Trust but Verify: Accountability for Network Services

Aydan R. Yumerefendi and Jeffrey S. Chase
Department of Computer Science
Duke University
{aydan,chase}@cs.duke.edu

Abstract

This paper promotes *accountability* as a central design goal for dependable networked systems. We define three properties for accountable systems that extend beyond the basic security properties of authentication, privacy, and integrity. These accountability properties reduce the vulnerability of network services to subversion, tampering, corruption, and abuse. For example, actions taken in accountable systems and their clients are provable or even legally binding, to support contractual relationships in federated systems.

We propose a framework for accountable network services, and explore its applicability and limitations. The foundation of our approach is to preserve digitally signed records of actions and/or internal state snapshots of each service, and use them to detect tampering, verify the consistency of actions and behavior, and prove responsibility for unexpected states or actions. We outline the key challenges in generalizing the principles and methodology of accountable design for practical use.

1 Introduction

System builders have powerful tools at their disposal to build secure network services on today's insecure networks. Basic support for secure sockets and other uses of public key cryptography is now widely deployed. Communication between clients and servers is increasingly authenticated and encrypted for important services such as e-mail, Web commerce, and remote access.

But secure communication alone is not sufficient for real security. This reality is increasingly evident as services grow more complex and interdependent. Today's services often consist of interacting components or peers, which may span trust domains. In a utility, grid, or P2P setting they may run on servers controlled by untrusted third parties, where their code and data may be altered. Secure communication by itself offers little protection if one or more components are compromised.

In this paper we define and explore new *accountability* properties for network services. These properties help to determine if a system is functioning correctly and to assign responsibility if things go wrong. They can contain the damage from an attacker, and block a compromised component from subverting other components that depend on it. Accountability is particularly important when services and their clients act on behalf of real-world principals—people and organizations—to enter into and fulfill contractual responsibilities in their various roles. It is essential for safe deployments of mission-critical services in areas such as health care, infrastructure control, government, commerce, and finance.

We argue that system builders should view accountability as a first-class design goal of services and federated distributed systems, alongside other properties that have been the subject of intensive study: performance, availability, dependability, and so on. One contribution of this paper is to outline the elements of a methodology for building accountable network services that can apply to a wide class of services. These elements build upon cryptographic primitives including secure hashes and digital signatures. Many of the basic techniques of accountable design have been pioneered by research on secure use of untrusted storage (e.g., [9]), authenticated or self-certifying data structures [10, 12, 3, 1, 7], and trustworthy implementations of fundamental security services such as timestamping and key archiving [8].

2 Overview

For our purposes, a *service* is an instance of a program that executes requests from clients according to some request/response interface definition. For example, it might be a classical RPC service, such as a network file service, or a SOAP/XML Web service. A service has internal state that drives its behavior. Client operations transform and/or query this state; thus a service acts as a focal point for its clients to interact and share resources or data. Services may act as clients of other services.

A number of techniques exist to protect, validate, and assign responsibility in distributed systems. These techniques have varying power, generality and cost. For example, checks for type safety and acceptable parameter values protect against incorrect invocation (e.g., buffer overflow), but they cannot defend against valid operations that are issued in error or with malice. It is common to establish a wider "secure" perimeter around cooperating components, e.g., by authenticating their communication and/or isolating the network from dangerous traffic using firewalls. But perimeter defenses are inadequate if the components reside in different security domains, and they do not prevent a subverted component from infecting or manipulating the others.

2.1 Accountability Properties

Accountability denotes assurances of *semantic* behavior that extend beyond basic perimeter security and the mechanisms for message authentication, encryption, and integrity. A real-world example of an accountable service is certified mail. Encryption can provide for the privacy and integrity of the mail; certified mail goes beyond that in holding the postal service accountable for accepting a message for transmission, and in holding the receiver accountable for accepting its delivery.

