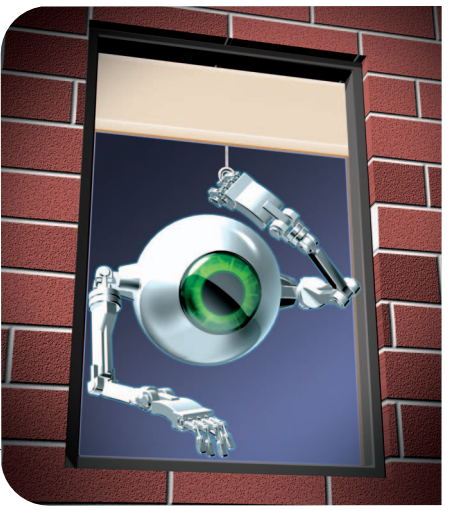
Implementing Trustworthy Services Using Replicated State Machines

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



Fred B. Schneider
Cornell University

Lidong Zhou
Microsoft Research Silicon Valley

"Divide and conquer" can be a powerful tool for disentangling complexity when designing a computing system. However, some aspects of system design are inseparable. Treating these as though they were independent leads to one interfering with the other, and "divide and be conquered" perhaps better characterizes the consequences. For some years, we have been investigating how to construct systems that continue functioning despite component failures and attacks. A question we have pondered is to what extent does divide and conquer apply? Somewhat less than you might hope is, unfortunately, the answer.

One could argue that attacks can be seen as just another cause for component failure. The *Byzantine fault model* asserts that a faulty component can exhibit arbitrarily malicious (so-called "Byzantine") behavior; a system that tolerates Byzantine faults should then be able to handle anything. Moreover, because any component can be viewed abstractly in terms of its state and a set of possible next-state transitions—in short, a *state machine*—fault-tolerant services can be built by assembling enough state-machine copies so that outputs from the ones exhibiting Byzantine behavior are outvoted by the correctly functioning ones. The fault-tolerance of the ensemble thus exceeds the fault-tolerance of any individual state machine, and a *distributed fault-tolerance* is the result.

A closer look at such *replicated state machines*, however, reveals problems when attacks are possible. Specific difficulties with the approach and how we can overcome these are described later in this article, but the overall vision remains compelling: place more trust in an ensemble than in any of its individual components. In analogy with distributed fault-tolerance, then, we are seeking ways to implement *distributed trust*.

The state-machine approach

The details for using replicated state machines and implementing a Byzantine fault-tolerant service[1,2] are well known:

1. Start with a server, structured as a deterministic state machine, that reads and processes clients' *requests*, which are the sole means to change the server's state or cause it to produce an output.
2. Run replicas of that server on distinct hosts. These hosts communicate through narrow-bandwidth channels, thus forming a distributed system.
3. Employ a replica-coordination protocol to ensure that all non-faulty server replicas process identical sequences of requests.

Correctly operating server replicas will produce identical outputs for each given client request. Moreover, the majority of the outputs produced for each request will come from correct replicas provided that at most t server replicas are faulty and that 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 the service's output to be any response produced by a majority of the server replicas.

Implicit in the approach are two assumptions. First, we assume that a replica-coordination protocol exists.

Second, we assume *processor independence*—that the individual state-machine replicas do not influence each other 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 earlier is thus seriously deficient because

- 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). Because 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. The independence assumption, manifestly plausible for hardware failures and many kinds of software failures (such as 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 $(Pr_1)^m$; therefore, replication does not improve the service's trustworthiness.
- Replica-coordination protocols are typically designed assuming the *synchronous* model of distributed computation. This is problematic because denial-of-service (DoS) attacks can invalidate such timing assumptions. Once an attacker has invalidated an assumption on which the system depends, correct system operation is no longer guaranteed.

Using cryptography or algorithms for coordination can remedy a few of these deficiencies; other deficiencies drive current research. This article's goal is to provide a principled account of that landscape: instead of dwelling on individual features, we show 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 as well, 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 article as a starting point and consult the articles we cite (and their reference lists) for a more in-depth study.

Finally, it's worth emphasizing that the replication-based approaches we discuss 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 entire trustworthiness problem requires determining that a state machine's semantics cannot be abused; unfortunately, this is still an open research problem.

