Implementing Trustworthy Services using Replicated State Machines

Fred B. Schneider[1]

Department of Computer Science
Cornell University
Ithaca, New York 14853

Lidong Zhou

Microsoft Research Silicon Valley
1065 La Avenida
Mountain View, California 94043

February 12, 2005

[1]Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-03-1-0156, 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 Grants 9703470 and 0430161, and grants from Intel Corporation and Microsoft 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

A thread of research has emerged to investigate the interactions of replication with threshold cryptography for use in environments satisfying weak assumptions. The result is a new paradigm known as distributed trust, and this article attempts to survey that landscape.

1 Introduction

The use of replicated state machines for implementing a Byzantine fault-tolerant service [19, 29] is well known:

1. Start with a server, structured as a deterministic state machine, that reads and processes client submitted *requests*. Requests are the sole means to change the server's state and/or cause the server to produce an output.

2. Run replicas of that server on distinct hosts. These hosts communicate through narrow-bandwidth channels and thus form a distributed system.

3. Employ a replica-coordination protocol to ensure that all non-faulty server replicas process identical sequences of requests.

Correct server replicas will all produce identical outputs for each given client request. Moreover, the majority of the outputs produced for each request will come from correct replicas provided (i) at most t server replicas are faulty, and (ii) the service comprises at least $2t + 1$ server replicas. So we succeed in implementing availability and integrity for a service that tolerates at most t faulty replicas by defining an output of the service to be any response produced by a majority of the server replicas.

Implicit in the approach are two assumptions. The first is to assume that a replica-coordination protocol exists. The second is to assume that the individual state machine replicas are independent if executed on separate hosts in a distributed system. That is, the probability pr_m of m replicas exhibiting Byzantine behavior is approximately $(pr_1)^m$ where pr_1 is the probability of a single replica exhibiting Byzantine behavior.

A *trustworthy* service must tolerate attacks as well as failures. Availability, integrity, and confidentiality are typically of concern. The approach outlined above is thus seriously deficient:

- Confidentiality is not just ignored, but n-fold replication actually increases the number of sites that must resist attack because they store (copies of) confidential information. Even services that do not operate on confidential data *per se* are likely to store cryptographic keys (so responses can be authenticated). Since these keys must be kept secret, support for confidentiality is needed even for implementing integrity.

- Any vulnerability in one replica is likely present in all, enabling attacks that succeed at one replica to succeed at all replicas. The independence assumption, manifestly plausible for hardware failures and many kinds of software failures (i.e., Heisenbugs), is thus unlikely to be satisfied once vulnerabilities and attacks are taken into account. So the probability that more than t servers are compromised is now approximately pr_1 rather than $(pr_1)^m$, and trustworthiness of the service is not improved by the replication.

- Replica coordination protocols are typically designed assuming the *synchronous* model of distributed computation. This is problematic because denial of service attacks can invalidate such timing assumptions. Once an assumption on which the system depends has been invalidated, correct system operation is no longer guaranteed.

A few of these deficiencies can be remedied by using cryptography or by using algorithms for different kinds of coordination; other of the deficiencies are driving current research. The goal of this article is to provide a principled account of that landscape, not dwelling on individual features but instead making clear how each contributes to implementing trustworthy services with replicated state machines. Each of the landscape's individual features is well understood in one or another research community, and some of the connections are too. But what is involved in putting them together is not widely documented nor broadly understood. Space limitations, however, allow only a superficial survey of the related literature, so view this paper as a starting point and consult the articles we do cite (and their reference lists) for more in-depth study.

Finally, it is worth emphasizing that the replication-based approaches discussed in this paper only address how to implement a more-trustworthy version of some service whose semantics are defined by a single state machine. We thus do not address vulnerabilities intrinsic in what that single state machine does. To solve the real trustworthiness problem requires determining that a state machine's semantics cannot be abused, which, unfortunately, is today an open research problem.

2 Compromise and Proactive Recovery

Two general types of components are involved in building trustworthy services: processors and channels. Processors serve as hosts; channels enable hosts to communicate.