More specifically, the behavior, state, and actions of accountable systems should be:

- **Undeniable.** Actions of an accountable actor are provable and non-repudiable. That is, a service or its clients cannot plausibly deny their actions, and those actions may be legally binding.
- **Certifiable.** A client, peer, or external auditor may verify that an accountable service is behaving correctly, and prove any misbehavior to an arbitrary third party. For example, a service may be prompted to prove cryptographically that its actions are justified by the sequence of operations issued by its clients, in accordance with its defined semantics.
- **Tamper-evident.** Any attempt to corrupt the service state incurs a high probability of detection. In particular, an external auditor may determine if the internal state could or could not result from the sequence of operations issued on the service.

Secure hashes and digital signatures are fundamental technologies to enable these properties. We assume that all entities have the means to digitally sign and verify accountable actions with public/private key pairs. The certified mail example illustrates the role of such signed “receipts”: participants retain signed, timestamped records of actions by other participants. In the United States, national legislation in 2000 affirmed the legal validity of digital signatures when keys are securely bound to accountable principals; a public key infrastructure (PKI) is one way to meet this requirement, but it is not the only way. Digital signatures are practical for use at the granularity of individual requests and responses, particularly for critical services in wide-area networks. The choice of key length always allows a continuum of trade-offs of security and overhead, independent of advancing processor speeds.

Our approach is to maintain digitally signed records of the actions and/or internal state snapshots of each entity, and use them to detect tampering and to verify the consistency of actions and behavior. The ultimate goal is to enable a server to present a sequence of signed actions that produced a given server state or action, according to action rules specified for the service. Accountability for a service implies accountability for its clients, to the extent that it is possible to determine whether and how the service state and responses resulted from the actions of its clients, and even to generate a proof that is sufficient to assign legal responsibility for the state or action.

It has been observed that most cryptosystem failures occur when insiders leak or abuse the keys representing a principal [2]. One goal of our approach is to limit the damage an attacker can inflict if it subverts some component, e.g., by manipulating its state or even by stealing its private key. Although a principal is always accountable for actions taken using its key, integrating signed action records into the service data structures can prevent a compromised service from misrepresenting the actions of its clients, as well as preventing the clients from repudiating or denying their actions. This is an important form of *defense in depth* to supplement perimeter defenses [17].

2.2 Example: SHARP

Accountability is necessary to support contracts and enforcement in federated distributed services that involve exchanges of goods or services across trust domains. As a concrete example, consider our recent work with SHARP [4], an architecture for secure federated resource peering in PlanetLab and grid systems. SHARP defines a framework to delegate control over resources at distributed sites to a collection of pluggable resource managers (brokers or *agents*), creating a foundation for a decentralized resource economy. Agents issue signed XML contracts (*claims*) to users or other agents to control shares of site resources for specified time intervals.

SHARP enforces the accountability of agent allocations to protect the system and its users from abuse. For example, suppose an agent fraudulently commits the same resources to more than one entity for overlapping time intervals. This *oversubscription* of resources has a legitimate role in improving availability in SHARP, so the system does not seek to prevent it. However, resource allocation in SHARP is accountable in the following sense: if oversubscription results in a broken contract, the SHARP site that owns the resource can identify the responsible agent and issue a cryptographic proof that the agent has entered into contract obligations that exceed its resource holdings. This property makes it possible for a third party such as a legal authority or reputation service to hold the offending agent accountable by imposing some sanction.

To maintain accountability, each SHARP site retains records of redeemed claims signed by the agent that issued them. If a client attempts to redeem a conflicting claim, the site can identify the accountable agent and prove its guilt from the aggregation of signed, timestamped claims issued by that agent to all of its clients and subsequently redeemed at the site for the relevant resource set and time interval. Sites organize redeemed claims in an internal tree structure that supports efficient retrieval of the relevant claim records.

