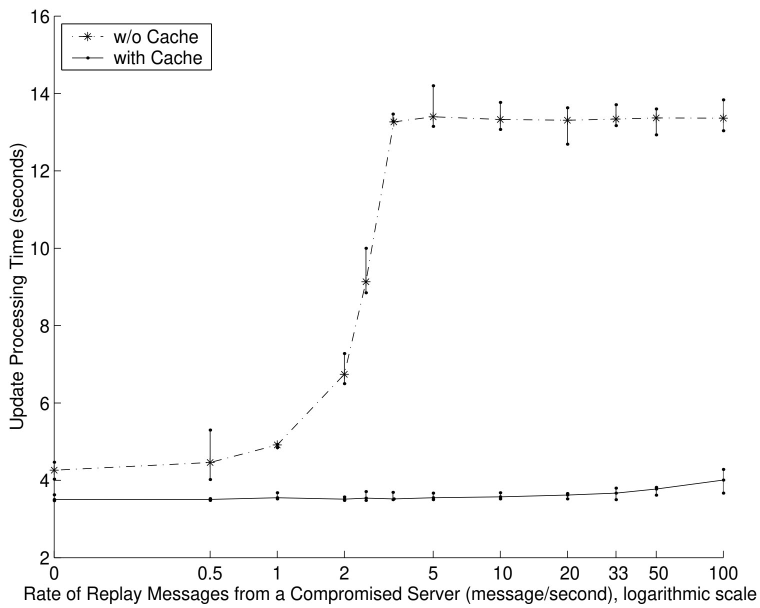

Figure 7: Performance of Update processing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

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.

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 normal circumstances assumptions N1 and N2 hold, and the optimized protocols require participation by p. A delay in messages from p thus delays the protocols.

The second phase is entered after the delay for p causes servers to suspect that normal circumstances assumptions N1 and N2 do not hold. These servers initiate redundant processing, creating additional delegates for in-

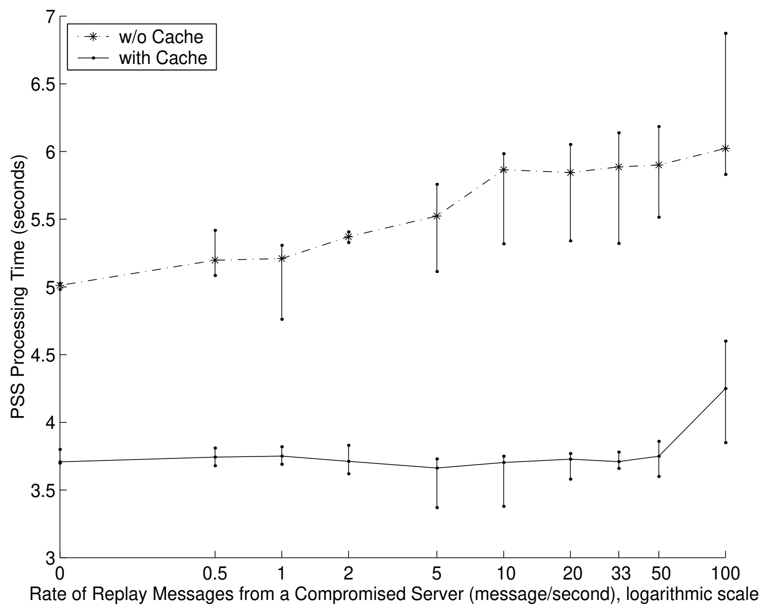

Figure 8: Performance of proactive secret sharing under the simulated denial of service attack from a compromised server: with cache vs. without cache.

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.[15]

The third phase—a plateau in response time—is reached when the delay for p is sufficiently high so that it imposes little load on other servers.

[15]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. This is because, in the case of proactive secret sharing, processing messages from server p involves some new (therefore not cached) expensive cryptographic operations, while, in other two cases, expensive cryptographic operations can be avoided due to caching.

