System Support for Strong Accountability

by

Aydan Rafet Yumerefendi

Department of Computer Science
Duke University

Date: __________________________

Approved:

_____________________________
Dr. Jeffrey S. Chase, Supervisor

_____________________________
Dr. Landon Cox

_____________________________
Dr. Petros Maniatis

_____________________________
Dr. Jun Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2009
ABSTRACT

System Support for Strong Accountability

by

Aydan Rafet Yumerefendi

Department of Computer Science
Duke University

Date: ____________________________

Approved:

________________________________
Dr. Jeffrey S. Chase, Supervisor

________________________________
Dr. Landon Cox

________________________________
Dr. Petros Maniatis

________________________________
Dr. Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2009
Abstract

Deployed software frequently spans across trust domains and enables the interactions of self-interested participants with potentially conflicting goals. With systems becoming more complex and interdependent there is a growing need to localize, identify, and isolate faults and unfaithful behavior.

Conventional techniques for building secure systems, such as secure perimeters and Byzantine fault tolerance, are insufficient to ensure that trusted users and software components are indeed trustworthy. Secure perimeters are rarely secure and do not extend across trust domains. They offer limited protection when a participant acts within the limits of the existing security policy and deliberately manipulates the system to her own advantage. BFT uses replication and voting to tolerate misbehavior, but it is not sufficient when all replicas are under the control of a single entity or are vulnerable to the same attack.

This thesis addresses the problems of misbehavior and abuse by offering tools and techniques to integrate accountability into computer systems. These accountability techniques complement the existing solutions to identify improper behavior and actions, limit the propagation of incorrect information, and assign responsibility when things go wrong. In particular, we focus on building and understanding the limitations of strongly accountable systems, i.e. systems that offer means to identify and expose semantic misbehavior by their participants by providing undeniable evidence to demonstrate misbehavior—any detected violation can be proven to a third party without making assumptions about the truthfulness of the accuser.

This dissertation shows how to design, implement, and evaluate a generic methodology for integrating accountability into network services and applications, and demonstrates its use to construct a shared accountable file service. Our state-based approach achieves generality by decoupling application state management from application logic to enable services to demonstrate that they maintain their state in compliance with user requests, i.e., state changes do take place, and the service presents a consistent view to all clients and observers. Internal state managed in this way can then be used to feed application-specific verifiers to determine the correctness the service’s logic and to identify the responsible party.

We also study the issues of extending responsibility to end users for actions performed by software
components on their behalf and show ways to provide weaker accountability properties for leaks of sensitive information. To help separate the actions of software and its users we design, implement, and evaluate a novel information flow technique to detect information leaks in a commodity operating system. Our approach makes software accountable for its actions to the user and also prevents the end users from becoming responsible for misbehaving software acting on their behalf.

Our work also explores how to extend accountability to services in which actors enter into explicit contracts that constrain their behavior and must conform to certain application-specific invariants. We study the invariants and accountability requirements of an example application—a lease-based virtual resource economy—and present the design and implementation of several key elements needed to provide accountability in the system. In particular, we describe a solution to the problem of resource delegation of partitionable resources, and present a novel approach to making virtual currency operations accountable.
Acknowledgements

This thesis would not have been possible without the unconditional support of my wife Anca, for which I am deeply grateful and eternally indebted.

My advisor Jeff Chase has always been there for me guiding me through the long and bumpy quest for knowledge and understanding.

Landon Cox provided me with the opportunity to work on the TightLip project at a time when my Ph.D. career looked gloomy and uncertain.

The summer internship I spent working with Petros Maniatis helped spur my interest in the topic of this thesis work.

I have had the pleasure and honor to work with a number of bright people. In particular, I would like to thank David Irwin and Laura Grit for working together on the ORCA project.

