CODEX: A Robust and Secure Secret Distribution System

Michael A. Marsh, *Member, IEEE,* Fred B. Schneider, *Member, IEEE*

***Abstract*— CODEX (COrnell Data EXchange) stores secrets for subsequent access by authorized clients. It also is a vehicle for exploring the generality of a relatively new approach to building distributed services that are both fault-tolerant and attack-tolerant. Elements of that approach include: embracing the asynchronous (rather than synchronous) model of computation, use of Byzantine quorum systems for storing state, and employing proactive secret sharing with threshold cryptography for implementing confidentiality and authentication of service responses. Besides explaining the CODEX protocols, experiments to measure their performance are discussed.**

***Index Terms*— Distributed systems, Fault tolerance, Access controls, Client/server and multitier systems, Information storage**

1. Introduction

CODEX (COrnell Data EXchange) is a distributed service for storage and dissemination of secrets. It was designed to be one of the components in a secure publish/subscribe communications infrastructure, providing the support for storing secret keys used to encrypt published information objects and ensuring that (only) authorized subscribers retrieve those secret keys. Confidentiality, integrity, and availability of the secret keys being stored is crucial; server failures must be tolerated; and attacks must be thwarted.

The protocols to coordinate CODEX servers avoid a large class of vulnerabilities by making only weak assumptions about the execution environment. For example, correct operation of the protocols does not depend on assumptions about message delivery delays or processor execution times. Since denial of service attacks invalidate such assumptions about timing, we build into CODEX an intrinsic defense against certain denial of service attacks by designing for this *asynchronous model* of execution. In particular, the confidentiality, integrity and availability of secret keys stored by CODEX cannot be compromised by attacks to cause delays.

But adopting the asynchronous model brings challenges. Protocols for consensus and other replica-coordination problems arising in distributed systems require stronger assumptions [14]. CODEX does employ replication, so this gap must somehow be bridged. The solution employed by CODEX exemplifies what we are finding to be a general approach—carefully choosing a semantics for the service that skirts the need to do certain types of coordination but nevertheless can meet the needs of clients.

Embracing the asynchronous model also means that CODEX cannot offer real-time guarantees to clients. Availability is eventual. However, nothing about the CODEX protocols *per se* introduces unpredictable delays in processing client requests. Real-time bounds could therefore be derived for the case where the system and environment are not under attack. Admittedly, some clients require that responses always be generated in a timely manner—this is simply an unsatisfiable requirement if denial of service attacks can cause arbitrary slowdowns.

The design of CODEX also explicates a second set of technical problems that arise when moving beyond fault-tolerance to supporting attack-tolerance as well. That is the tension between employing replication and protecting confidentiality. Integrity and availability are enhanced by replication; confidentiality is not. In CODEX, besides client secrets (which are stored in encrypted form and therefore can safely be replicated), secrets are used to implement the service itself

- for encrypting the client secrets that CODEX is storing, and
- for authenticating CODEX responses to clients.

These service secrets cannot be stored in encrypted form, since they must be available for use by CODEX servers. So such secrets cannot be replicated.

To preserve confidentiality of secrets used for implementing confidentiality and authentication, CODEX employs secret sharing [38], [1] with proactive refresh [27], [25], [24], [17], [16], [41] in concert with threshold cryptography [8], [9], [6], [7], [18], [31]: secrets are split into *shares*; each secret share is periodically refreshed and stored by a separate server; and cryptographic operations are performed by first having each server compute a *partial result* using its local secret share and then combining the partial results to obtain what would have been computed if the cryptographic operation were performed using the original secret rather than the shares.

CODEX was motivated by and satisfies an application need, but the project was actually undertaken to address broader scientific questions. The COCA distributed certification authority [40] demonstrated that a fault-tolerant and attack-tolerant service could be implemented by embracing the asynchronous model and by employing proactive secret sharing in conjunction with threshold cryptography. We wondered whether COCA represented a singular point or did it instantiate a general recipe that could be repeated for other services too?

M. A. Marsh is with Institute for Advanced Computer Studies, University of Maryland.

F. B. Schneider is with Department of Computer Science, Cornell University.

Supported in part by AFOSR grant F49620-00-1-0198 and F49620-03-1-0156, National Science Foundation Grant 9703470, and Department of Defense CIPIA Grant F49620-01-1-0312. 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.

Building another system (CODEX) was expected to shed light on these questions about generality.

This paper thus should be seen not only as a discussion of how one might implement a service for storing and disseminating secrets but as an exercise in evaluating (and refining) a promising recipe for building services that are both fault-tolerant and attack tolerant. The CODEX system itself is implemented as approximately 42K lines of C++ code, most of which consists of general-purpose libraries designed to allow rapid development of new services, especially those employing quorum replication.[1]

We proceed as follows. In §2, assumptions made by CODEX about the execution environment are discussed. The client interface to CODEX is next explained in §3. Then, §4 gives a high-level view of the protocols that characterize CODEX (and its predecessor COCA); protocol details are given in Appendix A and Appendix B. CODEX performance is the subject of §5, and related work is described in §6. Section §7 contains some concluding remarks.

2. Assumptions about the Environment

CODEX makes only a few, rather weak assumptions about the environment in which it executes and about attackers. We believe these few assumptions constitute a realistic approximation of the hosts and links comprising today's Internet.

Communications links between hosts are presumed to be unreliable and insecure.

> **Insecure Links Assumption**: Messages in transit may be disclosed to, deleted, or altered by adversaries; new messages may be injected. But a message sent sufficiently often from one host to another will eventually be delivered.

The only weaker assumption we can imagine is to offer no guarantee that hosts can use links to communicate, but without communication it would be impossible for clients to coordinate with CODEX or with each other.

Under the Insecure Links Assumption, message delivery delays are potentially unbounded, which models network congestion and certain denial of service attacks. A second source of delay in CODEX arises because clients and servers are executed by hosts that have finite resources. Here, parallel activity (perhaps from a denial of service attack) means that message transit and process execution can be slowed arbitrarily. The situation is abstractly characterized by:

> **Asynchrony Assumption**: Message delivery and server computation times are unbounded.

Note, the Asynchrony Assumption is actually a non-assumption about timing.

Finally, the only assumption we make about hosts is that not too many fall under control of the adversary.

> **Compromised Hosts**: A host is either *correct* or *compromised*. A compromised host might deviate arbitrarily from its protocols and/or disclose information. But fewer than one-third of the hosts runing CODEX servers are assumed to be compromised at any time.

[1]The complete CODEX source is available for download from the CODEX homepage `http://www.umiacs.umd.edu/∼mmarsh/CODEX`

If all hosts run the same software then an adversary that compromises one replica will probably be able to exploit that same vulnerability at other replicas and compromise them too. A single exploit could then cause Compromised Hosts to be violated. One solution is to employ diversity, so that CODEX servers are not identical in their design or implementation and therefore do not have common vulnerabilities. For some system components, diverse implementations already exist. For other components, diversity must be created—here, automatic introduction of diversity during compilation, loading, or in the run-time environment is a promising approach.

3. CODEX Client Interface

CODEX binds secrets to names. Bindings are write-once—only a single value is ever bound to each name. The three[2] CODEX operations enable clients to manipulate and retrieve bindings:

- create introduces a new name;
- write associates a (presumably secret) value with a name;
- read returns the value associated with a name.

Having create and write be distinct—rather than a single compound operation—provides the flexibility to separate the administration of a secret from associating a value with that secret. We expect distinct principals will be concerned with these two kinds of operations, and we expect the operations to occur at different times.

Clients of CODEX can expect the following security properties to hold:

> **CODEX Availability**: Authorized invocations of create, read and write that are not concurrent with other invocations involving the same CODEX name, if repeated sufficiently often, cause the corresponding operations to be performed.

> **CODEX Confidentiality**: Executing read is the only way to learn a value that CODEX stores.

> **CODEX Integrity**: Executing write, giving a name that does not yet have a value associated, is the only way to bind a value to that name.

Authorization policies specify which invocations of operations are performed and which are ignored by CODEX. A client p presents credentials $\mathcal{C}_\theta(p, N)$ when invoking an operation θ for a name N; the authorization policy $\mathcal{P}_\theta(N)$ for operation θ and name N defines a relation $\models_\theta$ such that

$$\mathcal{C}_\theta(p, N) \models_\theta \mathcal{P}_\theta(N)$$

holds if credentials $\mathcal{C}_\theta(p, N)$ are sufficient for the operation to be performed. CODEX does not include an implementation for $\models_\theta$ nor does it fix a representation for credentials. However, contemporary authorization engines, like SDSI[35], KeyNote[2] or the work of Hayton *et al.* [23], do provide such implementations and could well be incorporated into CODEX.