This example of accountability—and the means to support it—is specific to the SHARP system. In some respects, the SHARP context is simpler than other service examples: an agent’s action to issue a claim is irrevocable until the claim expires, at which point all state related to it may be discarded. Moreover, similar techniques are not applicable to enforce accountability of the SHARP sites themselves with respect to their primary function: to accept valid claims and deliver the promised resource. The only obvious way to certify compliance is to use some form of monitoring by a trusted third-party auditor (such as Keynote). Even so, the principles of accountability for SHARP agents can extend to other important contexts.

2.3 Toward More General Accountability

We are interested in the more general class of data-driven services whose operation semantics are specified as transformations to their internal state, together with queries on states resulting from the actions of other clients. These services can often self-certify that they are functioning correctly by exposing actions, events, and internal states that allow external observers to verify their behavior. Ideally, the service can offer evidence of correctness as annotations to its responses. Servers may also be subject to asynchronous auditing with some frequency chosen to balance auditing cost and the assurance of exposure if the service is compromised.

A number of techniques exist to address the need for more general defense against subversion. At one extreme, secure hardware uses physical security to guarantee untampered execution [18, 16]. This is a powerful approach that makes no changes to applications, but it has a high impact on hardware and operating systems. At the other extreme, Byzantine fault tolerant (BFT) systems (e.g., [11, 14]) use replication and voting to identify and isolate compromised servers. BFT applies to services executing arbitrary computations, potentially on untrusted hardware, and the service can continue to function if a quorum of servers is correct.

Both of these approaches are general but also expensive. Neither can defend against vulnerabilities in the software itself, or assure accountability of actions by the software. For example, BFT is subject to collusion among a quorum of faulty replicas. Accountable design can provide a stronger means to detect and isolate faults, and to contain them without relying on replication and voting; indeed, it may be viewed as an approach to Byzantine failure *detection*. A key limitation of accountable design is that it is not fully general: it must be "designed in" to application structure and protocols, and it can detect failures only when operation semantics are well-specified and relatively cheap to verify. Even so, accountable design appears feasible and promising in key contexts where it is most essential, including key infrastructure services for federated distributed systems.

3 A Framework for Accountable Services

The elements of accountable design include externally published digests of internal service state, annotations that prove the correctness of actions relative to published states, and auditing interfaces to guard against replay attacks or discarding of accepted actions. Secure logging and auditing [15] can help maintain and verify the execution history of components.

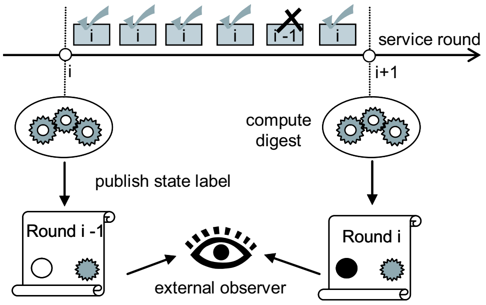

Figure 1: Round Processing. The service processes sequences of requests in rounds. All accepted requests bear a timestamp matching the round. At the end of the round the service computes a state digest and publishes it externally.

This section outlines a general framework for accountable services, building on the approach used in KASTS [8] to build a secure key archive. The goal is to enable strong accountability for a general class of services that access and update internal state in response to invocations from clients and peers. We make the following assumptions for simplicity. The state consists of an indexed set of named, typed, data objects. Execution proceeds in a sequence of numbered *rounds*: the service accepts at most one update to any given state variable in each round, and retrieval operations execute from the values at the start of the current round. At the end of each round the service publishes a signed, timestamped, nonrepudiable digest of its internal state to external observers, as depicted in Figure 1.

Figure 2 depicts request handling in our framework. Each client request is timestamped and digitally signed by the client. The signature makes the request unforgeable and nonrepudiable, and the timestamp helps to defend against replay or reordering attacks. The service returns a signed response annotated with evidence that it executed the request correctly, as described below.