Finally, I want to thank my thesis committee for their patience and understanding.
Contents

Abstract iv
Acknowledgements vi
List of Tables xiii
List of Figures xiv
1 Introduction 1
  1.1 Problem ................................................................. 2
    1.1.1 Law-governed Interactions .......................................... 3
    1.1.2 Means and Motives for Abuse ...................................... 3
    1.1.3 Interdependence and Scale ......................................... 4
  1.2 Solution ................................................................. 5
    1.2.1 Design Principle of Dependable Network Services ............... 7
    1.2.2 Focusing Questions ................................................. 8
  1.3 Hypothesis and Contributions ......................................... 10
  1.4 Thesis Overview ....................................................... 11
2 Accountability: Means to an End in Dependable Software Systems 13
  2.1 Accountable Actions ................................................... 13
    2.1.1 Basic Properties .................................................. 16
    2.1.2 Means to System Dependability .................................... 17
    2.1.3 Strong Accountability ............................................. 18
    2.1.4 Digital Signatures ................................................. 19
  2.2 Toward General Accountability ........................................ 20
    2.2.1 State-based Accountability ....................................... 21
    2.2.2 Application-specific Accountability ............................. 21
2.3 Related Work ................................................................. 22
  2.3.1 Theoretical Approaches ........................................... 23
  2.3.2 Secure Perimeters .................................................. 24
  2.3.3 Secure Hardware ................................................... 25
  2.3.4 Byzantine Fault Tolerance ....................................... 26
  2.3.5 Security Standards ............................................... 26
  2.4 Summary ................................................................. 27

3 State-based Accountability ................................................. 28
  3.1 Stateful Network Services .......................................... 28
    3.1.1 Service Model .................................................. 28
    3.1.2 Separation of State and Logic ................................. 32
    3.1.3 Threat Model ................................................... 32
    3.1.4 Trust Assumptions ............................................. 34
    3.1.5 Challenges and Audits ........................................ 36
    3.1.6 Discussion and Limitations .................................. 37
  3.2 State-based Approach ................................................. 38
    3.2.1 Signed Action Histories ...................................... 39
    3.2.2 State Digests and Commitment ............................... 40
    3.2.3 Proofs .......................................................... 41
    3.2.4 Request Ordering and Consistency ........................... 43
    3.2.5 Freshness and Auditing ...................................... 45
    3.2.6 Action Correctness ............................................ 49
  3.3 “Fresh” Untrusted Data Structures ............................... 50
  3.4 Related Work .......................................................... 53
  3.5 Summary ............................................................... 55

4 CATS Toolkit .............................................................. 56
4.1 Overview ......................................................... 56
4.2 Data Structures ................................................... 57
  4.2.1 Cryptographic Accumulators ................................ 57
  4.2.2 Authenticated Dictionaries .................................. 57
4.3 Design Issues ................................................... 59
  4.3.1 Data Layout ................................................ 59
  4.3.2 Tree Degree ............................................... 61
  4.3.3 Data Size ................................................ 63
  4.3.4 Persistence ............................................... 64
  4.3.5 Concurrency ............................................... 67
  4.3.6 Finalization ............................................... 68
  4.3.7 Recovery ................................................. 69
4.4 Implementation ................................................. 70
  4.4.1 Main Memory ............................................... 70
  4.4.2 External Memory ......................................... 71
4.5 Evaluation .................................................... 73
  4.5.1 Methodology .............................................. 74
  4.5.2 Main Memory .............................................. 74
  4.5.3 External Memory ......................................... 80
4.6 Related Work .................................................. 85
4.7 Summary ....................................................... 86

5 CATS Storage Service ................................. 87
5.1 Overview ....................................................... 87
  5.1.1 Threat Model ............................................... 88
  5.1.2 Toolkit-based Design ..................................... 89
5.2 Implementation ............................................... 92
5.3 Evaluation ................................................................. 94
   5.3.1 Methodology ..................................................... 94
   5.3.2 Saturation Throughput ....................................... 95
   5.3.3 Workload Contention ....................................... 97
   5.3.4 Challenges and Audits ..................................... 98
5.4 Related Work ....................................................... 99
5.5 Summary ............................................................. 100

6 State-based Approach Extensions ........................................ 101
   6.1 Updates to Multiple Objects ................................. 101
   6.2 Accountable Access Control ................................. 104
   6.3 Accountable Lock Service .................................. 106
   6.4 Denial of Service ............................................. 109
   6.5 Accountable State Machines ................................ 110
   6.6 Summary ........................................................ 111