[2]The lack of any sort of key deletion operation in CODEX is deliberate. After a key has been deleted, information encrypted using that key becomes unavailable, which we saw as inconsistent with the archival nature of the publish/subscribe system CODEX was designed to support.

1) Client p sends to CODEX the invocation message $M_{C(N)}$: $\langle \textsf{create}, N, \mathcal{C}_C(p,N), \mathcal{P}_W(N), \mathcal{P}_R(N)\rangle_p$
2) Client p awaits confirmation message $\hat{M}_{C(N)}$: $\langle \textsf{bound}, M_{C(N)}\rangle_{\textsf{CODEX}}$

Fig. 1. Client protocol for the create operation.

1) Client p sends to CODEX the invocation message $M_{W(N)}$: $\langle \textsf{write}, N, \mathcal{C}_W(p,N), E(s), \hat{M}_{C(N)}, \Pi(s,p)\rangle_p$
2) Client p awaits confirmation message $\hat{M}_{W(N)}$: $\langle \textsf{stored}, N, sig(M_{W(N)})\rangle_{\textsf{CODEX}}$

Fig. 2. Client protocol for the write operation.

CODEX associates a separate authorization policy with each of the three operations it supports. $\mathcal{P}_C(N)$ controls which clients can create a new CODEX name N. Having such an authorization policy helps defend against resource-exhaustion attacks—for example, the CODEX name space might be partitioned into groups, with a unique client authorized to do create for CODEX names in a group. $\mathcal{P}_W(N)$ governs which clients can write the value that CODEX binds to N. And $\mathcal{P}_R(N)$ governs which clients can read that value.

To invoke a CODEX operation, a client p opens a TCP connection to CODEX and sends an *invocation message*; invocation messages for unauthorized or otherwise ill-formed operations are ignored by CODEX. Upon completing an operation, CODEX replies to p with a *confirmation message*. If no confirmation message is received after a respectable interval, then the client may resend the invocation message; receipt of duplicate invocation messages does not cause problems for CODEX, because the three CODEX operations are idempotent.

Invocation messages are digitally signed by the client invoking the operation; confirmation messages are digitally signed by CODEX and contain either the invocation message m or its digital signature $sig(m)$, so that a client can ascertain that the response is not a replay.

3.1 *create*

Figure 1 gives the invocation message $M_{C(N)}$ and confirmation message[3] $\hat{M}_{C(N)}$ for a create operation that is on behalf of client p and that defines a new name N to have write policy $\mathcal{P}_W(N)$ and read policy $\mathcal{P}_R(N)$.

Each CODEX name can be defined only once. Duplicate create invocation messages bring copies of the same confirmation message; subsequent or concurrent create invocation messages for a given name but with different authorization policies are considered ill-formed and are ignored.

3.2 *write*

Figure 2 gives the invocation message $M_{W(N)}$ and confirmation message $\hat{M}_{W(N)}$ for a write operation by a client p intending to associate value s with name N. Keyword write, name N, and credentials $\mathcal{C}_W(p,N)$ in $M_{W(N)}$ should not be surprising.

Value s is being sent in encrypted form (fourth field of $M_{W(N)}$) so passive wiretappers that intercept and read $M_{W(N)}$ are unable to compromise CODEX Confidentiality.

The inclusion of create confirmation message $\hat{M}_{C(N)}$ in $M_{W(N)}$ (fifth field) ensures that name N has been created before p attempts to associate a value with N. Even though a client might be careful to send the write invocation message after sending the create invocation message, CODEX will not necessarily receive this pair of messages in that order (due to the Insecure Links Assumption coupled with message retransmissions). By requiring that a write invocation message contain a create confirmation message, the ordering pathology is avoided.

$\Pi(s,p)$, the final field of $M_{W(N)}$, is a non-malleable proof[4] that p knows plaintext s (as opposed to knowing only encrypted plaintext $E(s)$, which is already the fourth field of $M_{W(N)}$). Eliminate the requirement that $\Pi(s,p)$ appear in $M_{W(N)}$ and a malicious client q could then read s as follows: Intercept $M_{W(N)}$; copy $E(s)$ from $M_{W(N)}$ into a write invocation message for some new name N_q that q creates and for which $\mathcal{P}_R(N_q)$ includes q; then, perform a read naming N_q. Such attacks are ruled out if write invocation messages require demonstrations of plaintext knowledge, because now attacker q is unable to construct needed final field $\Pi(s,q)$ due to the non-malleability of $\Pi(s,p)$.

Because CODEX names are write-once, subsequent write invocation messages are considered ill-formed and ignored if they attempt to bind different values to a name. Also, concurrent write invocation messages containing the same name but different values are considered ill-formed and ignored.

3.3 *read*

The confirmation message for a read must convey the value $val(N)$ that CODEX binds to a name N, but not as cleartext or else a passive wiretapper intercepting that message would be able to compromise CODEX Confidentiality.

A naive solution would be to encrypt $val(N)$ using the public key of the destination client (say) p. But CODEX is implemented by a set of servers, and according to the Compromised Hosts Assumption no server can be trusted to have $val(N)$ as cleartext. So this naive solution would require a protocol to transform $E(val(N))$ (which is what CODEX servers store) into $val(N)$ encrypted by client p's public key, without plaintext becoming available during intermediate steps. Such *re-encryption protocols* (e.g., [26], [39]) unfortunately involve considerable communication overhead; the cost makes them infeasible for use here.

Therefore, CODEX employs blinding[4] to protect the confidentiality of $val(N)$ while in transit. A client performing a read includes $E(b_p)$ in the invocation message, where $E(b_p)$ is a fresh random *blinding factor* b_p encrypted with

[3]Henceforth, we write $\langle m\rangle_p$ to denote a message m digitally signed by client p and write $E(v)$ to denote the result of encrypting a value v according to the CODEX public key.

[4]A proof $\Pi(s,p)$ that p has knowledge of s is defined to be *non-malleable* if that proof cannot be transformed into a proof $\Pi(s',p)$ that p knows some other secret s' or transformed into a proof $\Pi(s,q)$ that some other principal q knows secret s. The scheme used by CODEX is described in Appendix B.

1) Client p sends to CODEX the invocation message
$M_{R(N)}: \langle \mathsf{read}, N, \mathcal{C}_R(p, N), E(b_p), \Pi(b_p, p)\rangle_p$
2) Client p awaits confirmation message
$\hat{M}_{R(N)}: \langle \mathsf{blind}, N, val(N) \times b_p, sig(M_{R(N)})\rangle_{\mathsf{CODEX}}$

Fig. 3. Client protocol for the read operation.

the CODEX public key. By construction, for a homomorphic cryptosystem such as RSA,

$$E(val(N)) \times E(b_p) = E(val(N) \times b_p)$$

holds and, by design, each CODEX server stores $E(val(N))$. Thus, each CODEX server can compute $E(val(N) \times b_p)$ from information available to it (*viz* $E(val(N))$ and $E(b_p)$) and can then employ threshold decryption to obtain $val(N) \times b_p$ without ever materializing $val(N)$ or b_p; so the confirmation message sent to p contains $val(N) \times b_p$, which is undecipherable without b_p. And client p (which does know b_p) recovers $val(N)$ by dividing $val(N) \times b_p$ by b_p.

Figure 3 details invocation message $M_{R(N)}$ and confirmation message $\hat{M}_{R(N)}$ for a read operation by a client p to retrieve the value CODEX binds to a name N. Given the preceding discussion, the presence of keyword read, name N, credentials $\mathcal{C}_R(p, N)$, and encrypted blinding factor $E(b_p)$ in $M_{R(N)}$ should not be surprising.

To understand the need for including knowledge of plaintext proof $\Pi(b_p, p)$ in $M_{R(N)}$, consider a client q that is authorized to read N_q but is under control of an adversary. And suppose there were no requirement that clients provide knowledge of plaintext proofs in invocation messages. Here is an attack that allows q to determine $val(N)$: By intercepting an invocation message from a client p, client q learns $E(b_p)$; by intercepting the corresponding confirmation message, client q learns $val(N) \times b_p$. Client q now initiates read, twice, naming N_q:

- first, q sends encrypted blinding factor $E(b_p)$ (obtained by interception); $val(N_q) \times b_p$ is returned
- second, q sends encrypted blinding factor $E(b_q)$ for some fresh b_q known to q; $val(N_q) \times b_q$ is returned.