Compromise and proactive recovery

Two general 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 behavior. Component compromise is caused by failures or attacks. We make no assumption about the behavior of compromised components, but we do conservatively assume that a component C compromised by a successful attack is then controlled by the adversary, with any secrets C stores then becoming known to the adversary.

Secrets the adversary learns by compromising one component might subsequently lead to the 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, a service's trustworthiness is often specified solely in terms of which or how many host compromises it can tolerate; the possibility of channel compromise distinct from host compromise is ignored in such a specification. This simplification—also adopted in this article—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.

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 that a component should execute a re-

covery protocol after detecting a failure or attack is obvious. Less obvious are benefits that accrue from a component executing a recovery protocol periodically, even though no compromise has been detected.[3] To wit, such *proactive recovery* defends against undetected attacks and failures by transforming a service that tolerates t compromised hosts over its 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.

The adversary that cannot compromise $t + 1$ hosts within a window of vulnerability is foiled and forced to begin anew.

DoS 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 the adversary can 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 haven't been detected.

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 a response's origin 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 (for example, BFT mentioned in Table 2) that have recently become available to system builders.

Service private keys

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 servers must select new secrets 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 scalable service, we seek a scheme whereby clients don't need to be informed of periodic changes to server keys. Because 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 a way to digitally sign responses if and only if 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 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.[4,5] An $(n, t + 1)$ secret sharing for a secret s is a set of n random *shares* such that: s can be recovered with knowledge of $t + 1$ shares, and 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[6,7] exist that allow construction of a digital signature for a message from $t + 1$ *partial signatures*, where each partial signature is computed using as inputs the message along with a single share of the private key. Thus, a system can implement TC1 and TC2 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 yet bogus service responses. Such an attacker is known as a *mobile adversary*,[8] because 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 create a new and independent secret sharing for the service's private key, and then delete the old shares, replacing them with new ones. Because the new and old secret sharings are independent, the mobile adversary can't combine new and old shares 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 to succeed.

Proactive secret sharing

Protocols to create new, independent sharings of a secret are called *proactive secret sharing* protocols and have been developed for the synchronous model[3] as well as for the *asynchronous model*, which makes no assumptions about process execution speeds and message delivery delays.[9,10] Proactive secret sharing protocols are tricky to design. First, 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.) And, second, the protocol must work correctly in the presence of as many as t compromised servers, which might provide bogus shares to the protocol.

Server key refresh

Secure communication channels between servers are required for proactive secret sharing and for various other protocols that servers execute. Because a host stores the keys used to implement the secure channels with which it communicates, 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.

Trusted hardware

Although not in widespread use today, special-purpose cryptographic hardware that stores keys and performs cryptographic operations (encryption, decryption, and digital signing) 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 this hardware eliminates the need to refresh server keys as part of proactive recovery for as long as that hardware can be trusted.

However, using special-purpose cryptographic hardware for all cryptographic operations doesn't 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 against such attacks is to maintain an integer counter in stable memory (so that the counter's value will persist across failures and restarts) that's 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. However, recent announcements from industry groups like the Trusted Computing Group (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.

Offline 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 offline 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 offline and is known only to H's administrator. Each new server private key k_A for a host A would be generated offline. 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.

Attack awareness

Instead of relying on a full-fledged tamper-proof coprocessor, a scheme suggested by Ran Canetti and Amir Herzberg[11] uses nonmodifiable storage (such as 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. A certificate that associates the new public key with server H, signed using the special service private key, represents the endorsement.

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 servers can detect bogus certificates using the service public key stored in their ROM. A server becomes aware of an attack if it doesn't receive a valid certificate for its new public key within a reasonable amount of time or if it receives two conflicting requests that are both signed by the same server's private key during the same window of vulnerability. In either case, system administrators should implement actions in order to re-introduce the server into the system and remove the possible imposter.