To compute state digests efficiently, the service may organize its internal state as an authenticated data structure such as a Merkle tree or other secure hash tree. A prominent example is a *persistent authenticated search tree* (PAST) [1], which maintains object references and indexes in a self-certifying hash tree, and retains a tamper-evident persistent version history of updates to the tree. Like other hash tree variants, a PAST recursively annotates each node of the tree with a hash over its children, using a *one-way*, *collision-resistant* hash function such as SHA-1. The root hash certifies the entire state of the index and the data objects that it references. To generate a state digest, the service appends the current root hash with a round timestamp and digitally signs it.

The state digests play a role in enabling other participants to certify that the service accurately represents and updates its state. Consider how a PAST can allow par-

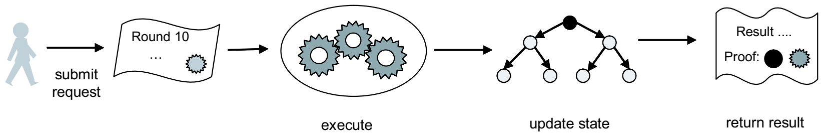

Figure 2: Request Execution. Clients digitally sign each request and its timestamp. The service verifies and executes the request, updates its state, and returns a result annotated with cryptographic evidence to certify its correctness relative to the published round digests.

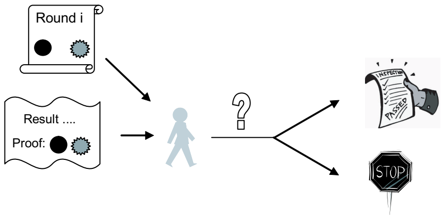

Figure 3: Response Verification. Clients verify the self-certifying responses by checking their evidence against the published digests.

ticipants to verify correct execution histories for a simple storage service. On each update, the service retains the signed request, applies it, and returns a proof that the next state digest reflects the update. The proof includes the set of node hashes along the path from the element's position in the tree to the root (the round digest), including sibling hashes used to compute the hash of each node. It is computationally infeasible to synthesize a false set of hashes that combine with the new element hash to produce the root hash, so the service must incorporate all updates for the round into its digest. On retrievals, the PAST can return a similar chain of node hashes as a *membership proof* to show that a given named element is or is not present in the tree at a given round. If the element is present, the service can prove that its value was reflected in the published digest, and enclose a timestamped write hash signed by the issuing client to prove that the value derives from a valid write in some previous round. Any client can verify these proofs against the published digests (Figure 3).

This approach requires that round digests are visible to all participants. A service may try to cheat by presenting different digests for the same round to different participants. This is a generalization of a *forking attack* [9], in which the server hides the actions of one or more clients from the others. A trusted publishing medium or secure broadcast could detect or prevent such an attack. Alternatively, the service may include recent digests in its messages; a collaborative auditing approach can detect inconsistencies, as outlined below.

3.1 The Freshness Problem and Auditing

A key limitation of authenticated data structures is that they do not ensure that updates persist in subsequent rounds. For example, a compromised service might execute and certify an update in a given round, but revert to a previous value in a later round (a form of replay attack), or even remove the element completely. A correct service cannot prove its correctness: it cannot easily certify that its responses are fresh, i.e., that no subsequent update supersedes the data used to generate the response. Maniatis [7] suggests that no data structure exists that can certify freshness, although this has not yet been proven.

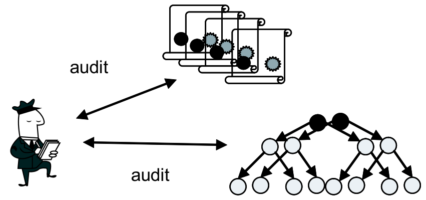

Figure 4: Auditing. To ensure long-term integrity, an external auditor inspects the service state and verifies its digests.

We propose to address the freshness problem with probabilistic auditing. Clients or an external auditor (Figure 4) may request a proof that the service maintained any element's value correctly across any interval of time. The service returns the element's value for each round, together with a set of hashes proving that the non-repudiable published round digest derived from the element's asserted value. If the element's value changed in a round, the service must present a signed valid write with a timestamp for that round; thus the service cannot lie to the auditor, and it is accountable for the consistency of its published state digests with respect to the writes it claims to have executed.