Client q can now recover b_p by computing:

$$\frac{val(N_q) \times b_p}{(val(N_q) \times b_q)/b_q}$$

This knowledge of b_p, then allows client q to calculate $val(N)$ from $val(N) \times b_p$ (intercepted earlier). Note, however, that q would be unable to provide $\Pi(b_p, q)$ for its first read invocation, so this attack is foiled by requiring read invocation messages to contain knowledge of plaintext proofs for encrypted blinding factors.

4. COCA Redux: CODEX as a Distributed Service

The protocols used by CODEX resemble those in COCA, since a reason for building CODEX was to explore how broadly applicable those COCA protocols are. Grouped according to function, here is a high-level description of the protocols.

4.1 Coordinating Server Replicas

COCA and CODEX both employ server replication to ensure availability; a Byzantine quorum system [28] stores the service state. Each CODEX operation is implemented as a sequence of *service steps*, where a service step involves the servers comprising some *quorum*. The same quorum is not necessarily involved in all service steps for a given operation, because quorums are defined so that the intersection of any two quorums contains a correct server—when performing a service step Σ, results of all previous service steps are thus available from the correct servers in the quorum performing Σ. For a system with $3t+1$ servers, as many as t compromised servers can be tolerated if the set of quorums comprises all sets containing $2t + 1$ servers.

The service steps comprising an operation and the existence of quorums is hidden from CODEX clients. Besides simplifying client interactions with CODEX, hiding such internal details allows server cryptographic keys to be changed periodically, a powerful defense (as discussed below in §4.3).

CODEX internals are hidden (as also done in COCA) by employing a *delegate*—itself a CODEX server—to receive the invocation message from a client and then to orchestrate execution of the operation by initiating the various service steps. The delegate is also responsible for constructing the confirmation message that is sent back to the client when operation execution is finished; responses received from servers are combined to form this response.

A single delegate could be compromised. Clients therefore send each invocation message to $t+1$ servers, recruiting $t+1$ delegates so that at least one is correct. A response from a correct delegate will be correct. Moreover, because delegate responses include cryptographically-secure information obtained from a quorum of services, clients are able to identify and reject corrupted responses from compromised delegates. (The method for doing this—self-verifying messages—is sketched below.) All responses from correct delegates will thus be consistent; any responses from a compromised delegate will either be consistent with the correct response or will be detectable.

The existence of $t + 1$ delegates leads to considerable duplication of server effort, but idempotence of service steps ensures these executions do not interfere with each other. Still, there is the matter of performance, as each delegate forwards the same requests to all servers, waits for (the same) responses, and so on. Fortunately, an optimization employed in COCA applies to CODEX as well.

- Servers cache copies of the response they compute for a given request, so a server only has to do the real work once and can reply to a duplicate request with a cached response.
- Most of the delegates delay before starting, and each delegate immediately terminates its activity if ever it receives evidence[5] that another delegate has sent a confirmation message back to the client. A delegate will see evidence because, being a server, it is also processing delegates'

[5]Only messages that cannot be forged constitute evidence that may be used to cause delegate termination.

requests on behalf of that same client's t other invocation messages.

Each service step is performed by a subset of the CODEX servers; not by all servers. States at correct servers could thus diverge, and replica coordination becomes tricky. In COCA, which supports operations to read and update values associated with names, server states include unforgeable integrity checks and unforgeable ordered labels. When a COCA operation is performed by a quorum, the state of one server in the quorum is selected—specifically, a state is selected that satisfies the integrity checks and has the largest label. This selection criterion yields the most recent correct state, because every pair of quorums intersects and every operation is performed by some quorum.

In CODEX, the semantics of operations provides the basis for a simpler way to determine which server's state is most recent for any given CODEX name. This is because the state-altering CODEX operations—create and write—are performed only once for a given name, and the create must precede the write. So a state in which a name N has been defined is more recent than a state where N has not been defined, and a state in which some value is bound to N is more recent than a state in which no value is bound to N. Moreover, by including the create confirmation message in a write invocation (it is the fifth field; see Figure 2), a server that was not involved in performing the create for a given name but that is involved in processing a write for that name always receives the justification it needs to perform a create retroactively for that name.

The possibility does exist in CODEX for concurrent invocation of conflicting operations with a given name, such as multiple create operations (but with differing authorization policies) or multiple write operations (but with differing values). CODEX makes no guarantees about termination for conflicting create and write operations, pushing the problem onto CODEX clients. By transferring this burden to clients, CODEX avoids an unsolvable agreement problem (given the Asynchrony Assumption). So CODEX clients must synchronize with each other and/or partition the name space to ensure that conflicting operations are not executed concurrently. Authorization policies $\mathcal{P}_C(N)$ and $\mathcal{P}_W(N)$ provide the means to enforce name-space partitioning.

Finally, some defense is needed against compromised clients or servers, since they might not follow the CODEX protocols and might send bogus messages in an effort to subvert CODEX. CODEX employs the same defenses here as COCA. All CODEX messages are constructed to be *self-verifying*, which means the receiver of a message m has a *validity check* to determine that information conveyed in m is not a replay and is consistent with the CODEX protocols. A message that passes the check is said to be *valid*. Receivers ignore self-verifying messages that are not valid, effectively transforming Byzantine server failures to message loss.

Typically, a message is made self-verifying by adding cryptographically-protected information, such as a digital signature (perhaps even a threshold digital signature) or a cryptographic proof of plaintext knowledge. Sometimes a validity check will embody inferences involving messages from multiple distinct senders. One example is that if $t + 1$ servers attest to a statement P, then at least one correct server has, so P must hold and a message attesting to P should pass the validity test; another example is that if a quorum of servers attest to a statement then it is safe to conclude that no quorum of servers would attest to the negation of that statement.

4.2 Secure Links from Insecure Links

In CODEX, as in COCA, repeated message retransmission is used to overcome message loss admitted by the Insecure Links Assumption. Repeated retransmission of a given message is ended once the sender has been notified of successful receipt. This notification is usually signaled in a subsequent message (sometimes, but not always, an acknowledgment) from the receiver or from some other process that has received a message (directly or indirectly) from the receiver.

Some CODEX protocol steps require that a message be conveyed to any set comprising $AckNo$ out of the $3t + 1$ CODEX servers. This is implemented by executing $3t + 1$ repeated sends in parallel, and terminating all once responses have been received from $AckNo$ servers.

Confidentiality of message contents is implemented in COCA by encryption; CODEX in addition uses blinding, for the performance reasons outlined in §3.3. Receivers detect message alteration in both COCA and CODEX by employing digital signatures.

4.3 Servers and Service Authentication

In CODEX, as in COCA, each server has a public/private key pair, with the public key known to all servers (hence, known to all delegates). Delegates can thus authenticate responses from servers and determine when responses have been received from a quorum. Clients do not know server public keys. This allows server private keys to be changed without incurring an obligation to inform clients of the corresponding new public keys.

The private key of a server is not only changed when server compromise is detected but it is also changed periodically. Such periodic key refresh is known as *proactive security*, since it anticipates and defends against undetected server compromise as well as detected server compromise.

Recall from §3, confirmation messages are signed by CODEX and secrets stored by CODEX are encrypted using its public key. The corresponding private key is shared by the n CODEX servers using an $(n, t + 1)$ secret sharing scheme, so no CODEX server has to be trusted with that private key.[6] Threshold cryptography is then employed to generate signatures on confirmation messages and to decrypt content that was encrypted under the CODEX public key. Specifically, a delegate recruits $t + 1$ CODEX servers to each generate partial cryptographic results using its share; these results are then combined by the delegate. A set of $t + 1$ servers, by assumption, includes one that is not compromised, so each threshold cryptographic operation is performed only if

[6]With an $(n, t + 1)$ sharing, there are n shares, any subset of size $t + 1$ suffices for recovering the secret, but nothing about the secret can be learned from a smaller subset.

some correct server has received sufficient evidence to justify executing the operation.

An adversary must know at least $t+1$ shares in order to construct the CODEX private key. Whereas the Compromised Hosts Assumption rules out the adversary controlling $t+1$ servers, it does not rule out the adversary compromising one server and learning the CODEX private key share stored there, being evicted, compromising another, and ultimately learning $t+1$ shares. To defend against such *mobile virus attacks*, both COCA and CODEX employ the APSS [41] proactive secret sharing protocol. This protocol is periodically executed, each time generating a new sharing of the private key but without ever materializing the private key at any server. Because older secret shares cannot be combined with new shares, a mobile virus attack would succeed only if it is completed in the interval between successive executions of APSS, and this interval can be as short as a few minutes. (Executions of APSS measured by Zhou *et al.* take a few seconds [40].)