A *correct* component only exhibits intended behavior; a *compromised* component can exhibit other behaviors. Component compromise is caused by failures and/or attacks. We make no assumption about the behavior of compromised components (the so-called Byzantine failure model), but we do conservatively assume that a component C compromised by a successful attack is then controlled by the adversary, with any secrets stored by C becoming then known to the adversary.

Secrets the adversary learns by compromising one component might facilitate the subsequent compromise of other components. For example, a correct channel protects the confidentiality, integrity, and authenticity of messages it carries. This channel functionality is typically implemented cryptographically, with keys stored at those hosts serving as the channel's endpoints. An attack that compromises a host thus yields secrets that then allow the adversary to compromise all channels attached to the host.

Because channel compromise is caused by host compromise, trustworthiness for a service is often specified solely in terms of which or how many host compromises can be tolerated; the possibility of channel compromise distinct from host compromise is ignored. This simplification, adopted in this paper too, is most defensible when the network topology provides several physically independent paths to each host, because then the channel connecting a host is unlikely to fail independent of that host.

Proactive Recovery

The system builder has little control over how and when a component transitions from being correct to being compromised. A *recovery protocol* provides the means to reverse such transitions. For a faulty component, the recovery protocol might involve replacing or repairing hardware. For a component that has been attacked, the recovery protocol must:

- evict the adversary, perhaps by restoring code from clean media (ideally with the recently exploited vulnerabilities patched),
- reconstitute state, perhaps from other servers, and
- replace any secret keys the adversary might have learned.

The reason to execute a recovery protocol after detecting a failure or attack is obvious. Less obvious are benefits that accrue from executing a recovery protocol periodically, even though no compromise has been detected [15]. To wit, such *proactive recovery* defends against undetected attacks and failures by transforming a service that tolerates t compromised hosts over its entire lifetime into a system that tolerates up to t compromised hosts during each *window of vulnerability* delimited by successive executions of the recovery protocol. The adversary that cannot compromise $t+1$ hosts within a window of vulnerability is foiled and forced to begin anew on a system with all defenses restored to full strength.

Denial-of-service attacks slow execution, thereby lengthening the window of vulnerability and increasing the interval available to perpetrate an attack. Whether such a lengthened window of vulnerability is significant will depend on whether it affords the adversary an opportunity to compromise more than t servers during the window. But whatever the adversary, systems with proactive recovery can, in principle, be more resilient than those without it, simply because proactive recovery (if implemented correctly) affords an opportunity for servers to recover from past compromises—including some compromises that have not been detected.

3 Service Key Refresh and Scalability

With the state machine approach, a client, after making a request, awaits responses from servers. When the compromise of up to t servers must be tolerated, the same response received from fewer than t servers cannot be considered correct. But if the response is received from $t + 1$ or more servers then that response was necessarily produced by a correct server. So sets of $t+1$ servers together *speak for* the service, and clients require some means to identify when equivalent responses have come from $t + 1$ distinct server replicas.

One way to ascertain the origin of responses from (correct) servers is to employ digital signatures. Each server's response is digitally signed using a private key known only to that server; the receiver validates the origin of a response by checking the signature using that server's public key. A server's private key thus speaks for that server. Less expensive schemes, involving message authentication codes (MAC) and shared secrets, have also been developed; such schemes contribute to the performance reported for toolkits (e.g., BFT) that have recently become available to system builders.

The use of secrets—be it private keys or shared secret keys—for authenticating server replicas to clients impacts the scalability of a service that employs proactive recovery. This is because new secrets must be selected at the start of each window of vulnerability, and clients must then be notified of the changes. If the number of clients is large then performing the notifications will be expensive, and the resulting service ceases to be scalable.

To build a service that is scalable, we seek a scheme whereby clients need not be informed of periodic changes to server keys. Since sets of $t+1$ or more servers speak for the service, a client could identify a correct response from the service if the service has some means to digitally sign responses exactly when a set of servers that speak for the service agree on that response:

TC1: Any set of $t+1$ or more server replicas can cooperate and digitally sign a message on behalf of the service.

TC2: No set of t or fewer server replicas can contrive to digitally sign a message on behalf of the service.