7 Privacy Protection and Accountability ................................ 113
   7.1 Accountability for Data Dissemination ................. 113
      7.1.1 Problem I: No State Changes ....................... 114
      7.1.2 Problem II: No Trusted Path ....................... 114
      7.1.3 Detect Now or After the Fact? ..................... 115
   7.2 Data Leaks and Access Control Misconfigurations .......... 116
   7.3 Overview ...................................................... 117
      7.3.1 Identifying Sensitive Files ....................... 118
      7.3.2 Sensitivity Tracking and Breach Detection ........ 118
      7.3.3 Disclosure Policies ................................ 120
   7.4 Limitations ................................................... 121
   7.5 Design ........................................................ 123
7.5.1 Reducing Doppelganger Overhead ............................................ 123
7.5.2 Doppelganger Inputs and Outputs ............................................ 124
7.5.3 Example: Secure Copy (scp) .................................................... 131
7.5.4 Future Work ......................................................................... 132
7.6 Implementation ....................................................................... 133
7.6.1 File Systems ......................................................................... 133
7.6.2 Data Structures ..................................................................... 133
7.6.3 System Calls ......................................................................... 134
7.6.4 Process Swapping .................................................................. 134
7.6.5 Future Implementation Work .................................................. 135
7.7 Evaluation .............................................................................. 135
7.7.1 Application Micro-benchmarks ............................................... 136
7.7.2 Web Server Performance ...................................................... 139
7.8 Related Work ........................................................................... 140
7.9 Summary and Conclusions ....................................................... 142
8 Application-specific Accountability .............................................. 144
8.1 Preliminaries ............................................................................ 144
8.1.1 Virtual Resource Economies .................................................. 144
8.1.2 Identity ................................................................................ 145
8.1.3 Brokers ............................................................................... 147
8.1.4 Broker Hierarchies ............................................................... 149
8.1.5 ORCA .................................................................................. 150
8.2 Accountability in a Virtual Resource Economy ......................... 151
8.3 Accountable Resource Delegation ............................................. 154
8.3.1 The SHARP Resource Delegation Model .............................. 154
8.3.2 Divisible Resources ............................................................. 155
xi
8.3.3 Preserving Location Information ........................................ 157
8.3.4 Extended SHARP Resource Delegation Protocol .................... 158
8.3.5 Assignment Tree and Assignment Forest .............................. 164
8.3.6 Implementation ..................................................... 165

8.4 Virtual Currency ........................................................ 166
  8.4.1 Self-recharging Virtual Currency ................................. 168
  8.4.2 Accountable Virtual Currency Design ............................ 169
  8.4.3 Auditing ........................................................... 171
  8.4.4 Implementation .................................................... 172

8.5 Application-specific Accountability .................................... 173

8.6 Summary ............................................................... 174

9 Final Thoughts ......................................................................... 175
  9.1 Contributions ................................................................ 175
  9.2 Future Directions ......................................................... 178

Bibliography ........................................................................... 181

Biography ................................................................................ 189
# List of Tables

3.1 Summary of components and trust assumptions. ........................................... 34

5.1 Summary of attacks and defenses. ............................................................ 89

5.2 Accountable storage service operations. The service accepts reads and writes to a set of named versioned objects. All client writes and all server responses are digitally signed so that actors can be held accountable for their actions. Challenges and audits provide the means for clients or third-party auditors to verify the consistency of a server's actions relative to periodic non-repudiable digests generated by that server and visible to all actors. ........................................................ 90

7.1 Doppelganger-kernel interactions. ......................................................... 123

7.2 Average total number of additional pages created during a run of the data transfer micro-benchmarks. Each run transfers 600 files for a total of 133MB. .......... 138
List of Figures

3.1 Signed action histories are at the heart of accountability. An action history consists of digitally signed action records. The signature on each record helps identify the requestor/executor of the action and protects the integrity of the record’s contents. .................................................................................................................. 39

3.2 Each actor commits to its execution history by periodically distributing to other actors a view of its history. To preserve privacy and for improved efficiency, actors perform this step by computing state digests: compact representations of their action histories. Consecutive digests are chained, to prevent against addition/removal of state digests. .................................................................................................................. 40