5. Performance Measurements

Performance measurements of CODEX were made both for a LAN deployment and for an Internet deployment. Both deployments comprised four servers running on separate hosts, so the system was capable of tolerating a single compromised server (i.e., $t=1$ is being assumed). We also assumed for our experiments that the first delegate that a client contacted was correct, which we modeled by deploying a single delegate (rather than $t+1$ delegates) on one of the hosts running a CODEX server.[7] Our measurements were performed using the Unix `getrusage` system call, which has an inherent granularity of 10ms; values presented are the means and RMS variances of the distributions.

ElGamal [12] is used in the prototype for encryption, and RSA [34] is used for digital signatures.[8] The public moduli for RSA and ElGamal are 1024 bits. Private keys are split into five shares—four are randomly generated and the fifth constrains the sum of all the shares to be the shared secret. Each server receives three of the four random shares and the fifth "public" constraint share.[9]

In the CODEX prototype, secure communication is established using the OpenSSL implementation of the TLS version 1 protocol. Connections between servers are maintained for as long as possible; the impact on protocol execution times caused by using these secure links is thus minimal.

We conjectured that modular exponentiations for cryptographic operations would dominate the most expensive CODEX operations, so we decomposed the cost of cryptographic operations accordingly. To simplify the exposition, costs are normalized to T_{sig}, the cost of performing one exponentiation with an exponent on the order of the size of the public modulus.

Message Signing: This operation involves one exponentiation with an exponent on the order of the size of the public modulus. The cost for message signing is thus, by definition, T_{sig}.

Partial Signatures: This operation involves one exponentiation but the size of a threshold RSA share is approximately twice that of the public modulus, so that exponentiation takes twice as long. The exponentiation must be done for each of the four shares held by a server. We therefore expect cost of computing a partial signature to be approximately $8T_{sig}$.

Partial Decryption: This operation is similar to partial signature generation, except that threshold ElGamal does not require shares larger than the public modulus. The expected time for a partial decryption is thus approximately $4T_{sig}$.

$\Pi(s,p)$ **Verification:** This operation involves two exponentiations, one approximately the size of the public modulus and the other using a 160-bit (SHA-1) hash output: an approximate cost of $1+160/1024$ with our normalization. For a delegate, this computation must be done twice, since the delegate also receives the request as a server. Therefore the cost of checking $\Pi(s,p)$ proofs is approximately $2.3T_{sig}$ for a delegate and $1.2T_{sig}$ for other servers.

DLProof Generation: When using ElGamal, a simultaneous discrete log proof is needed to demonstrate that a partial decryption result is correct. Generating this proof requires two exponentiations and must be done for each of the four shares held, so the cost for generating a simultaneous discrete log proof is approximately $8T_{sig}$.

DLProof Verification: Verifying a simultaneous discrete log proof involves four exponentiations, where two use hash results as exponents. The hash algorithm (SHA-1) has a 160-bit output, so the normalized cost of each verification is $2+2\cdot(160/1024)$. Each server must perform this verification on the partial results from five shares when examining the supporting evidence for a read confirmation message, which costs $11.5T_{sig}$. The delegate must also verify the proofs from the partial results of four shares contributed by each of the two $(t+1)$ servers needed to form a threshold decryption, or eight additional verifications costing $18.5T_{sig}$, for a total cost to the delegate of $30T_{sig}$ for 13 verifications.

From these individual costs and the protocol details (see Appendix A), we can predict the total cost of each CODEX operation. Note, the cost of checking a signature is negligible, so this is ignored in the accounting that follows. A create operation costs[10] each server roughly $9\ T_{sig}$. A write operation

[7]The case where the first delegate contacted is compromised is no different for CODEX than for COCA. Since the performance implications of this were explored extensively in the evaluation of COCA, making the simplifying assumption here of a correct delegate seemed defensible.

[8]A suitable proof of plaintext knowledge for RSA was not known to us at the time we ran these experiments or we would have used RSA for encryption as well as for digital signatures. In fact, a variant of the Fiat-Shamir identification protocol can be used for proofs of plaintext knowledge [30], [21].

[9]Here, we follow the convention of [31], in which all private shares are generated in the same way and are independent of one another. A simpler scheme would generate only four shares, where the fourth adds the necessary constraint to the three random shares and each server receives only three shares, but the security of such a sharing has not been proved.

[10]Each server signs the request as part of its ACCEPT in step 3a, which costs T_{sig}, and each server generates a partial signature in step 4b, which costs $8T_{sig}$.

TABLE 1

PERFORMANCE OF CODEX OVER A LAN.

Operation	Total CPU Time (ms)	Non-Idle Time (ms)	Idle Time (ms)
create	172.8 ± 4.1	170.8 ± 5.1	7.0 ± 4.0
write	241.6 ± 4.7	239.3 ± 6.1	7.4 ± 4.8
read	1055 ± 5.3	1052 ± 7.3	8.0 ± 5.3

The statistics for each of the measurements come from 110 requests. All times are measured on the delegate.

will cost about 12.3 T_{sig} for the delegate[11] and 11.2 T_{sig} for other servers[12]. A read operation will cost the delegate[13] roughly 53.3 T_{sig} and other servers[14] 33.8 T_{sig}. In the absence of other significant time costs, we would then expect (for the delegate) write to take about 1.37 times as long as create and expect read to take about 5.92 times as long as create. This is consistent with actual measurements reported below.

5.1 LAN Deployment

The LAN deployment of our CODEX prototype comprised four dual-Pentium III systems (1130MHz processors) running Linux. Round-trip times for ICMP echo packets typically measured well under 1ms, making network delays unobservable. The hosts and the network were relatively quiescent during the experiment. The client was executed on a separate machine; its processing and latency times are not included in our measurements.

Mean execution times measured for CODEX create, write, and read operations are shown in Table 1, and the fractions of time spent performing various actions are shown in Table 2. To minimize the impact of network latency and process scheduling on the values reported in Table 2, time spent on cryptographic operations is compared to the non-idle time rather than the total time. Observe that a CODEX read takes the longest of the three CODEX operations and the cost of read is dominated by creating and verifying proofs of correct partial decryption, as expected. The other significant processor time cost for read is computation of partial cryptographic results, also in agreement with predictions.

Table 3 gives direct comparisons with predictions of the costs for the various cryptographic operations. All measured ratios are consistent with our predictions, so we feel confident that modular exponentiations are indeed the dominant cost of the CODEX protocols in a LAN deployment.

An adversary can launch a number of attacks on the service. We consider two, both attempted denials of service:

[11]The delegate verifies $\Pi(s,p)$ proofs in steps 1 and 3, costing 2.3 T_{sig}, generates signatures in steps 3a and 5a, for a combined cost of 2 T_{sig}, and generates a partial signature in step 6b, which costs 8 T_{sig}.

[12]A non-delegate server only has to verify one $\Pi(s,p)$ proof, in step 3, so the verification cost is now only 1.2 T_{sig}.

[13]The delegate verifies $\Pi(b_p,p)$ proofs in steps 1 and 2, costing 2.3 T_{sig}. In step 2a it performs a partial decryption (4 T_{sig}), generates a discrete log proof (8 T_{sig}), and generates a signature (T_{sig}). It verifies 13 separate discrete log proofs in steps 3 and 3a, for 30 T_{sig}, and generates a partial signature in step 3b costing 8 T_{sig}.

[14]Only one $\Pi(b_p,p)$ proof is verified by a non-delegate server, so that cost is reduced from 2.3 T_{sig} to 1.2 T_{sig}. Similarly, only five discrete log proofs need to be verified (the others are performed by the delegate in order to assemble the response and evidence), reducing the DLProof verification cost to 11.5 T_{sig}.

TABLE 2

COSTS OF OPERATIONS FOR CODEX OVER A LAN.

	create	write	read
TLS	0.012 ± 0.025	0.011 ± 0.021	0.006 ± 0.006
Message Signing	0.104 ± 0.027	0.144 ± 0.023	0.017 ± 0.004
Partial Signature	0.818 ± 0.040	0.590 ± 0.018	0.130 ± 0.004
Partial Decryption			0.071 ± 0.003
$\Pi(s,p)$ Verification		0.212 ± 0.026	0.048 ± 0.005
DLProof Generation			0.130 ± 0.003
DLProof Verification			0.593 ± 0.007
Other	0.043 ± 0.045	0.035 ± 0.031	0.007 ± 0.008

The values shown are fractions of the non-idle time spent on the operation. The statistics for each of the operations come from 110 requests. All times are measured on the delegate.

TABLE 3

RATIOS OF TIME SPENT IN VARIOUS CRYPTOGRAPHIC OPERATIONS RELATIVE TO MESSAGE SIGNING.