TC1 implies that information held by $t+1$ or more servers enables them to together construct a digital signature for a message (namely, for the service's response to a request), whereas TC2 implies that no coalition of t or fewer servers has enough information to construct such a digital signature. In effect, TC1 and TC2 characterize a new form of private key for digital signatures—a key that is associated with the service rather than with the individual servers. This private key speaks for the service but is never entirely materialized at individual servers comprising the service.

A private key satisfying TC1 and TC2 can be implemented using secret sharing [30, 2]. An $(n, t+1)$ *secret sharing* for a secret s is a set of n random *shares* such that (i) s can be recovered with knowledge of $t+1$ shares, and (ii) no information about s can be derived from t or fewer shares. Not only do protocols exist to construct $(n, t+1)$ secret sharings but *threshold digital signature* protocols [3, 10] exist that allow a digital signature to be constructed for a message from $t+1$ *partial signatures*, where each partial signature is computed using as inputs the message along with only a single share of the private key. Thus, TC1 and TC2 can be implemented by using $(n, t+1)$ secret sharing and dividing the service private key among the server replicas—one share per replica—and then having servers use threshold digital signatures to collaborate in signing responses.

If the shares are fixed then, over time, an attacker might compromise $t+1$ servers, obtain $t+1$ shares, and thus be able to speak for the service, generating correctly signed bogus service responses. Such an attacker is known as a *mobile adversary* [25], since it attacks and controls one server for a limited time before moving to the next. The defense against mobile adversary attacks is, as part of proactive recovery, for servers periodically to (i) create a new and independent secret sharing for the service private key, and then (ii) delete the old shares, replacing them with the new shares. Because the new and old secret sharings are independent, the mobile adversary cannot combine new shares and old shares in order to obtain the service's signing key. And because old shares are deleted when replaced by new shares, a mobile adversary must compromise more than t servers within a single window of vulnerability in order to succeed.

Protocols to create new, independent sharings of a secret are called *proactive secret sharing* protocols and have been developed for the synchronous model [15] as well as for the *asynchronous* model, which makes no assumptions about process execution speeds and message delivery delays [4, 36]. Proactive secret sharing protocols are tricky to design because:

- The new sharing must be computed without ever materializing the shared secret at any server. (A server that materialized the shared secret, if compromised, could reveal the service's signing key to the adversary.)
- The protocol must work correctly in the presence of as many as t compromised servers, which might provide bogus shares to the protocol.

4 Server Key Refresh

Secure communication channels between servers are required for proactive secret sharing and for various other protocols that servers execute. Since keys used to implement a secure channel are stored by hosts at the endpoints of that channel, we conclude that, not withstanding the use of secret sharing and threshold cryptography for service private keys, there will be other cryptographic keys stored at servers. If these other keys can be compromised then they too must be refreshed during proactive recovery. Three classes of solutions for server key refresh have been proposed; they are the subject of this section.

4.1 Trusted Hardware

Although not in widespread use today, special-purpose cryptographic hardware that stores keys and performs cryptographic operations (e.g., encryption and decryption) does exist. This hardware is designed so that, if correctly installed, it will not divulge keys or other secret parameters, even if the software on the attached host has been compromised. When keys stored by a server cannot be revealed, there is no reason to refresh them. So, storing server keys in special-purpose cryptographic hardware attached to a server eliminates the need to refresh server keys as part of proactive recovery for as long as that hardware can be trusted.

However, use of special-purpose cryptographic hardware for all cryptographic operations does not prevent a compromised server from performing cryptographic operations for the adversary. The adversary might, for example, cause the server to generate signed or encrypted messages for later use in attacks. A defense here is to maintain an integer counter in stable memory (so the counter's value will persist across failures and restarts) that is part of the special-purpose cryptographic hardware. This counter is incremented every time a new window of vulnerability starts; and the current counter value is included in every message that is encrypted or signed using the tamper-proof hardware. A server can now ignore any message it receives that has a counter value too low for the current window of vulnerability.

The need for special-purpose hardware would seem to limit adoption of this approach. But recent announcements from industry groups like the Trusted Computing Group (see https://www.trustedcomputinggroup.org/home) and hardware manufacturers like IBM and Intel imply that standard PC computing systems soon will support reasonable approximations to this hardware functionality, at least for threats common on the Internet today.