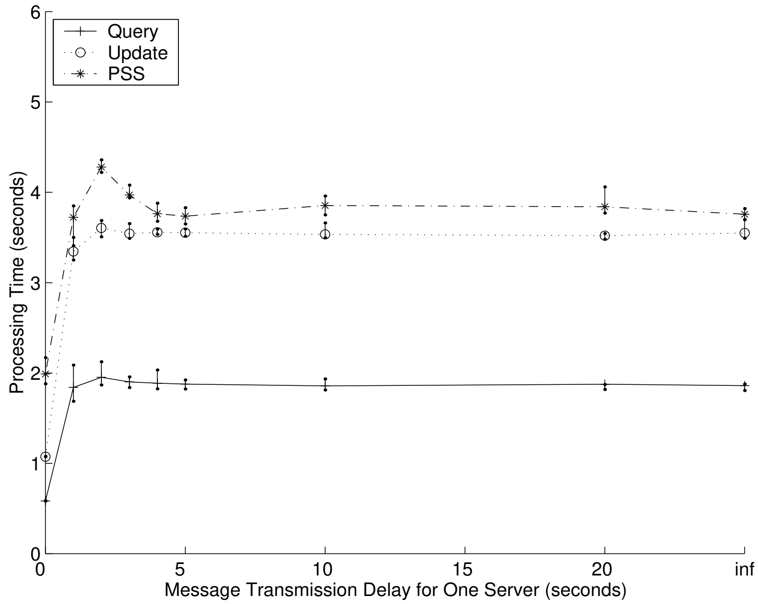

Figure 9: Performance of COCA vs. message delay for one server. Message delay of inf indicates the case where this one server is unavailable.

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 [65] 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 [66], 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. And, as discussed below, some vulnerability to

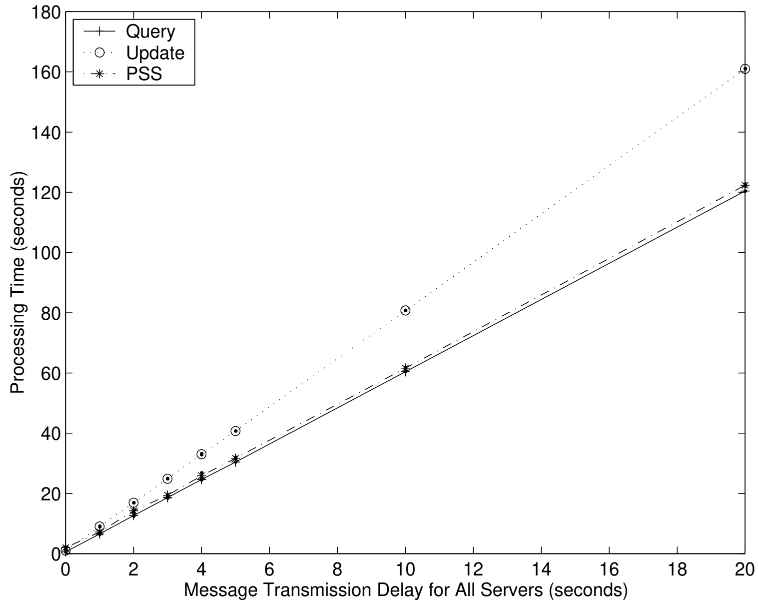

Figure 10: Performance of COCA vs. message delay for all servers.

denial of service attacks seems to be inherent. On the other hand, Ω does provide clients with key escrow operations, something that COCA does not currently support.[16]

Ω was built using middleware (called Rampart [63, 64]) that implements process groups in an asynchronous distributed system where compromised processors can exhibit arbitrary behavior. The Rampart middleware manages groups of replicas 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, which 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

[16] The same threshold decryption and blinding [10, 11, 12] that Ω uses for supporting this additional functionality would allow COCA to support these features too.

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 [5]. BFT extends the state machine approach [47, 69] to tolerate arbitrary failures in an asynchronous system. State machines are more powerful than the dissemination Byzantine quorum used by COCA. The additional power is not needed for implementing COCA's Query and Update but would be needed if the specification of Update were changed to take a less service-centric view of causality than COCA now takes. BFT also is extremely fast because, wherever possible, it uses MACs (message authentication codes) instead of public key cryptography. This replacement would also boost COCA's performance, although executing some public key cryptographic operations is inevitable in COCA for signing certificates and responses to clients.

As with COCA, BFT employs proactive recovery [6]. Even though BFT replicas do not store shares of a service private key, these replicas do need to refresh their key pairs and shared secret keys to combat mobile adversaries—secure co-processors are assumed for this task. BFT takes denial of service attacks into account and employs defenses similar to the mechanisms discussed for COCA in Section 4 [4]. A performance comparison would be interesting but no performance measurements are yet available.