	Observed Ratio	Predicted Ratio
Partial Signature	6.8 ± 1.3	8
Partial Decryption	3.6 ± 0.7	4
$\Pi(s,p)$ Verification	2.5 ± 0.5	2.3
DLProof Generation	6.8 ± 1.3	8
DLProof Verification	29.3 ± 5.4	30

Observed quantities are averaged over the operations in which they are used.

- an attack that increases message latencies between servers
- an attack that decreases CPU cycles available on servers.

For the first class of attacks, increased message latencies were simulated by modifying the CODEX binary so that message delivery could be delayed in a controlled way. We then simulated having one link under attack and then having two links under attack. (Recall, the client delegate is never attacked in our experiments.) The performance of CODEX under these attack scenarios is shown for create and write in Fig. 4; there was little point in measuring the performance of read, because it requires participation of only 2 (i.e., $t+1$) servers so delaying 2 out of 4 servers does not delay completion of this operation.

The horizontal axis in the graph of Fig. 4 is the fixed latency (in seconds) added to message delivery for links under attack; the vertical axis is the time taken to process a request. Notice that CODEX performance does not degrade when only a single link is under attack. This is because a 4 server CODEX system can tolerate a single compromised host. But when two links are being attacked, any request requiring $2t+1$ (i.e., 3) responses will be affected by the added latency—each added second of latency adds one second to the processing time for such operations. So link latencies only affect rounds of communications requiring $2t+1$ responses, and we conclude:

- The threshold signatures in create and write are unaffected.
- The create operation requires one round of communications with $2t+1$ servers (the forwarding of the request and receipt of ACCEPT messages), so it is affected.
- The write operation requires two rounds of communications with $2t+1$ servers (one for $M_{W(N)}$ and ACCEPT, and another for VERIFY and VERIFIED), so it is delayed twice as much as create.

We next simulated an attack that steals CPU cycles from

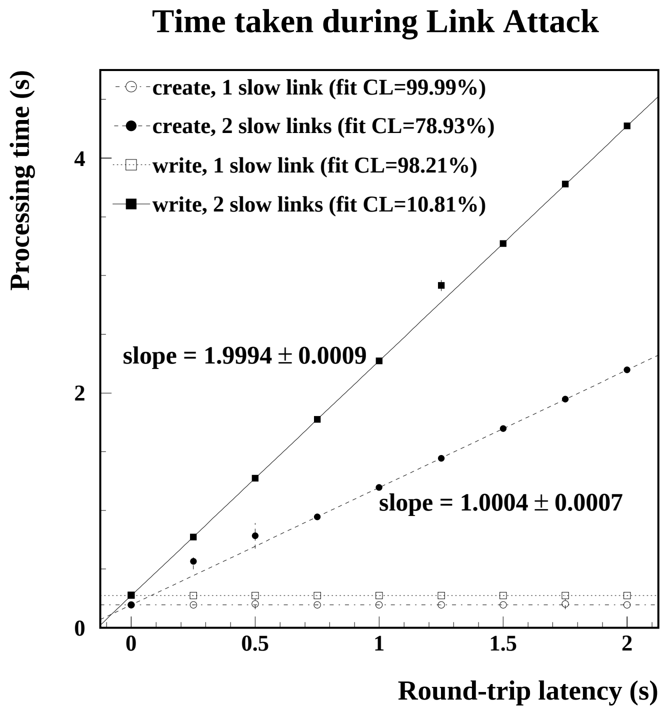

Fig. 4. Processing time for requests as a function of the effectiveness of an attack against the network. Times are measured by the wall clock. Each point represents fifty requests with standard-deviation error bars marked. Fits and confidence levels (CL) are shown, with the one-link points fit to a constant and the two-link points fit to a line.

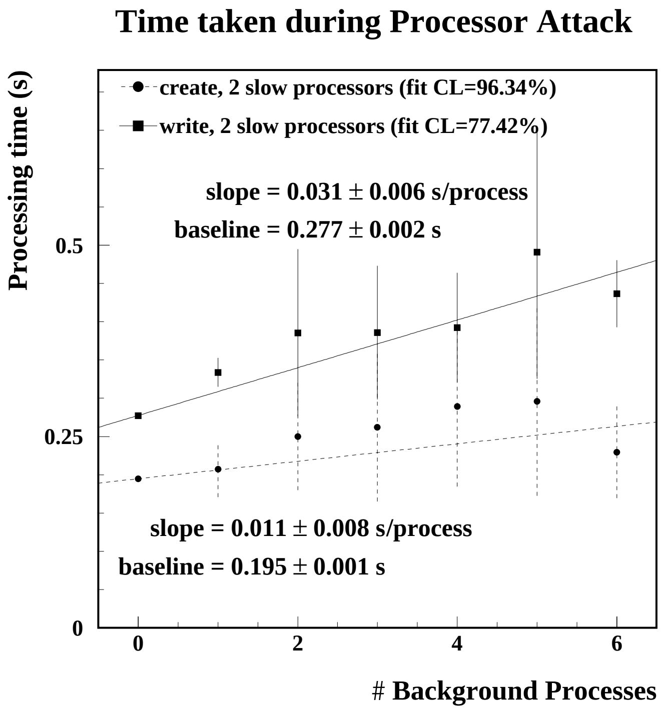

Fig. 5. Processing time for requests as a function of the effectiveness of an attack against servers. Times are measured by the wall clock, and each point represents fifty requests with standard-deviation error bars. Linear fits and confidence levels (CL) are shown for two-server attacks. For clarity one-server attack results are omitted. Baseline values represent the processing time in the absence of an attack.

CODEX by running additional CPU-intensive processes on servers under attack. The results are shown in Fig. 5. The horizontal axis shows the number of CPU-intensive background processes competing with the CODEX server process, and the vertical axis is the time required to complete a request. Processing times and variances are shown for varying attack strengths, with fits to the expected dependences.

As with the network attack, attacking a single server does not affect performance, since $2t+1$ servers remain unaffected. Fig. 5 shows that the time required to process a create or write request increases linearly with the additional load on the processors. But the slopes here are shallower because the computations are not concentrated in the protocol steps requiring full-quorum participation.

- For the create operation, over 80% of the time is spent in partial signature generation, which is unaffected, and that suggests the slope of the graph should be under 20% of the baseline time.
- For the write operation roughly 60% partial-signature time translates to a slope of less than 40% of the baseline.

In both cases the slopes in Fig. 5 are actually somewhat smaller, in part because of other expensive operations unaffected by the attack and in part because of issues such as process scheduling which might tend to favor the server process over the background processes.

TABLE 4

AVERAGE ROUND-TRIP TIMES FOR ICMP ECHO PACKETS.

	INRIA	Academia Sinica	UT Sydney
Cornell	135.3 ± 45.2	203.1 ± 0.6	223.5 ± 0.6
INRIA		296.4 ± 18.1	323.1 ± 2.8
Academia Sinica			376.9 ± 0.9

All times are measured in milliseconds and are averaged over ten round trips.

5.2 *Internet Deployment*

The Internet deployment of our CODEX prototype used for servers the PlanetLab[15] Pentium 4 processors running Linux and located at:

- Cornell University, Ithaca NY (1795MHz)
- INRIA, Sophia-Antipolis France (2193MHz)
- Academia Sinica, Taipei Taiwan (2392MHz)
- University of Technology at Sydney, Sydney Australia (1795MHz)

The server at Cornell also was the delegate. Average round-trip times for ICMP echo packets at the beginning of the experiment are shown in Table 4. As in the local deployment, the client was executed on a separate host (also located at Cornell) and was not included in the measurements.

Mean execution times for CODEX create, write, and read operations are given in Table 5; the fractions of time spent performing various actions are shown in Table 6 (again relative

[15]http://www.planet-lab.org

TABLE 5

PERFORMANCE OF CODEX OVER THE INTERNET.

Operation	Total CPU Time (ms)	Non-Idle Time (ms)	Idle Time (ms)
create	324.3 ± 41.8	197.9 ± 16.4	133.0 ± 38.7
write	490.4 ± 53.3	277.3 ± 28.6	215.5 ± 54.6
read	1441 ± 117.5	1404 ± 113.8	43.3 ± 33.0

The statistics for each of the measurements come from 110 requests. All times are measured on the delegate.