Auditing can also verify consistency of published round digests. Each service may publish its digests by including them in messages, e.g., responses to clients. Suppose that receivers exchange digests or forward them to a clearinghouse (e.g., a reputation service) with some probability. Then any receiver can detect any inconsistency among the digests it receives. Since the digests are signed and timestamped, any misbehavior can be proven; the approach is secure against false accusations.

The frequency and depth of auditing presents a tradeoff between a target level of accountability and overhead. Crucially, the auditor need not be trusted: any participant can audit a service and prove any misbehavior to a third party. However, the service must have some mechanism to deny excessive audit requests, or else the audit interface itself could be a vector for a denial of service attack.

4 Application Examples

This section considers three example services to illustrate different aspects of accountability, and issues for generalizing the framework. The basic function of each service is to mediate among clients and to represent their actions to other clients correctly. These primitive services are fundamental to a wide range of applications.

4.1 Storage Service

Consider a service that stores data objects (such as files or blocks) and organizes them in a hierarchical namespace. As in the previous section, the state of the service consists of a state variable for each object, indexed with respect to the namespace hierarchy. The implementation may organize any block map or directory hierarchy as a tree, and propagate updates through to the root; indeed, similar tree structures are the basis for atomic snapshots in industrial-strength file systems such as WAFL [5]. Thus it should be possible to build an accountable storage service by decorating the tree with hash digests and applying the framework directly.

SUNDR [9] and Plutus [6] are two recent storage systems for untrusted servers. These services are safe in the sense that clients may protect data from the server, and the clients can detect if the server modifies or misrepresents the data. Plutus emphasizes efficient support for encrypted sharing, while SUNDR defends against sophisticated forking attacks. We argue for a stronger notion of accountability in which the guilt or innocence of the server *or its clients* is provable to a third party. For example, our proposal defends against false accusations, e.g., a malicious client that "frames" the service by claiming falsely that the server accepted writes and later reverted them. Accountability extends to the clients: for example, if a client corrupts shared data, the server can identify the client and prove its guilt even after accepting writes to other parts of the tree. Moreover, published root digests subsume the need for clients to interact directly to compare their views of the stored data, e.g., to detect reverted writes or a forking attack as in SUNDR.

4.2 Authorization Service

One limitation of the storage service is that it does not certify that a client was authorized to issue a write; in fact the service could inject false writes by "making up" clients to certify them, as a form of sybil attack. Thus a comprehensive solution must also enable accountable authorization and access control. Accountability is also desirable for standalone authorization services (e.g., the Globus Community Authorization Service [13]), which are trusted and vulnerable to attack.

An authorization service is similar to a storage service, but the operations and response evidence are more complex. As a starting point, we can formalize the internal data as a set of boolean variables, one for every possible $(principal, object, right)$ triple. Then authorization actions and queries reduce to accountable reads and writes on these variables. But this is not enough: any write to such a variable must itself be subject to authorization, and any proof of validity for a query response must also prove that the variable's most recent write was authorized, and that this authorization is fresh.

We are investigating extensions to the framework to support such chains of proof, and their impact on auditing. A key challenge is that many access control schemes include notions of subject groups, object groups, or roles, which complicate these proof chains. For example, the justification to allow access includes (at minimum) a signed request from a granting principal to grant access for the subject and object, a signed action associating the requesting principal and its public key with that subject, and a signed action authorizing the granting principal to control access policy for the object. To fully justify a denial the system must prove that the principal is *not* associated with any subject or role enabled for the requested right. It is an open question whether fully accountable denials are practical, or whether we must rely on probabilistic auditing or more targeted challenges from the client to trap a server that falsely denies access. We are focusing on certifying the responses that allow access, which are the most critical for an accountable security policy.