Processor independence

We approximate the processor independence assumption to the extent that a single attack or host failure cannot compromise multiple hosts. Independence is reduced, for example, when hosts

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

One general way to characterize a service's trustworthiness is by describing which sets of components could together be compromised without disrupting the service's correct operation. Each vulnerability V partitions server replicas into groups, in which replicas in a given group share that vulnerability. For instance, attacks exist that compromise server replicas running Linux but not those running Windows (and vice versa), which leads to a partitioning according to the OS; 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 for the correct operation of the service can be specified using an *adversary structure*.[12,13] This is a set $\mathcal{A} = \{S_1, \ldots, S_r\}$, whose elements are sets of system servers that we assume the adversary can 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 OS would contain sets S_i, whose elements are servers all running the same OS.

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.[5] The basic state-machine approach described earlier 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 isn't 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, because doing so requires identifying all common vulnerabilities. Today's systems often employ commercial off-the-shelf (COTS) components, so access to their 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. Constructing 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 doesn't 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 is 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. Moreover, experiments have shown that distinct development groups working from a common specification will produce software that has the same bugs.[14]

Employ pre-existing diverse components. Here, system developers use 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.[2,15] One difficulty is in procuring diverse components that do have the requisite similar functionality. Some OSs have multiple, diverse implementations (for example, BSD Unix versus Linux), but other OSs do not; application components used in building a service are unlikely to have multiple diverse realiza-

Table 1. Systems that employ elements of distributed trust.

SYSTEM	DESCRIPTION
BFS[28]	An NFS file system implementation built using the BFT toolkit (see Table 2 for a description of the toolkit).
Cornell Online Certificate Authority (COCA)[23]	COCA is a trustworthy distributed certification authority. It avoids consensus protocols by using a Byzantine quorum system, which employs threshold cryptography to produce certificates signed by the service, using proactive recovery conjunction with offline administrator keys for maintaining authenticated communication links. COCA assumes the asynchronous model.
Cornell Data Exchange (CODEX)[33]	CODEX is a robust and secure distribution system for confidential data. It stores private keys using secret sharing with proactive refresh, uses threshold cryptography, and employs a distributed blinding protocol to send confidential information from the service to a client or another distributed service. CODEX assumes the asynchronous model.
E-Vault[34]	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.

tions. A second difficulty arises when components don't provide access to internal nondeterministic choices they make during execution (such as creating a "handle" that will be returned to a client), which makes writing the wrapper quite difficult.[15] 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 John Knight and Nancy G. Leveson's work[24]) or because they are built using some of the same components or tools.

Introduce diversity automatically during compilation, loading, or in the runtime environment.[16,17] Code can typically be generated and storage allocated in several ways for a given high-level language program; making choices in producing different executables introduces 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.

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*[18] 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 that satisfy certain assumptions.[19] In particular, deterministic consensus protocols don't exist for systems with unboundedly slow message delivery or process execution speeds—that is, systems satisfying the asynchronous model. This limitation arises because, to reach consensus in such a system, participating state-machine replicas must distinguish between those replicas that have halted (due to failures) and thus should be ignored, and those replicas that, although 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. Various robust storage systems[20–22] have been structured in this way, as have richer services such as the Cornell online certification authority (COCA; detailed in Table 1),[23] which implements operations involving both reading and writing service state.

To constitute a quorum system, servers are associated with groups (where 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—one operation's effect 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*,[13] involving larger groups and a larger intersection, is required.

A second example of abandoning consensus replication can be seen in the Asynchronous Proactive Secret Sharing (APSS) protocol.[10] Here, each participating server computes a new sharing of some secret; a consensus protocol would seem the obvious way for all correct servers to agree on which new sharing to adopt. But instead, in APSS, each server embraces all of the new sharings; a consensus protocol for the asynchronous model is then not needed. Clients of APSS refer to individual shares by using names that tell a server which sharing is involved. So here, 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 and is 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 sometimes can be an option, albeit one that is too rarely considered.