TABLE 6

COSTS OF OPERATIONS FOR CODEX OVER THE INTERNET.

	create	write	read
TLS	0.007 ± 0.017	0.010 ± 0.019	0.004 ± 0.005
Message Signing	0.108 ± 0.022	0.145 ± 0.021	0.018 ± 0.008
Partial Signature	0.855 ± 0.040	0.606 ± 0.040	0.238 ± 0.033
Partial Decryption			0.059 ± 0.007
$\Pi(s,p)$ Verification		0.209 ± 0.026	0.039 ± 0.005
DLProof Generation			0.117 ± 0.009
DLProof Verification			0.515 ± 0.030
Other	0.031 ± 0.043	0.028 ± 0.031	0.008 ± 0.006

The values shown are fractions of the non-idle time spent on the operation. The statistics for each of the operations come from 110 requests. All times are measured on the delegate.

to non-idle time to reduce sensitivity to latencies and scheduling); and comparisons with predictions are given in Table 7. A processing time of 1.4s for a CODEX read suggests that current-generation commodity hardware could handle roughly 40 requests per minute.

The time required for CODEX to produce a response is longer for the Internet deployment than the for LAN deployment. Mostly, this is a result of slower connections between servers. But processing times also increased—despite having faster processors in the Internet deployment. We believe this increased processing time can be attributed to memory swapping with the Internet deployment. The entire CODEX executable resided in RAM with our LAN deployment, but an examination of the Cornell PlanetLab machine revealed only 50% of the CODEX executable to be resident, with the rest in swap space.

6. RELATED WORK

CODEX is structurally similar to e-Vault[19], which is a data storage and key management system but, in contrast to CODEX, is not intended to distribute secrets. Both e-Vault and CODEX employ blind decryption of public-key encrypted data, a transparent client interface, and secure data storage. Whereas CODEX holds one private key for the service as a split secret, e-Vault holds the private keys of all its clients as split secrets. By using a client's public/private key pair to protect its symmetric keys, e-Vault obviates the need for the proofs of plaintext knowledge that CODEX requires. But with many more private keys as split secrets, the cost of the periodic share-refresh is much higher for e-Vault than for CODEX. Also, e-Vault makes a (weak) synchrony assumption, so it exhibits a vulnerability to network-delay attacks.

Fraga and Powell[15] present what is perhaps the first intrusion-tolerant data storage service in the context of a file system. In their system, files are fragmented and stored on a set of archive servers. Access to files is governed by security servers, to which a client must authenticate. A separate symmetric key must be held by the client for each server. Unlike CODEX, this system employs concurrency control in order to make operations atomic, though atomicity need only be enforced on a per-file basis.

Other work closely related to CODEX concerns key distribution and key management systems, including multicast key distribution systems, fault-tolerant key distribution centers (KDCs), and key escrow systems.

Multicast Key Distribution. Large-scale key distribution has been studied extensively for encrypted broadcast and multicast applications (see [13] for a survey). The goal is to distribute secrets from a sender to authorized recipients. These schemes are predominantly coordinated by the sender (or a service acting as its proxy) and require the sender to know all of the authorized recipients.

The design of [11] shifts some of the coordination to servers acting as "subgroup managers" and employs capability certificates to authorize recipients so that the recipient set need not be known *a priori*. A dual encryption scheme isolates most recipients from changes necessitated by new clients joining or leaving the broadcast group. The sender still keeps track of what clients belong to the group, though this is only needed to prevent a client from joining multiple subgroups and forcing a greater number of key changes. This design has very good scaling properties; it is not particularly fault-tolerant, though. Specifically, the protocols are ill-equipped to handle any faults that are not fail-stop and there is no integrity guarantee for the keys distributed. These properties—scalability without strong delivery assurance—are sensible for some applications, such as pay-per-view television broadcasts or online multimedia streaming. CODEX, in contrast, provides stronger delivery assurance, albeit without as good scaling properties.

Fault-tolerant KDCs. Much work has been done in constructing fault-tolerant KDCs. These are primarily credentials-issuing services, creating fresh secrets (symmetric keys) rather than distributing existing secrets (as CODEX does).

Deswarte *et al.* [10] describe a system that performs both authentication and authorization. Both operations are performed by distributed services and result in a client receiving one or more session keys (possibly included in tickets). For authentication, the client must authenticate itself to each server, and in order to tolerate faulty servers the client must share a distinct secret with each server. This makes adding new clients and servers expensive, though the authors note that a public key cryptosystem could be used instead of shared

TABLE 7

RATIOS OF TIME SPENT IN VARIOUS CRYPTOGRAPHIC OPERATIONS RELATIVE TO MESSAGE SIGNING.

	Observed Ratio	Predicted Ratio
Partial Signature	8.5 ± 2.9	8
Partial Decryption	3.3 ± 1.0	4
$\Pi(s,p)$ Verification	2.3 ± 0.7	2.3
DLProof Generation	6.3 ± 1.8	8
DLProof Verification	27.2 ± 7.7	30

Observed quantities are averaged over the operations in which they are used.

secrets. Authorization uses the session key returned by the authentication phase, and a single key can be used for all authorization servers.

The Kuperee[22] authentication service combines the functionalities of a KDC and a certification authority; it comprises a single KDC and replicated ticket-granting servers, and thus, unlike CODEX, has a single point of failure. Because Kuperee uses public keys to identify clients, it obviates the usual KDC requirement that a secret be shared with each client. Kuperee still must maintain some information about clients, namely their public keys, but these need not be protected using proactive secret sharing.

More traditional KDCs are presented by Gong[20] and Naor *et al.*[29]. The service shares a secret with each authorized client in these. Both systems tolerate Byzantine faults; the system discussed in [20] assumes synchrony between replicas, hence it is vulnerable to certain denial of service attacks. The system in [29] involves no synchrony assumption (like CODEX), but (unlike CODEX) requires secure links between clients and individual servers, which further increases the overhead of proactive recovery by requiring clients periodically to receive updated server public keys or new link-specific symmetric keys. When a client needs a new symmetric key, it participates in a threshold calculation of a pseudo-random function dependent on a KDC secret key. This direct client participation is more efficient than the transparent design of CODEX, and it might be appropriate in situations where operational efficiency is significantly more important than recovery efficiency. We could have designed CODEX along these lines, but it would have meant abandoning the transparent interface and incurring higher proactive recovery cost.

Key Escrow. Key escrow systems store client secrets (private/symmetric keys) for access by third parties (eg, law enforcement) with appropriate authority. In the context of CODEX, this is equivalent to requiring authorization policies for the read operation to recognize special credentials.

Chen *et al.*[5] describe a key escrow system that mediates Diffie-Hellman key agreement between clients, potentially in different domains, and in which a threshold number of servers must participate in order to recover the negotiated session key, which is split among the servers using secret sharing. This differs from a KDC in that the system merely acts as a proxy; it does not enforce any sort of authorization. The primary functionality of this system is computation, rather than storage or distribution.

The Ω key management service[33] also implements key escrow, though it manages private keys rather than symmetric keys. Ω also serves as a CA, a private key repository, and a decryption service. It is built using the Rampart[32] toolkit[16] and tolerates Byzantine failures by using state machine replication. Ω assumes a synchronous system, relying on timeouts to make progress, and requires secure links between clients and servers, again requiring that clients be kept abreast of new server public keys. Because escrowed keys are stored using secret sharing, proactive recovery requires that shares be regenerated for each escrowed key.

[16]A similar toolkit is discussed in Cachin[3].

7. Concluding Remarks

We expected that building CODEX would be a straightforward exercise in applying the architecture that we had developed for COCA. In some ways it was; in other ways it wasn't.

The idea of having a service key and implementing it as a shared secret that is proactively refreshed but never materialized at any server worked well in CODEX, just as it did with COCA. The idea of carefully choosing a service interface so that impossible problems (e.g., agreement in an asynchronous system) need not be solved within the service also again worked well. Some services cannot be built while following these tenets; by constructing first COCA and now CODEX, we have contributed a bit to better understanding which services can.

A surprise in developing CODEX was that read and write invocations must include proofs of plaintext knowledge or else attackers can learn secrets CODEX is storing. The approach to attack-tolerance embodied in CODEX (and COCA) does not address vulnerabilities in the service operations, their interfaces, or their semantics. Such a separation is convenient, but attacks that abuse the service interface must ultimately be addressed too. Methods to identify service-interface vulnerabilities are badly needed.

Appendix A
CODEX Protocol Details

Descriptions of CODEX protocols for create, write, and read operations are given below. These protocols assume each server has a public/private key pair, where the public key is known to all servers; for a message m, we write $\sigma_i(m)$ to denote the signature server S_i computes using that private key. Each server also stores a share of the CODEX private key, with shares periodically refreshed using the APSS [41] proactive secret sharing protocol.