4.2 Off-line Keys

In this approach to server key refresh, new keys are distributed using a separate secure communications channel that the adversary cannot compromise. This channel typically is implemented cryptographically by using secrets that are stored and used in an off-line stand-alone computer, thereby ensuring inaccessibility to a network-borne adversary. For example, an administrative public/private key pair could be associated with each server H. The administrative public key $\hat{K}_H$ is stored in ROM on all servers; the associated private key $\hat{k}_H$ is stored off-line and is known only to the administrator of H. Each new server private key k_A for a host A would be generated off-line. The corresponding public key K_A would then be distributed to all servers by including K_A in a certificate signed using the administrative private key $\hat{k}_A$ of server A.

4.3 Attack Awareness

Instead of relying on a full-fledged tamper-proof co-processor, a scheme suggested in Canetti and Herzberg [7] uses non-modifiable storage (e.g., ROM) to store a special service-wide public key, whose corresponding private key is shared among servers using an $(n, t + 1)$ secret sharing. To refresh its server key pair, a server H generates its new private/public key pair, signs the new public key using the old private key, and then requests that the service *endorse* the new public key. Such an endorsement is represented by a certificate that associates the new public key with server H and that is signed using the special service private key.

The service private key is refreshed periodically using proactive secret sharing, thereby guaranteeing that an attacker cannot learn the service private key provided the attacker cannot compromise more than t servers in a window of vulnerability. Therefore, an attacker cannot fabricate a valid endorsement because bogus certificates are detected by servers using the service public key stored in their ROM. A server becomes aware of an attack if it does not receive a valid certificate for its new public key within a reasonable amount of time or if it receives two conflicting requests signed by the same server's private key during the same window of vulnerability. In either case, actions should be initiated to re-introduce the server into the system and remove the possible imposter.

5 Processor Independence

The processor independence assumption is approximated to the extent that a single attack or host failure cannot cause multiple hosts to become compromised. Independence is reduced, for example, when

- hosts employ common software (and thus replicas have the same vulnerabilities),
- hosts are operated by the same organization (because a single maleficent operator could then access and compromise more than a singled host), or
- hosts rely on a common infrastructure, such as name servers or routers used to support communications, since the compromise of that infrastructure violates an assumption needed for the hosts to function.

One general way to characterize the trustworthiness of a service is by describing which sets of components could together be compromised without disrupting correct operation of the service. Each vulnerability V partitions server replicas into groups, where replicas in a given group share that vulnerability. For instance, there exist attacks that compromise server replicas running Linux but not those running Windows (and *vice versa*), which leads to a partitioning according to operating system; and the effects of a maleficent operator are likely localized to server replicas under that operator's control, which leads to a partitioning according to system operator.

Sets of a system's servers whose compromise must be tolerated can be specified using an *adversary structure* [16, 22]. This is a set $\mathcal{A} = \{S_1, \ldots, S_r\}$ whose elements are sets of system servers the adversary is assumed able to compromise during the same window of vulnerability. A trustworthy service is then expected to continue operating as long as the set of compromised servers is an element of $\mathcal{A}$. Thus, the adversary structure $\mathcal{A}$ for a system intended to tolerate attacks on the operating system would contain sets S_i whose elements are servers all running the same operating system.

When there are n server replicas and $\mathcal{A}$ contains all sets of servers of size at most t, the result is known as an (n, t) *threshold* adversary structure [30]. The basic state machine approach involves a threshold adversary structure, as does much of the discussion throughout this article. Threshold adversary structures correspond to systems in which server replicas are assumed to be independent and equally vulnerable. They are, at best, approximations of reality. The price of embracing such approximations is that single events might actually compromise all of the servers in some set that is not an element of the adversary structure—the service would then be compromised.

Protocols designed for threshold adversary structures frequently have straightforward generalizations to arbitrary adversary structures. What is less well understood is how to identify an appropriate adversary structure for a system, since doing so requires identifying the common vulnerabilities. Today's systems often employ commercial off-the-shelf (COTS) components, and therefore access to internal details is restricted. Yet those internal details are what is needed in identifying common vulnerabilities.