4.3 Certified State Machine

We now describe a general template for a class of services that incorporate simple computations as well as storage and queries. The internal state comprises a set of named, typed elements, each with a finite state variable. Each element type defines a set of operations or events, whose semantics are given by a deterministic state machine defining state transition rules for each event. An event may include arbitrary data, which is maintained in an event history.

To illustrate, consider an application service to manage and automate workflow in a university department. The service models each student as a state element. Authorized clients (e.g., faculty members) use the service to apply transitions associated with events such as course completion, exams, defenses, etc. If an element reaches a designated final state, then the service can prove to the university registrar that (for example) the student has met the requirements for a degree. The proof is self-certifiable and undeniable. The student records are tamper-evident: even if an attacker compromises the database, it cannot falsify the records without detection unless it also steals the private keys of a quorum of faculty members.

For full accountability as defined in the framework, each operation on an element should return some proof of correct execution. We can partially reduce the problem to the primitive storage case by including a timestamp field in each element to store the round number of its most recent transition. On each event, the server transitions the element to its new state, updates its timestamp, and returns a set of hashes showing that the round digest reflects the new state and timestamp. Even if the client does not verify that the state is correct, it knows that the service cannot hide the update from an auditor. If the service is audited for that element and round, then it must present the signed operation to justify the state change to the auditor, which can verify the transition. (Note that the auditor must know the transition rules.)

Reads are also similar to the primitive storage case, with a crucial difference that illustrates the impact of incorporating computation into an accountable service in the framework. On a read, the service can prove that the returned element state was reflected in the previous round digest as before. However, the reader can no longer verify that the element's state derived from a recent write event without also verifying its previous state; it must now rely on auditing to verify the integrity of the state transitions as well as freshness.

5 Conclusions and Future Work

The increasing dependence on mission-critical services requires in-depth protection that goes beyond the traditional defenses of authentication and authorization. The accountability properties studied in this paper reduce the vulnerability of network services and help identify malicious entities; actors in an accountable system cannot repudiate or deny actions and are legally responsible for any operation they commit.

Accountable systems expose interfaces that allow them to verify the correctness of their actions. Services in our framework maintain signed records of operations and state transitions, and use them to justify actions and systems states. The practical applicability of this approach to systems with very general and complex state transitions is doubtful, however, we have shown examples of basic infrastructure services for which accountability is closer to reality. Accountability in these primitive services is an essential building block for accountability in more complex systems.

Currently we are working on estimating the cost of the accountability techniques and their relevance to real-life scenarios. We are also exploring the relationship between auditing rates, overhead, and levels of accountability, based on prototypes of the example services described. Quantifying this relationship can offer different mechanisms for environments with varying requirements.

Acknowledgements

We thank Petros Maniatis for discussions of the ideas in this paper, which offered useful insights on accountability. We also thank the anonymous reviewers, whose comments helped to improve the paper.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security, ISC,* October 2001.

[2] R. J. Anderson. Why Cryptosystems Fail. *Comunications of the Association for Computing Machinery*, 37(11), Nov. 1994.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable Certificate Management Using Undeniable Attestations. In *Proceedings of the 7th ACM Conference of Computer and Communications Security*, pages 9–17, 2000.

[4] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 133–148, October 2003.

[5] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Annual Technical Conference*, pages 235–246, January 1994.

[6] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the Second Conference on File and Storage Technologies (FAST 03)*, March 2003.

[7] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, August 2003.

[8] P. Maniatis and M. Baker. Enabling the Archival Storage of Signed Documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 31–45, January 2002.

[9] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[10] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Sysmposium on Security and Privacy*, pages 122–133, April 1980.

[11] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of Symposium on Operating Systems Design and Implementation,* February 1999.

[12] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[13] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, June 2002.

[14] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[15] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, May 1999.

[16] S. W. Smith, E. R. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89, 1998.

[17] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 165–180, October 23-25 2000.

[18] B. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.