3.3 Services can make provable statements about the contents of their action records. In particular, it is possible to construct proofs to demonstrate that an action record is a member of the action history set at a particular time instance. Similarly, services can construct proofs showing that an action is not a member of the set. Proofs of inclusion and exclusion can be verified independently against the published state digests of the service. .................................................................................................................. 41

3.4 Consistency protocol for accountable services. Each state variable is annotated with a version number. Update requests specify the expected version number for each state variable in the action write set $W$. The service honors requests with matching version numbers, and must reject requests with stale version numbers. Rejections optionally supply the new values of all affected variables. The client resolves the conflict and issues an updated request (if necessary). .................................................................................................................. 43

3.5 At time $t_n$, the malicious operator reverts the effect of the update performed at time $t_m$ and sets the corresponding value to the one valid at time $t_k$. An audit that tracks the value through a sequence of snapshots detects this violation, as long as a read is performed in the interval $[t_m,t_n)$. .................................................................................................................. 45

3.6 The longer a reverted update remains undetected (decreasing $p$), the larger the number of snapshots that must be inspected to ensure that an audit detects misbehavior with a given fixed probability. .................................................................................................................. 47

4.1 The leaves of a Merkle tree organize the elements of a given set. Each internal tree node contains an authentication label computed over its subtree. The tree can construct a proof to demonstrate the existence of a given element. For example, the proof for element 25 includes the labels stored at the shaded nodes. Proofs of non-existence, however, are equal to the whole tree. .................................................................................................................. 58

4.2 An authenticated search tree [24] maintains an ordered authenticated index of a set of data. The tree organization allows to construct efficient existence and non-existence proofs. .................................................................................................................. 59
4.3 Data layout affects authentication label computation. Depending on data layout, label computation takes a different number of arguments: leaf-oriented designs use only the child labels and the internal key, while node-oriented design require, in addition, the hash of the data stored at each internal tree node. .......................... 60

4.4 Red-black tree average proof size depending on data layout. The leaf-oriented design results in smaller proofs. ......................................................... 61

4.5 Computing an authentication label. Each computation takes as input one child label component, and \((k - 1)\) label and split key components. ................................. 62

4.6 Mapping a binary search tree to disk blocks for efficient storage. The idea is to use a large degree external memory tree, such as a btree, and to organize the keys within each block into a binary search tree. This organization achieves good I/O performance and produces membership proofs of reasonable size. .................. 64

4.7 Expected stretch factor. The stretch factor of a persistent dictionary is logarithmic and depends on the epoch length. The smaller the epoch, the larger the stretch factor. Larger stretch factor corresponds to a larger data structure. ......................... 66

4.8 The CATS toolkit consists of several layers. At the bottom, an I/O abstraction layer provides access to the underlying storage medium. A block cache resides above the I/O layer and provides caching and consistent access to disk blocks. The dictionary module uses the block cache to store and organize its contents. The write-ahead logger helps record state about each dictionary update to enable recovery after a crash. 71

4.9 Cost of hashing using MD5 and SHA-1. For small data sizes both hash functions offer similar performance. As the size of hashed data increases, SHA-1 is noticeably more expensive. ......................................................... 75

4.10 Membership proof size as a function of the number of proof components (SHA-1, data at leaves). Single proof components have constant size. The number of proof components is logarithmic relative to the number of elements of the data set. For most practical purposes, proofs are less than 2KB long. ......................... 76

4.11 Membership proof verification time as a function of the number of proof components. For most practical purposes, proof verification takes less than 1ms. Note that the times on the graph do not include the time spent to obtain/verify snapshot authenticators and validate signing certificates. ................................. 76

4.12 Average path length for a main memory search tree. ......................... 77

4.13 Membership proof size for a main memory persistent authenticated search tree. As expected, proofs are less than 2K bytes large. ......................... 77

4.14 Membership proof verification time for a main memory persistent authenticated search tree. As expected, verification takes less than 1ms. ......................... 77
4.15 Main memory persistent tree stretch factor. As epoch size decreases the stretch factor increases, making the resulting data structure larger. .................................................. 78