The PASIS (Perpetually Available and Secure Information Systems) architecture [74] 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 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 [32]. 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 [50] 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 [50] 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 [49] for dishonest writers and employs a dissemination quorum system for honest writers.[17]

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 at IBM T.J. Watson Research Center implements Rabin's information dispersal algorithm [61] for storing and retrieving files [37, 26]. 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. Such stronger assumptions constitute a denial of service vulnerability—an attacker that is able to overload processors or clog the network can invalidate these assumptions and cause protocols to fail. Like with COCA, clients of e-vault communicate with the system through a

[17]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 fourth of servers. Recall, a dissemination quorum system tolerates one third of its servers being compromised.

single server (there called a gateway).

Cryptographic Building Blocks. COCA employs threshold cryptography [16, 17, 14, 15, 24] and proactive secret sharing [38, 36, 35, 23, 22] as building blocks. Because this prior work was not intended for systems in which (only) our Fair Links and Asynchrony assumptions hold, it was necessary to design new protocols for COCA [78, 77]. Implementations of threshold cryptography and proactive secret sharing schemes for stronger system models are reported in [2, 73, 19, 9].

Most previous work on public key infrastructure (e.g., [27, 72, 48, 41]) advocates off-line CAs, which issue certificates and certificate revocation lists (CRLs). Trade-offs associated with CRLs and related mechanisms are discussed in [68, 56, 44, 21, 51]. Stubblebine [71] 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 such a solution.

Alternatives to using an off-line CA include on-line certificate status checking (OCSP) [57, 56, 44] and on demand revocation lists [51]. These rely on some sort of trusted on-line service (a responder, a validation authority, and so on) and therefore our experience implementing and deploying COCA is directly applicable.

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 being an on-line CA, 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 compromised certificates is reduced. Marrying COCA with an off-line CA would achieve the advantages of both [48, 71, 57]. 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.

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. COCA serves as a vehicle to allow investigations into interactions between fault-tolerance and security mechanisms. Divide-and-conquer does not always apply, and COCA demonstrates that fault-tolerance and security—two crucial dimensions of trustworthiness—can be inseparable.

Naive application of fault-tolerance approaches like replication actually can increase a system's vulnerability to attacks:

- When a component's state is replicated, any secrets stored by that component are also replicated. Such replication increases the number of sites available to attackers for compromise.

- Protocols to coordinate replicas invariably require assumptions about processor speed and/or communications channels. The existence of these assumptions provides attackers with the opportunity to violate them.

COCA employs threshold cryptography and proactive secret sharing so that replication does not introduce vulnerabilities.

COCA, in composing mechanisms for fault-tolerance and security, implements a secure multi-party computation [75, 31, 3, 13]. 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.

Careful attention paid to the assumptions that characterize COCA's environment led to a system with inherent defenses to denial of service attacks. While additional denial of service defenses are described in Section 4, enumerating and countering specific attacks can be unsettling as a sole means of defense: What if some unanticipated attack is launched? Defenses based on weak assumptions are, by construction, accompanied by a characterization of the vulnerabilities—the assumptions themselves. And, by their very nature, weak assumptions are difficult to violate.

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 and Miguel Castro enabled us to sharpen and clarify the exposition. Li Gong and Cathy Meadows also provided helpful feedback on a draft of this paper. Discussions with Yaron Minsky and Zygmunt Hass were influential during the early stages of our investigations.

A Detailed Description of Protocols

This appendix gives details for the protocols described in Section 3.[18] 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* are parameters related to the request, and *cred* is credentials that authorize the request.

Clients use the following protocol to communicate with COCA.

[18] See [78] for a description of the proactive secret sharing protocol used by COCA.

1. To invoke Query for the certificate associated with name *cid*, client c composes a request:
$$\mathcal{R} = \langle \mathsf{query}, c, seq, cid, cred\rangle_c$$
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:
$$\mathcal{R} = \langle \mathsf{update}, c, seq, \zeta', \langle cid, key\rangle, cred\rangle_c$$

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 $\langle \mathcal{R}, \mathsf{done}\rangle_k$.

A.2 Threshold Signature Protocol

The following describes threshold signature protocol $threshold\_sign(m, \mathcal{E})$[19], 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 $\mathcal{E}$.
$$[\forall q.\ p \longrightarrow q:\ \langle \mathsf{sign\_request}, p, m, \mathcal{E}\rangle_p] \qquad \text{(i)}$$

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

[19] 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., [8, 34]). 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.