Independence by Avoiding Common Vulnerabilities

Eliminating software bugs eliminates vulnerabilities that would impinge on replica independence. The construction of bug-free software is quite difficult, however. So instead we turn to another means of increasing replica independence: diversity. In particular, the state machine approach does not require that server replicas be identical in either their design or their implementation—only that different replicas produce equivalent responses for each given request. Such diversity can be obtained in three ways:

- Develop multiple server implementations. This, unfortunately, can be expensive. The cost of all facets of system development are multiplied because each replica now has its own design, implementation, and testing costs. In addition, interoperation of diverse components is typically more difficult to orchestrate, not withstanding the adoption of standards. Finally, experiments have shown that distinct development groups working from a common specification will produce software having the same bugs [18].

- Employ pre-existing diverse components that have similar functionality and then write software wrappers so that all implement the same interface and the same state machine behavior [29, 28]. One difficulty here is procuring diverse components that do have the requisite similar functionality. Some operating systems have multiple, diverse implementations (e.g., BSD UNIX vs. Linux) but other operating systems do not; and application components we use in building a service are unlikely to have multiple diverse realizations. A second difficulty arises when components do not provide access to internal non-deterministic choices they make during execution (e.g., for creating a "handle" that will be returned to a client), since now writing the wrapper can be quite difficult [28]. And, finally, there still remains a chance that the diverse components will share vulnerabilities because they are written to the same specification (exhibiting a phenomenon like that reported in [18]) or because they are built using some of the same components or tools.

- Introduce diversity automatically during compilation, loading, or in the runtime environment [12, 34]. Code can typically be generated and storage allocated in any number of ways for a given high-level language program; making choices in producing different executables introduces a measure of diversity. Different executables for the same high-level language program are still implementations of the same algorithms, though, so executables obtained in this manner will continue to share any flaws in those algorithms.

6 Replica Coordination

In the state machine approach, not only must state machine replicas exhibit independence but all correct replicas must reach consensus about the contents and ordering of client requests. Therefore, the replica-coordination protocol must include some sort of *consensus protocol* [26] to ensure that

- all correct state machine replicas agree on each client's request, and

- if the client sends the same request R to all replicas then R is the consensus they reach for that request.

This specification involves both a safety property and a liveness property. The safety property prohibits different replicas from agreeing on different values or orderings for any given request; the liveness property stipulates that an agreement is always reached.

Consensus protocols exist only for systems satisfying certain assumptions [11]. In particular, deterministic consensus protocols do not exist for systems having unboundedly slow message delivery or process execution speeds—systems satisfying the asynchronous model. This limitation arises because, to reach consensus in such a system, participating state machine replicas must distinguish between (i) those replicas that have halted (due to failures) and thus should be ignored and (ii) those replicas that, though correct, are executing very slowly and thus cannot be ignored.

The impossibility of implementing a deterministic consensus protocol in the asynchronous model leaves three options.

Option I: Abandon Consensus. Instead of arranging that every state machine replica receive every request, we might instead employ servers that are not as tightly coordinated. One well known example is the use of a *quorum system* to implement a storage service from individual *storage servers*, each of which supports local read and write operations. And various robust storage systems [21, 23, 33] have been structured in this way, as have richer services such as the COCA [37] certification authority, which implements operations involve both reading and writing service state.

To constitute a quorum system, servers are associated with groups; each operation is executed on all servers in some group. Moreover, these groups are defined so that pairs of groups intersect in one or more servers—the effect of one operation can thus be seen by any subsequent operation. Various quorum schemes differ in the size of the intersection of two quorums. For example, if faulty processors simply halt then as many as t faulty processors can be tolerated by having $2t + 1$ processors in each group and $t + 1$ in the intersection. If faulty processors can exhibit arbitrary behavior then a *Byzantine quorum system* [22], involving larger groups and a larger intersection, is required.