Option II: Employ randomization

Mike Fischer and colleagues' impossibility result[19] doesn't rule out protocols that use randomization, and practical randomized asynchronous Byzantine agreement protocols have been developed. One example is Cristian Cachin and colleagues' consensus protocol,[24] which builds on some new cryptographic primitives, including a noninteractive threshold signature scheme and a threshold coin-tossing scheme; the protocol is part of the Secure Intrusion-Tolerant Replication Architecture (Sintra) toolkit[25] 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 a consensus protocol's liveness property might temporarily be relaxed in those circumstances. After all, there are no real-time guarantees in the asynchronous model. The crux of this option, then, is to employ a consensus protocol that satisfies its liveness property only while the system satisfies assumptions somewhat stronger than found in the asynchronous model but that always satisfies its safety property (so that different state-machine replicas still agree on the requests they process). Leslie Lamport's Paxos protocol[26] is a well-known example of trading liveness for the weaker assumptions of the asynchronous model. Other examples include Gregory Chockler, Dahlia Malkhi, and Mike Reiter's protocol[27] and BFT.[28]

Computing with server confidential data

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

Adversaries can gain access to information stored unencrypted on a server 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 are now supported[29]). 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 multiparty computations*.[30] Much is known about what can and cannot be done as a secure multiparty computation; less is known about what is practical, and the prognosis is not good for efficiently supporting arbitrary computations (beyond cryptographic operations like decryption and signing).

It's 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 that the service here has no way to control which principals are able to access unencrypted confidential data.

In cases in which we desire the service—and not the 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 the face of server compromises and failures.

Two schemes have been proposed for clients to retrieve encrypted data:

Table 2. Toolkits for implementing distributed trust.

SYSTEM	DESCRIPTION
BFT[28]	BFT is 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 each replica's code and state. 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 coprocessor and a watchdog timer. BFT doesn't provide support for storing confidential information or for maintaining a service private key that is required for scalability.
Intrusion Tolerance via Threshold Cryptography (ITTC)[35]	The ITTC toolkit includes a threshold RSA implementation with distributed key generation and share refreshing, which 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[21]	Phalanx is middleware for implementing scalable persistent survivable distributed object repositories. In Phalanx, a Byzantine quorum system allows Byzantine failures to be tolerated, even in the asynchronous model. Randomized 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)[36]	This is a toolkit for maintaining proactively secure communication links, private keys, and data storage in synchronous systems. The design employs the attack-awareness approach (with ROM) for refreshing the servers' public–private key pairs.
Secure INtrusion-Tolerant Replication Architecture (Sintra)[25]	Sintra is 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.

- *Re-encryption.* A re-encryption protocol produces a ciphertext encrypted under one key from a ciphertext encrypted under another and does so without the plaintext becoming available during intermediate steps. Such protocols exist for public-key cryptosystems in which the private key is shared among a set of servers.[31] To retrieve a piece of encrypted data, the service executes a re-encryption protocol on data encrypted under the service public key; the result is data encrypted under the public key of an authorized client.
- *Blinding.* A client chooses a random blinding factor, encrypts it using the service public key, and sends that to the service. If the service deems that client authorized for access, then the service multiplies the encrypted data by this blinding factor and then employs threshold decryption to compute unencrypted but blinded data, which is then 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 it's essentially encryption with a one-time pad (the random blinding factor). Unlike the re-encryption scheme in Marcus Jakobsson's work,[31] 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 in which a client itself is a distributed service with a service public key, whereas the blinding-based scheme cannot be used without further modification. In fact, a re-encryption scheme based on blinding appears in other work;[32] in it, ciphertext encrypted under the service public key is transformed into ciphertext encrypted under the client public key (as with the re-encryption scheme in Jakobsson's work), thereby allowing a flexible partition of work between client and service.

Tables 1 and 2 summarize the various systems that have been built using the elements we've just outlined. Clearly, there's much to be learned about how to engineer systems based on these elements, and only a small part of the landscape has been explored.

A system's trustworthiness is ultimately tied to a set of assumptions about the environment in which that system must function. Systems users should prefer weaker assumptions, because then there is less risk that these assumptions will be violated by natural events or attacks. However, adopting this view renders irrelevant much prior work in fault-tolerance and distributed algorithms.

Until recently, the synchronous model of computation has generally been assumed, but there are good reasons to

investigate algorithms and system architectures for asynchronous models of computation: specifically, concern about DoS attacks and interest in distributed computations that span wide-area networks. Also, 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 state or state transitions, usual approaches to specification and system refinement, which are concerned with what actions components perform, are not germane.