3. Server p periodically repeats step 1 until it receives partial signatures from a quorum of servers[20] (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.[21] 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 it 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 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:

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

3. Server p repeats step 1 until it receives the 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 query_response messages. Server p invokes $threshold\_sign(m, \mathcal{E})$, where m is $(\mathcal{R}, \zeta)$ and $\mathcal{E}$ is the 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:[22]

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

[20] 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.

[21] 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 [29, 28] that can reduce the cost using error correction codes.

[22] 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.

A.4 Update processing protocol

1. Upon receiving a request $\mathcal{R} = \langle \mathsf{update}, c, seq, \zeta', \langle cid, key\rangle, cred\rangle_c$ from a client c, server p first checks whether $\mathcal{R}$ is valid, based on the credentials *cred* provided. If it 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, and invokes $threshold\_sign(m, \mathcal{E})$, 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 $threshold\_sign(m, \mathcal{E})$, 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

[1] R. Baldoni, J.-M. Helary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 273–282, New York, NY USA, June 25–28, 2000.

[2] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS'99)*, pages 18–27, Singapore, November 1999.

[3] M. Ben-Or and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*, pages 1–10, Chicago, IL USA, May 2–4, 1988.

[4] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA, November 2000.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, LA USA, February 22–25, 1999. USENIX Association, IEEE TCOS, and ACM SIGOPS.

[6] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'00)*, San Diego, CA USA, October 22–25, 2000. USENIX Association, IEEE TCOS, and ACM SIGOPS.

[7] CCITT. Recommendation X.509: The directory-authentication framework, 1988.

[8] M. Cerecedo, T. Matsumoto, and H. Imai. 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):532–545, April 1993.

[9] CertCo, Inc. http://www.certco.com.

[10] D. Chaum. Blind signatures for untraceable payments. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology—Crypto'82, A*

Workshop on the Theory and Application of Cryptography, Santa Barbara, CA USA, August 23–25, 1982, pages 199–203. Plenum Press, New York, 1983.

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

[12] D. Chaum. Blinding for unanticipated signatures. In D. Chaum and W. L. Price, editors, *Advances in Cryptology—Eurocrypt'87, Workshop on the Theory and Application of Cryptographic Techniques, Amsterdam, The Netherlands, April 13–15, 1987, Proceedings*, volume 304 of *Lecture Notes in Computer Science*, pages 227–233. Springer-Verlag, 1988.

[13] D. Chaum and I. Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*, pages 11–19, Chicago, IL USA, May 2–4, 1988.

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

[15] Y. Desmedt. Some recent research aspects of threshold cryptography. In E. Okamoto, G. Davida, and M. Mambo, editors, *Information Security, The 1st International Workshop, ISW'97, Tatsunokuchi, Ishikawa Japan, September 17–19, 1997, Proceedings*, volume 1396 of *Lecture Notes in Computer Science*, pages 158–173. Springer, February 1998.

[16] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology—Crypto'89, the 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20–24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1990.

[17] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures (Extended Abstracts). In J. Feigenbaum, editor, *Advances in Cryptology—Crypto'91, the 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11–15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 1992.

[18] A. Doudou, B. Garbinato, and R. Guerraoui. Modular abstractions for devising Byzantine-resilient state machine replication. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nürnberg, Germany, October 16–18, 2000.

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

[20] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, pages 427–437. IEEE, October 12–14, 1987.

[21] B. Fox and B. LaMacchia. Certificate revocation: Mechanics and meaning. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 158–164. Springer, 1998.

[22] Y. Frankel, P. Gemmel, P. MacKenzie, and M. Yung. Optimal resilience proactive public-key cryptosystems. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, pages 384–393, Miami Beach, Florida, October 20–22, 1997. IEEE.

[23] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Proactive RSA. In B. S. K. Jr., editor, *Advances in Cryptology—Crypto'97, the 17th Annual International Cryptology Conference, Santa Barbara, CA USA, August 17–21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 1997.

[24] Y. Frankel and M. Yung. Distributed public key cryptosystem. In H. Imai and Y. Zheng, editors, *Public Key Cryptography, the 1st International Workshop on Practice and Theory in Public Key Cryptography, PKC'98, Pacifico Yokohama, Japan, February 5–6, 1998, Proceedings*, volume 1560 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1998.

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

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

[27] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed systems security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, Baltimore, MD USA, October 10–13, 1989. National Institute of Standards and Technology (NIST), National Computer Security Center (NCSC).