A second example of abandoning consensus replication can be seen in the APSS asynchronous proactive secret sharing protocol[36]. Here, each participating server computes a new sharing of some secret, and a consensus protocol would seem the obvious way for all correct servers to agree on which new sharing to adopt. But instead, each server embraces all of the new sharings; a consensus protocol for the asynchronous model is not then needed. Clients of APSS refer to individual shares by using names that enable servers to know which sharing is involved. So here is a place where establishing consensus turns out to be unnecessary after the problem specification is changed slightly—APSS creates at most n new and independent sharings of a secret when started with n sharings, rather than creating a single new sharing from a single sharing.

Certain service specifications cannot be implemented without solving a consensus problem, so abandoning consensus is not always an option. But it is an option, albeit one that is (too) rarely considered.

Option II: Employ Randomization. The impossibility result of Fischer *et al.* [11] does not rule out protocols that use randomization, and practical randomized asynchronous Byzantine agreement protocol have been developed. A practical example is the consensus protocol of Cachin *et al.* [5], which builds on some new cryptographic primitives including a non-interactive threshold signature scheme and a threshold coin-tossing scheme; the protocol is part of the Sintra toolkit [6] developed at the IBM Zurich Research Center. Sintra supports a variety of broadcast primitives needed for coordination in replicated systems.

Option III: Sacrifice Liveness (Temporarily). A service cannot be very responsive when processes and message delivery have become glacially slow, so the liveness property of a consensus protocol might temporarily be relaxed in those circumstances. After all, there are no real-time guarantees in the asynchronous model anyway. The crux of this option, then, is to employ a consensus protocol (i) that satisfies its liveness property only while the system satisfies assumptions somewhat stronger than found in the asynchronous model but (ii) that always satisfies its safety property (so different state machine replicas still agree on requests they process). Lamport's Paxos protocol [20] is a well known example of trading liveness for operation under the weaker assumptions of the asynchronous model. Other examples include the protocol of Chockler, Malkhi and Reiter [9] and BFT [8].

7 Computing with Server Confidential Data

Some services involve data that must be kept confidential. Unlike secrets used in connection with cryptography (*viz.* keys), such server data cannot be changed periodically as part of proactive recovery, because values now have significance beyond just being secret and could be part of computations that support the services semantics.

Information stored unencrypted on a server becomes known to the adversary if that server is compromised. Thus, confidential service data must always be stored in some sort of encrypted form—either replicated or partitioned among the servers. Unfortunately, few algorithms have been found that perform interesting computations on encrypted data (although some limited search operations can now be supported [31]). Even temporarily decrypting the data on a server replica or storing it on a backup in unencrypted form risks disclosing secrets to the adversary.

One promising approach is to employ *secure multi-party computations* [14]. Much is known about what can and cannot be done as a secure multi-party computation; less is known about what can and cannot be done efficiently, and the prognosis is not good for efficiently supporting arbitrary computations (beyond cryptographic operations like decryption and signing).

It is not difficult to implement a service that simply stores confidential data for subsequent retrieval by clients. An obvious scheme has the client encrypt the confidential data and forward that encrypted data to a storage service for subsequent retrieval. Only the client and other principals with knowledge of the decryption key would then be able to make sense of the data they retrieve. Note, the service here has no means to control which principals are able to access unencrypted confidential data.

In cases where we desire the service—and not client that initially stores the confidential data—to implement access control, then simply having a client encrypt the confidential data no longer works. The key elements of the solution to this problem have already been described, though.

- The confidential data (or a secret key to encrypt the data) is encrypted using a service public key.

- The corresponding private key is shared among replicas using an $(n, t+1)$ secret sharing scheme and refreshed periodically using proactive secret sharing.

- A copy of the encrypted data is stored on every replica to preserve its integrity and availability in face of server compromises and failures.

Two schemes have been proposed for clients to retrieve the encrypted data.

Re-encryption. A re-encryption protocol produces a ciphertext encrypted under one key from a ciphertext encrypted under another but without the plaintext becoming available during intermediate steps. Such protocols exist for public key cryptosystems where the private key is shared among a set of servers [17]. To retrieve a piece of encrypted data, the service executes a re-encryption protocol on data encrypted under the service public key; data encrypted under the public key of an authorized client is the result.