Our growing dependence on networked computers makes us hostage not only to failures but also to attacks.

The system design approach outlined in this article has been referred to as implementing *distributed trust*[37] 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 has 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 20 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 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 also seen today 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 article were provided by Martin Abadi, Ulfar Erlingsson, Andrew Myers, Mike Schroeder, Gun Sirer, and Ted Wobber. We're also very grateful to three anonymous reviewers for insightful suggestions and pointers to literature we overlooked, all of which helped us 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.

This work was supported in part by ARPA/RADC grant F30602-96-1-0317, Air Force Office of Scientific Research grant F49620-03-1-0156, DARPA and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, US National Science Foundation grants 9703470 and 0430161, and grants from Intel and Microsoft. 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 US government.

References

1. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, 1978, pp. 558–565.
2. F.B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990, pp. 299–319.
3. A. Herzberg et al., "Proactive Secret Sharing or: How to Cope with Perpetual Leakage," D. Coppersmith, ed., *Advances in Cryptology: Proc. 15th Int'l Cryptology Conf.*, LNCS 963, Springer-Verlag, 1995, pp. 457–469.
4. G.R. Blakley, "Safeguarding Cryptographic Keys," *Proc. 1979 AFIPS National Computer Conf.*, vol. 48, R.E. Merwin, J.T. Zanca, and M. Smith, eds., AFIPS Press, 1979, pp. 313–317.
5. A. Shamir, "How to Share a Secret," *Comm. ACM*, vol. 22, no. 11, 1979, pp. 612–613.
6. C. Boyd, "Digital Multisignatures," *Cryptography and Coding*, H. Baker and F. Piper, eds., Clarendon Press, 1989, pp. 241–246.
7. Y. Desmedt and Y. Frankel, "Threshold Cryptosystems," *Advances in Cryptology: Proc. 9th Int'l Cryptology Conf.*, LNCS 435, G. Brassard, ed., Springer-Verlag, 1990, pp. 307–315.
8. R. Ostrovsky and M. Yung, "How to Withstand Mobile Virus Attacks," *Proc. 10th Ann. Symp. Principles of Distributed Computing* (PODC 91), ACM Press, 1991, pp. 51–59.
9. C. Cachin et al., "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems," *Proc. 9th ACM Conf. Computer and Comm. Security*, ACM Press, 2002, pp. 88–97.
10. L. Zhou, F.B. Schneider, and R. van Renesse, "APSS: Proactive Secret Sharing in Asynchronous Systems," *ACM Trans. on Information and System Security*, vol. 8, no. 3, 2005, pp. 1–28.