[28] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In N. Koblitz, editor, *Advances in Cryptology—Crypto'96, the 16th Annual International Cryptology Conference, Santa Barbara, CA USA, August 18–22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[29] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In U. M. Maurer, editor, *Advances in Cryptology—Eurocrypt'96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12–16, 1996, Proceedings*, volume 1233 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 1996.

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

[31] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual Conference on Theory of Computing, STOC'87*, pages 218–229, New York, NY USA, May 25–27, 1987. ACM.

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

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

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

[35] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public-key and signature schemes. In *Proceedings of the 4th Annual Conference on Computer Communications Security*, pages 100–110, Zurich, Switzerland, April 1–4, 1997. ACM.

[36] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology—Crypto'95, the 15th Annual International Cryptology Conference, Santa Barbara, CA USA, August 27–31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 1995.

[37] A. Iyengar, R. Cahn, C. Jutla, and J. Garay. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC'98)*, pages 123–135, Vienna, Austria and Budapest, Hungary, August 31–September 4, 1998. International Federation for Information Processing, TC11 : Security and Protection in Information Processing Systems.

[38] S. Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA, September 1995.

[39] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA USA, February 4–5, 1999. Internet Society.

[40] P. Karn and W. Simpson. Photuris session key management protocol. RFC 2522, March 1999.

[41] C. Kaufman. DASS: Distributed authentication security service. Request for Comments 1507, September 1993.

[42] S. T. Kent, D. Ellis, P. Helinek, K. Sirois, and N. Yuan. Internet infrastructure security countermeasures. Technical Report 8173, BBN, January 1996.

[43] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of International Conference on Principles of Distributed Systems (OPODIS'97)*, pages 61–76, Chantilly, France, December 10–12, 1997.

[44] P. C. Kocher. On certificate revocation and validation. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 172–177. Springer, 1998.

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

[46] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. In *Proceedings of the Annual Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97)*, pages 115–126, Cannes, French Riviera, France, September 16–18, 1997. ACM.

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

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

[49] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[50] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 51–58, West Lafayette, IN USA, October 20–22, 1998. IEEE Computer Society.

[51] P. McDaniel and A. Rubin. A response to "Can we eliminate revocation lists?". To appear in Proceedings of Financial Cryptography 2000 (FC'2000).

[52] C. Meadows. A formal framework and evaluation method for network denial of service. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 4–13, Mordano, Italy, June 28–30, 1999. IEEE Computer Society Press.

[53] C. Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 2000. To appear.

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

[55] J. K. Millen. Denial of service: A perspective. In F. Cristian, G. L. Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications 4*, pages 93–108. Springer, 1995.

[56] M. Myers. Revocation: Options and challenges. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 165–171. Springer, 1998.

[57] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet public key infrastructure online certificate status protocol (OCSP). Request For Comments 2560, June 1999.

[58] OpenSSL Project. http://www.openssl.org.

[59] R. Oppliger. Protecting key exchange and management protocols against resource clogging attacks. In B. Preneel, editor, *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*, pages 163–175, Leuven, Belgium, September 20–21, 1999. International Federation for Information Processing, Kluwer Academic Publishers.

[60] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th Annual Symposium on Principles of Distributed Computing (PODC'91)*, pages 51–59, Montreal, Quebec, Canada, August 19–21, 1991. ACM.

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

[62] T. Rabin. A simplified approach to threshold and proactive RSA. In H. Krawczyk, editor, *Advances in Cryptology—Crypto'98, the 18th Annual International Cryptology Conference, Santa Barbara, CA USA, August 23–27, 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 1998.

[63] M. K. Reiter. The Rampart toolkit for building high-integrity services. In K. P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5–9, 1994, Selected Papers*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1995.

[64] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

[65] M. K. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[66] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Ω key management service. *Journal of Computer Security*, 4(4):267–297, 1996.

[67] M. Richtel and S. Robinson. Several web sites attacked following assaults on Yahoo. *New York Times*, February 8, 2000.

[68] R. L. Rivest. Can we eliminate revocation lists? In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 1998.

[69] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

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

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

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

[73] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, Washington, D.C. USA, August 22–26, 1999. USENIX Association.

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

[75] A. C. Yao. Protocols for secure computation. In *Proceedings of the 23rd Symposium on Foundations of Computer Science (FOCS'82)*, pages 160–164, Chicago, IL USA, November 3–5, 1982. IEEE.

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

[77] L. Zhou. *Towards Building Secure and Fault-tolerant On-line Services*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY USA, January 2001. In preparation.

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