Blinding. A client chooses a random blinding factor, encrypts it using the service public key, and sends that to the service. If that client is deemed by the service to be authorized for access then the service multiplies the encrypted data by this blinding factor and then employs threshold decryption to compute un-encrypted but blinded data, which is sent back to the client. The client, knowing the blinding factor, can then recover the data from that blinded data.

Blinding can be considered a special case of re-encryption, because blinding is essentially encryption with a one-time pad (the random blinding factor). Unlike the re-encryption scheme in [17], which demands no involvement of the client and produces a ciphertext for a different key in the same encryption scheme, our use of blinding requires client participation and yields a ciphertext under a different encryption scheme. So, re-encryption can be used directly for cases where a client itself is a distributed service with a service public key, while the blinding-based scheme cannot without further modification. In fact, a re-encryption scheme based on blinding appears in [35]. There, ciphertext encrypted under the service public key is transformed into ciphertext encrypted under the client public key (as with the re-encryption scheme in [17]), thereby allowing a flexible partition of work between client and service.

8 Status and Future Directions

Various systems have been built using the elements we have just outlined. These efforts are summarized in Figure 1 and Figure 2. There is clearly much to be

BFS [8]: An NFS file system implementation built using BFT. See Figure 2 for a description of the BFT toolkit.

COCA [37]: A trustworthy distributed certification authority. COCA avoids consensus protocols by using a Byzantine quorum systems. The system employs threshold cryptography to produce certificates signed by the service, using proactive recovery in conjunction with off-line administrator keys for maintaining authenticated communication links. COCA assumes the asynchronous model.

CODEX [24]: A robust and secure distribution system for confidential data. CODEX stores private keys using secret sharing with proactive refresh, uses threshold cryptography, and employs a distributed blinding protocol in order to send confidential information from the service to a client or to another distributed service. CODEX assumes the asynchronous model.

E-Vault [13]: A secure distributed storage system. E-vault employs threshold cryptography to maintain private keys, uses blinding for retrieving confidential data, and implements proactive secret sharing. E-vault assumes the synchronous system model.

Figure 1: Systems that Employ Elements of Distributed Trust.

learned about how to engineer systems based on these elements, and only a small part of the landscape has been explored.

The trustworthiness of a system is ultimately tied to a set of assumptions about the environment in which that system must function. Weaker assumptions should be preferred, since then there is less risk that they will be violated by natural events or attacks. But that renders irrelevant much prior work in fault-tolerance and distributed algorithms.

First, until recently, the synchronous model of computation has generally been assumed. But there are now good reason to investigate algorithms and system architectures for asynchronous models of computation: concern about denial-of-service attacks and interest in distributed computations that span wide-area networks. Second, most of the prior work on replication has ignored confidentiality. Yet confidentiality is not orthogonal to replication and poses a new set of challenges, so it cannot be ignored. Moreover, because confidentiality is not a property of an individual component's states or state transitions, usual approaches to specification and system refinement, which are concerned with what actions components perform, are not germane.

The system design approach outlined in this paper has been referred to as im-

BFT [8]: A toolkit for implementing replicated state machines in the asynchronous model. Services tolerate Byzantine failures and use a proactive recovery mechanism for periodically re-establishing secure links among replicas and restoring the code and the state of each replica. BFT employs consensus protocols and sacrifices liveness to circumvent the impossibility result for consensus in the asynchronous model. For proactive recovery, BFT assumes a secure cryptographic co-processor and a watchdog timer. BFT does not provide support for storing confidential information or for maintaining a service private key that is required for scalability.

ITTC (Intrusion Tolerance via Threshold Cryptography) [32]: A toolkit that includes a threshold RSA implementation with distributed key generation and share refreshing. Share refreshing is done when instructed by an administrator. No clear system model is provided, but the protocols seem to be suitable for use in the asynchronous model.

Phalanx [23]: Middleware for implementing scalable persistent survivable distributed object repositories. A Byzantine quorum system allows Byzantine failures to be tolerated, even in the asynchronous model. Ramdomized protocols are used to circumvent the impossibility result for consensus in the asynchronous model. Phalanx does not provide support for storing confidential information or for maintaining confidential service keys; it also does not implement proactive recovery.