11. R. Canetti and A. Herzberg, "Maintaining Security in the Presence of Transient Faults," *Advances in Cryptology: Proc. 14th Int'l Cryptology Conf.*, LNCS 839, Y. Desmedt, ed., Springer-Verlag, 1994, pp. 425–438.
12. M. Hirt and U. Maurer, "Player Simulation and General Adversary Structures in Perfect Multiparty Computation," *J. Cryptology*, vol. 13, no. 1, 2000, pp. 31–60.
13. D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, no. 4, 1998, pp. 203–213.
14. J. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Trans. Software Eng.*, vol. SE–12, no. 1, 1986, pp. 96–109.
15. R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance," *Proc. 18th ACM Symp. Operating System Principles*, ACM Press, 2001, pp. 15–28.
16. S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," *Proc. 6th Workshop Hot Topics in Operating Systems*, IEEE CS Press, 1997, pp. 67–72.
17. J. Xu, Z. Kalbarczyk, and R.K. Iyer, *Transparent Runtime Randomization for Security*, tech. report UILU-ENG-03-2207 (CRHC-03-03), Ctr. for Reliable and High-Performance Computing, Univ. of Illinois at Urbana-Champaign, May 2003.
18. M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM*, vol. 27, no. 2, 1980, pp. 228–234.
19. M.J. Fischer, N.A. Lynch, and M.S. Peterson, "Impossibility of Distributed Consensus with One Faulty Processor," *J. ACM*, vol. 32, no. 2, 1985, pp. 374–382.
20. B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection," *Proc. 5th ACM Symp. Principles of Distributed Computing*, ACM Press, 1986, pp. 29–39.
21. D. Malkhi and M. Reiter, "Secure and Scalable Replication in Phalanx," *Proc. 17th Symp. Reliable Distributed Systems*, IEEE CS Press, 1998, pp. 51–58.
22. J.J. Wylie et al., "Survivable Information Storage System," *Computer*, vol. 33, no. 8, 2000, pp. 61–68.
23. L. Zhou, F.B. Schneider, and R. van Renesse, "COCA: A Secure Distributed Online Certification Authority," *ACM Trans. Computer Systems*, vol. 20, no. 4, 2002, pp. 329–368.
24. C. Cachin, K. Kursawe, and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography," *Proc. 19th ACM Symp. on Principles of Distributed Computing* (PODC 00), ACM Press, 2000, pp. 123–132.
25. C. Cachin and J.A. Poritz, "Secure Intrusion-Tolerant Replication on the Internet," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 02), IEEE CS Press, 2002, pp. 167–176.
26. L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems*, vol. 16, no. 2, 1998, pp. 133–169.
27. G. Chockler, D. Malkhi, and M.K. Reiter, "Backoff Protocols for Distributed Mutual Exclusion and Ordering," *Proc. Int'l Conf. Distributed Systems*, IEEE CS Press, 2001, pp. 11–20.
28. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Computer Systems*, vol. 20, no. 4, ACM Press, 2002, pp. 398–461.
29. D. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," *Proc. 2000 IEEE Symp. Security and Privacy*, IEEE CS Press, 2000, pp. 44–55.
30. O. Goldreich, S. Micali, and A. Wigderson, "How to Play ANY Mental Game," *Proc. 19th Conf. Theory of Computing* (STOC 87), ACM Press, 1987, pp. 218–229.
31. M. Jakobsson, "On Quorum Controlled Asymmetric Proxy Re-encryption," *Public Key Cryptography: Proc. 2nd Int'l Workshop on Practice and Theory in Public Key Cryptography* (PKC 99), LNCS 1560, H. Imai and Y. Zheng, eds., Springer-Verlag, 1999, pp. 112–121.
32. L. Zhou et al., "Distributed Blinding for ElGamal Re-encryption," *Proc. 25th Int'l Conf. Distributed Computing Systems*, IEEE Press, 2005, pp. 815–824.
33. M.A. Marsh and F.B. Schneider, "CODEX: A Robust and Secure Secret Distribution System," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2003, pp. 34–47.
34. J.A. Garay et al., "Secure Distributed Storage and Retrieval," *Theoretical Computer Science*, vol. 243, nos. 1–2, 2000, pp. 363–389.
35. T. Wu, M. Malkin, and D. Boneh, "Building Intrusion Tolerant Applications," *Proc. 8th Usenix Security Symp.*, Usenix Assoc., 1999, pp. 79–91.
36. B. Barak et al., "The Proactive Security Toolkit and Applications," *Proc. 6th ACM Conf. Computer and Comm. Security* (CCS 99), ACM Press, 1999, pp. 18–27.
37. M.K. Reiter, "Distributing Trust with the Rampart Toolkit," *Comm. ACM*, vol. 39, no. 4, 1996, pp. 71–74.

Fred B. Schneider *is a professor at Cornell's computer science department and director of the AFRL/Cornell Information Assurance Institute. In addition to chairing the US National Research Council's study committee on information systems' trustworthiness and editing* Trust in Cyberspace (National Academy Press)*, Schneider is co-managing editor of Springer-Verlag's* Texts and Monographs in Computer Science*, associate editor in chief of* IEEE Security & Privacy*, and a member of several other journal editorial boards. He also co-chairs Microsoft's Trustworthy Computing Academic Advisory Board. Contact him at fbs@cs.cornell.edu.*

Lidong Zhou *is a researcher at Microsoft Research Silicon Valley. His main areas of research interests are distributed systems, computer networks, and wireless communication, with a focus on security and fault tolerance. Zhou has a PhD in computer science from Cornell University. Contact him at lidongz@microsoft.com.*