To simplify the exposition, the protocols below are formulated in terms of communications links implementing confidentiality, integrity, and mutual authentication of end hosts. This link semantics is achieved in CODEX with a shared session key established by TLS using the public/private key of each server.

A.1 Protocol Details: create

Client p sends invocation message

$$M_{C(N)}\colon \langle \textsf{create}, N, \mathcal{C}_C(p,N), \mathcal{P}_W(N), \mathcal{P}_R(N)\rangle_p$$

to $t+1$ delegates. Each delegate D upon receiving $M_{C(N)}$ from a client p proceeds as follows.

1) D determines the validity of $M_{C(N)}$ by checking that p is authorized to create this name and checking the signature on $M_{C(N)}$. If a *preliminary registration* exists for N at D, then D also checks that $M_{C(N)}$ is consistent with that registration.
2) a) If there is no registration for name N at D, then D stores $M_{C(N)}$, creating a preliminary registration for N. Whether or not a new preliminary registration for N was just created, D next forwards

$M_{C(N)}$ along with a nonce n to all $3t+1$ CODEX servers S_i

$$\forall i.D \rightarrow S_i : n, M_{C(N)}$$

using a repeated send primitive and awaiting acknowledgments from $2t+1$ servers.

   b) If there is a verified registration for name N at D matching $M_{C(N)}$ then D sends cached corresponding confirmation message $\hat{M}_{C(N)}$ to client p and terminates the protocol.

   c) Otherwise, D terminates the protocol.

3) Upon receipt of a message "$n, M_{C(N)}$" from D, a server S_i determines validity of the message by using the validity tests of step (1).

   a) If $M_{C(N)}$ is valid then S_i stores $M_{C(N)}$, creating a preliminary registration for name N and replies

$$S_i \rightarrow D : n, \mathsf{ACCEPT}, \sigma_i(M_{C(N)})$$

   b) If there is a verified registration for name N at S_i matching $M_{C(N)}$ then S_i sends cached corresponding confirmation message $\hat{M}_{C(N)}$ to delegate D.

$$S_i \rightarrow D : n, \hat{M}_{C(N)}$$

D, upon receipt and determining that this message is valid, stores the verified registration and replies to client p:

$$D \rightarrow p : \hat{M}_{C(N)}$$

   c) If $M_{C(N)}$ is not determined to be valid then S_i rejects the request:

$$S_i \rightarrow D : n, \mathsf{REJECT}$$

If D receives REJECT messages from $t+1$ servers then obtaining a quorum in step 2a will not be possible, so D terminates the protocol.

4) An ACCEPT message received by D is determined to be valid if third field $\sigma_i(M_{C(N)})$ checks. Each valid ACCEPT message received by D is added to an evidence set $\mathcal{E}_D$. Once $2t+1$ pieces of such evidence have been collected, no other registration for name N can be accepted, so D creates a *verified registration* for N and composes confirmation message $\hat{M}_{C(N)}$ by invoking a threshold signature protocol with all servers. Let $\hat{M}^{-}_{C(N)}$ be confirmation message $\hat{M}_{C(N)}$ without the CODEX signature.

   a) $\forall i.D \rightarrow S_i : \mathcal{E}_D, \hat{M}^{-}_{C(N)}$

   b) D awaits partial signatures from $t+1$ servers and uses those partial signatures to construct $\hat{M}_{C(N)}$. Confirmation message $\hat{M}_{C(N)}$ is cached at D.

   c) $D \rightarrow p : \hat{M}_{C(N)}$

A.2 Protocol Details: *write*

Client p sends invocation message

$$M_{W(N)} : \langle \mathsf{write}, N, \mathcal{C}_W(p,N), E(s), \hat{M}_{C(N)}, \Pi(s,p)\rangle_p$$

to $t+1$ delegates. Each delegate D upon receiving $M_{W(N)}$ from a client p proceeds as follows.

1) D determines the validity of $M_{W(N)}$ by checking that p is authorized to write this name, checking the signature on $M_{W(N)}$, checking the validity of create confirmation $\hat{M}_{C(N)}$, checking the validity of knowledge of plaintext proof $\Pi(s,p)$, and checking that $val(N)$ at D either already equals $E(s)$ or is uninitialized.

2) If $M_{W(N)}$ is valid then D locally binds $E(s)$ to name N and then forwards $M_{W(N)}$ along with a nonce n to all $3t+1$ CODEX servers S_i

$$\forall i.D \rightarrow S_i : n, M_{W(N)}$$

using a repeated send primitive and awaiting acknowledgments from $2t+1$ servers.

3) Upon receipt of a message "$n, M_{W(N)}$" from D, a server S_i applies the validity checks of step (1).

   a) If $M_{W(N)}$ is found to be valid then S_i locally binds value $E(s)$ to name N and replies

$$S_i \rightarrow D : n, \mathsf{ACCEPT}, \sigma_i(M_{W(N)})$$

   b) If $M_{C(N)}$ is not found to be valid then then S_i rejects the request:

$$S_i \rightarrow D : n, \mathsf{REJECT}$$

If D receives REJECT messages from $t+1$ servers then it terminates the protocol.

4) An ACCEPT message received by D is deemed valid if third field $\sigma_i(M_{W(N)})$ checks. Each valid ACCEPT message received by D is added to an evidence set $\mathcal{E}_D$. When $2t+1$ pieces of such evidence have been collected, D sends to all servers a message indicating the value to which N should become bound. In what follows, n' is a fresh nonce, $\hat{M}^{-}_{W(N)}$ is confirmation message $\hat{M}_{W(N)}$ but without the CODEX signature, and the send is repeated until $2t+1$ responses are received.

$$\forall i.D \rightarrow S_i : n', \mathsf{VERIFY}, M_{W(N)}, \hat{M}^{-}_{W(N)}, \mathcal{E}_D$$

5) A VERIFY message received by a server S_i is deemed valid if accompanying evidence set $\mathcal{E}_D$ contains $2t+1$ valid ACCEPT messages.

   a) Upon receipt of a valid VERIFY message, server S_i binds $E(s)$ from $M_{W(N)}$ to name N and replies to D:

$$S_i \rightarrow D : n', \mathsf{VERIFIED}, \sigma_i(\hat{M}^{-}_{W(N)})$$

   b) If the VERIFY message is not valid, then server S_i rejects the request:

$$S_i \rightarrow D : n, \mathsf{REJECT}$$

If D receives REJECT messages from $t+1$ servers then it terminates the protocol.

6) Each VERIFIED message received by D is added to evidence set $\mathcal{E}'_D$. Once $\mathcal{E}'_D$ contains $2t+1$ pieces of evidence, D invokes a threshold signature protocol with all servers:

   a) $\forall i. D \to S_i : \mathcal{E}'_D, \hat{M}^-_{W(N)}$
   b) D awaits partial signatures from $t+1$ servers and uses those partial signatures to construct $\hat{M}_{W(N)}$.
   c) $D \to p : \hat{M}_{W(N)}$

A.3 Protocol Details: read

Client p selects a random secret blinding factor b_p, encrypts it using the CODEX public key, and sends invocation message

$$M_{R(N)}: \langle \mathsf{read}, N, \mathcal{C}_R(p,N), E(b_p), \Pi(b_p,p)\rangle_p$$

to $t+1$ delegates. Each delegate D upon receiving $M_{R(N)}$ from a client p proceeds as follows.

1) D determines the validity of $M_{R(N)}$ by checking the signature on $M_{R(N)}$ and checking validity of knowledge of plaintext proof $\Pi(b_p, p)$. If D knows[17] $\mathcal{P}_R(N)$ then it also checks that p is authorized to read this name.
   a) If $M_{R(N)}$ is valid then D forwards $M_{R(N)}$ along with a nonce n to all $3t+1$ CODEX servers S_i

   $$\forall i. D \to S_i : n, M_{R(N)}$$

   using a repeated send primitive and awaiting $t+1$ REJECT messages or $t+1$ ACCEPT messages with partial decryptions from the same sharing.
   b) If $M_{R(N)}$ is not valid then D terminates the protocol.
2) Upon receipt of a message "$n, M_{R(N)}$" from D, a server S_i applies the validity checks of step (1) and checks whether some value $val(N)$ is locally bound to name N. If no value is locally bound to N, then D ignores the request. Otherwise:
   a) If the validity checks are passed then S_i computes blinded ciphertext $c_i = E(val(N) \times b_p)$, its partial decryption $D_i(c_i)$ according to the share stored by S_i of the CODEX private key, and proof DL_i of the validity of $D_i(c_i)$. This information is sent to D:

   $$S_i \to D : n, \mathsf{ACCEPT}, \sigma_i(M_{R(N)}), c_i, D_i(c_i), DL_i$$

   b) If the validity checks did not pass then S_i rejects the request:

   $$S_i \to D : n, \mathsf{REJECT}$$

   If D receives REJECT messages from $t+1$ servers then it terminates the protocol.
3) An ACCEPT message received by D is deemed valid if $\sigma_i(M_{R(N)})$ checks and DL_i is valid. Each valid ACCEPT message received by D is added to an evidence set $\mathcal{E}_D$. When $t+1$ pieces of such evidence have been collected, D invokes a threshold signature protocol with all servers to sign $\hat{M}^-_{R(N)}$ creating confirmation message $\hat{M}_{R(N)}$:
   a) $\forall i. D \to S_i : \mathcal{E}'_D, \hat{M}^-_{R(N)}$
   b) D awaits partial signatures from $t+1$ servers and uses those partial signatures to construct $\hat{M}_{R(N)}$.
   c) $D \to p : \hat{M}_{R(N)}$

[17] D might not have participated in the create operation (either directly or by receiving a valid write request), so D might not know $\mathcal{P}_R(N)$.