Proactive security toolkit (IBM) [1]: A toolkit for maintaining proactively secure communication links, private keys, and data storage in synchronous systems. The design employs attack-awareness approach (with ROM) for refreshing the servers' public/private key pairs.

SINTRA (Secure INtrusion-Tolerant Replication Architecture) [6]: A toolkit that provides a set of group communication primitives for implementing a replicated state machine in the asynchronous model where servers can exhibit Byzantine failures. Randomized protocols are used to circumvent the impossibility result for consensus in the asynchronous model. SINTRA does not provide support for storing confidential information or for maintaining a service private key that is required for scalability, although the design of an asynchronous proactive secret sharing protocol is documented elsewhere.

Figure 2: Toolkits for Implementing Distributed Trust.

plementing *distributed trust* [27], because it allows a higher level of trust to be placed in an ensemble than could be placed in a component. There is no magic here. Distributed trust requires that component compromise be independent. To date, only a few sources of diversity have been investigated and only a subset of those have enjoyed practical deployment. Real diversity is messy and often brought about by random and unpredictable natural processes, in contrast to how most computations are envisaged (as a preconceived sequence of state transitions). Think about how epidemics spread (from random, hence diverse, contacts between individuals) to wipe out a population (a form of "reliable broadcast"); think about how individuality permits a species to survive or how diverse collections of species allow an ecosystem to last.

Finally, if cryptographic building blocks, like secret sharing and threshold cryptography, seem a bit arcane today, it is perhaps worth recalling that twenty years ago, research in consensus protocols was considered a niche concern that most systems builders ignored as impractical. Today, systems designers understand and regularly use such protocols in order to implement systems that can tolerate various kinds of failures even though hardware is more reliable than ever. The promising technologies for trustworthiness, such as secret sharing and threshold cryptography, are today also seen by many as a niche concern. This cannot persist for long, given our growing dependence on networked computers which, unfortunately, makes us hostage not only to failures but also to attacks.

Acknowledgments

Helpful comments on earlier drafts of this paper were provided by Martin Abadi, Úlfar Erlingsson, Andrew Myers, Mike Schroeder, Gun Sirer, and Ted Wobber. We are also very grateful to three anonymous reviewers for insightful suggestions and pointers to literature we overlooked, all of which helped us to clarify the exposition.

Discussions with Robbert van Renesse were instrumental in developing our first prototype systems that embodied these ideas; Mike Marsh worked on a second system, leading to our work on distributed blinding.

References

[1] 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, Kent Ridge Digital Labs, Singapore, November 1999. ACM SIGSAC, ACM.

[2] 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.

[3] C. Boyd. Digital multisignatures. In H. Baker and F. Piper, editors, *Cryptography and Coding*, pages 241–246. Clarendon Press, 1989.

[4] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97. ACM, ACM Press, November 2002.

[5] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 123–132. ACM, July 2000.

[6] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, pages 167–176. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society, June 2002.

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

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[9] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the International Conference on Distributed Systems*, pages 11–20, Pheonix, Arizona, USA, 2001. IEEE Computer Society.

[10] 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.

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

[12] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, Cape Cod, MA, May 1997. Computer Society Press.

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

[14] 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.

[15] 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.

[16] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[17] 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.

[18] J. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE–12(1):96–109, January 1986.

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

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[21] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, Canada, August 1986. ACM Press.

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

[23] 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.

[24] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January-March 2003.

[25] 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.

[26] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

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

[28] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, Banff, Canada, October 2001. ACM.

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

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

[31] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium Security and Privacy*, pages 44–55, Oakland, CA USA, May 2000. IEEE Computer Society Press.

[32] 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. The USENIX Association.

[33] 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.

[34] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207 (CRHC-03-03), Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, May 2003.

[35] L. Zhou, M. A. Marsh, F. B. Schneider, and A. Redz. Distributed blinding for ElGamal re-encryption. In *Proceedings of the 25th International Conference on Distributed Computing Systems*, Columbus, Ohio, USA, June 2005. IEEE Computer Society. To appear.

[36] L. Zhou, F. B. Schneider, and R. van Renesse. APSS: Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, October 2002. Submitted for publication.

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