Appendix B
Non-malleable Proof $\Pi(m, C)$

For ElGamal [12], an encryption of m can be written $(g^r \mod P, m \times y^r \mod P)$, where (P, g, y) is the public key and r is a random exponent chosen by the encryptor. Knowledge of r allows us to compute m (the converse is not true), so for a non-malleable proof $\Pi(m, C)$ that principal C knows m given $E(m)$, it suffices to prove that C knows r.

We base the construction on a result by Schnorr and Jakobsson[37] in which they present a non-malleable form of ElGamal encryption using a Schnorr signature[36] to make modification of the ciphertext detectable. Constructing this signature requires knowledge of r, and we exploit this to construct $\Pi(m, C)$ by including C's identity in the signature. The signature is non-malleable, so no other identity can be substituted for that of C.

All operations are modulo a large prime P of the form $P = 2Q+1$, where Q is another large prime. The construction in [37] is:

1) Select r, s uniformly at random in $\mathbb{Z}_Q$.
2) Compute $c = H(g^s, g^r, m \times y^r)$, where H is a secure hash function, such as SHA-1.
3) Compute $z = s + cr$ (over $\mathbb{Z}$, not $\mathbb{Z}_P$).
4) Output the signed encryption: $E(m, r, s) = (g^r, m \times y^r, c, z) \triangleq (a, b, c, z)$

A signed ciphertext (a, b, c, z) is valid if $H(g^z a^{-c}, a, b) = c$ holds. Schnorr and Jakobsson prove that this construction is secure against adaptive chosen ciphertext attacks.

Our contribution is noticing that step 2 can be changed to include C's identity (such as a certificate or credentials), which we will also denote C. Step 2 then becomes

2′) Compute $c = H(g^s, g^r, m \times y^r, C)$.

This new construction is a valid signature if $H(g^z a^{-c}, a, b, C) = c$ holds. Thus in addition to non-malleability we have a binding of the ciphertext to a particular identity. The security of this scheme is the same as in [37].

Acknowledgments

Lidong Zhou's source code for COCA provided very useful examples of interfacing with the OpenSSL package. We also thank Lidong for useful discussions on the design of CODEX.

Thanks to Jonathan Katz for bringing the RSA proof of plaintext knowledge to our attention. Mark Linderman of AFRL/Rome pointed out some of the security problems associated with publish/subscribe infrastructures and was a valuable sounding board as the CODEX design evolved.

Finally, we are grateful to editor Ravi Iyer and three reviewers who not only provided detailed and helpful comments on the initial version of this paper but did so on an incredibly tight schedule.

References

[1] G. R. Blakley. Safeguarding cryptographic keys. In R. E. Merwin, J. T. Zanca, and M. Smith, editors, *Proceedings of the 1979 National Computer Conference*, volume 48 of *AFIPS Conference Proceedings*, pages 313–317, New York, NY USA, September 1979. AFIPS Press.

[2] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[3] C. Cachin and J. A. Poritz. Secure intrustion-tolerant replication on the internet. In *Dependable Systems and Networks, 2002, Proceedings*, pages 167–176. IEEE, June 23–26 2002.

[4] 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, Proceedings*, pages 199–203, New York, 1983. Plenum Press.

[5] L. Chen, D. Gollmann, and C. J. Mitchell. Key escrow in mutually mistrusting domains. In T. M. A. Lomas, editor, *Security Protocols, International Workshop, Cambridge, United Kingdom, April 10-12, 1996, Proceedings*, volume 1189 of *Lecture Notes in Computer Science*, pages 139–153, Cambridge, UK, 1997. Springer.

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

[7] 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, Proceedings*, volume 1396 of *Lecture Notes in Computer Science*, pages 158–173, Berlin, Germany, February 1998. Springer-Verlag.

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

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

[10] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion tolerance in distributed computing systems. In *Symposium on Research in Security and Privacy, 1991, Proceedings*, pages 110–121. IEEE Computer Society, May 20–22 1991.

[11] L. Dondeti, S. Mukherjee, and A. Samal. Scalable secure one-to-many group communication using dual encryption. *Computer Communications*, 23(17):1681–1701, November 2000.

[12] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[13] A. Eskicioglu. Multimedia security in group communications: Recent progress in wired and wireless networks. In *Proceedings of the IASTED International Conference on Communications and Computer Networks*, pages 125–133, Cambridge, MA, November 4–6 2002. Acta Press.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[15] J. Fraga and D. Powell. A fault and intrusion-tolerant file system. In J. Grimson and H.-J. Kugler, editors, *Proceedings of the Third IFIP International Conference on Computer Security (IFIP/Sec'85)*, pages 203–218, Dublin, Ireland, August 12–15 1985. Elsevier Science Publishers B.V.

[16] 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, FL USA, October 20–22 1997. IEEE.

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

[18] 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, Proceedings*, volume 1560 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, Germany, 1998. Springer-Verlag.

[19] 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.

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

[21] L. C. Guillou and J.-J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both trasmission and memory. In C. G. Günther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 123–128, Davos, Switzerland, May 25–27 1988. Springer.

[22] T. Hardjono and J. Seberry. Replicating the Kuperee authentication server for increased security and reliability. In J. Pieprzyk and J. Seberry, editors, *Information Security and Privacy, First Australasian Conference, ACISP'96, Proceedings*, volume 1172 of *Lecture Notes in Computer Science*, pages 14–26, Wollongong, NSW, Australia, June 24–26 1996. Springer.

[23] R. Hayton, J. Bacon, and K. Moody. Access control in an open distributed environment. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA USA, May 1998. IEEE Computer Society Press.

[24] 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, Zürich, Switzerland, April 1–4 1997. ACM SIGSAC, ACM Press.

[25] 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, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 457–469, Berlin, Germany, 1995. Springer-Verlag.

[26] M. Jakobsson. On quorum controlled asymmetric proxy re-encryption. In H. Imai and Y. Zheng, editors, *Public Key Cryptography, Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography (PKC'99)*, volume 1560 of *Lecture Notes in Computer Science*, pages 112–121, Berlin, Germany, 1999. Springer-Verlag.

[27] 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.

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

[29] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *Advances in Cryptology—Eurocrypt'99, International Conference on the Theory and Application of Cryptographic Techniques, Proceedings*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346, Berlin, Germany, 1999. Springer-Verlag.

[30] K. Ohta and T. Okamoto. A modification of the Fiat-Shamir scheme. In S. Goldwasser, editor, *Advances in Cryptology—CRYPTO '88, Proceedings*, volume 403 of *Lecture Notes in Computer Science*, pages 232–243, Santa Barbara, California, USA, August 21–25 1990. Springer-Verlag.

[31] 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, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 89–104, Berlin, Germany, 1998. Springer-Verlag.

[32] 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, Selected Papers*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110, Berlin, Germany, 1995. Springer-Verlag.

[33] 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.

[34] R. Rivest, A. Shamir, and L. Adleman. A method of obtaining digital signature and public key systems. *Communications of the ACM*, 21:120–126, 1978.

[35] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[36] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 4(3):161–174, 1991.

[37] C.-P. Schnorr and M. Jakobsson. Security of signed ElGamal encryption. In T. Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 73–89, Berlin, Germany, 2000. Springer-Verlag.

[38] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[39] L. Zhou, M. A. Marsh, F. B. Schneider, and A. Redz. Distributed blinding for ElGamal re-encryption. Technical Report TR 2004-1920, Cornell University, Ithaca, New York, January 2004.

[40] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368, 2002.

[41] L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, Ithaca, New York, October 2002.