4.16 Main memory tree insertion time without computing authentication labels. Smaller epochs take longer to perform a tree update since a smaller epoch requires making copies of a larger number of nodes during each update operation. .................................................. 78

4.17 Main memory insertion time with immediate authentication label computation. The cost of hashing is independent on epoch length and is the dominant cost of an update operation. .................................................. 79

4.18 Main memory insertion time using finalization. Delaying the computation of authentication labels until the end of an epoch can significantly reduce the cost of authentication for large epochs. As epoch length decreases, the benefit of finalization decreases. .................................................. 80

4.19 External memory tree average path length. .................................................. 81

4.20 External memory membership proof size is logarithmic relative to the number of unique keys in the index and is on the order of 1KB. The time to verify a membership proof is on the order of 500 \( \mu \text{seconds} \). .................................................. 81

4.21 The external memory tree’s stretch factor is logarithmic relative to the epoch length. Smaller epochs have significantly higher stretch factors resulting in larger indexes. 82

4.22 External memory tree update time without authentication depends on the epoch size. 82

4.23 External memory tree updates with immediate label computation. Label computation adds between 400 and 450 \( \mu \text{sec} \) overhead to update operations. .................................................. 83

4.24 Delaying label computation decreases the amortized overhead of authentication. The magnitude of the decrease depends on the epoch size. .................................................. 83

4.25 The time to apply an update to the authenticated dictionary is logarithmic relative to the number of unique keys in the index and increases as epoch length decreases. 84

4.26 Cache misses during delayed label computation can affect performance. 84

5.1 Overview of a Cats-based service: the accountable storage service. .................................................. 88

5.2 Storage service implementation. The storage service consists of an index and an append-only log. The service is implemented as a collection of stages. Each stage is associated with a pool of worker threads. Pools can grow and shrink depending on load. .................................................. 92
5.3 Maximum achievable data bandwidth for storage service RPC interface based on SOAP/DIME and WSE 2.0. The results suggest that this interface has high overhead and services using it will be potentially communication-bound. .......................... 95

5.4 Write saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. As object size increases, the relative cost of digital signatures decreases. ............................. 96

5.5 Read saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. Performance is seek limited due to the log-structured design. ........................................ 96

5.6 Rate of epoch creation as a function of request rate and workload contention. Higher request rates and contention create new epochs faster and can affect service longevity. 97

5.7 Epoch length as a function of request rate and workload contention. Epoch length is independent on request rate and is determined by workload contention. ............................. 97

5.8 Auditing saturation throughput for different depths, spans, and scopes. Smaller spans show better performance due to increased locality. The impact of scope is less pronounced. ................................. 99

6.1 Probability of not detecting a violation in a multi-object update of age 10 snapshots spanning 10 objects. ................................. 103

6.2 Some state variables can be defined as state machines with well-known states and transition functions. Action functions involving such variables simply select the correct transition and produce the new value. Such action functions have well-defined verification functions, which can be generalized and automated. ................................. 111

7.1 Using doppelgangers to avoid a breach. ................................. 119

7.2 Signaling that leads to divergence. ................................. 128

7.3 Apache relative transfer time. ................................. 137

7.4 NFS relative transfer time. ................................. 137

7.5 SSH relative transfer time. ................................. 138

7.6 SpecWeb99 throughput. ................................. 140

7.7 SpecWeb99 response time. ................................. 140

8.1 An example virtual resource leasing economy consisting only of a provider and a consumer. The provider and the consumer interact directly and require mutual trust relationships to authenticate each other. ................................. 147
8.2 An example virtual resource economy consisting of a provider, a consumer, and a broker. Dashed lines indicated pre-existing trust relationships, while the solid line indicates a transitive trust relationship. The consumer first interacts with the broker to obtain a ticket (a promise for resources). The consumer uses the ticket to redeem it with the provider for the actual lease. The ticket contains sufficient information to enable the provider and the consumer to interact without requiring a pre-existing trust relationship between them. .......................................................... 148

8.3 An example virtual resource economy consisting of a hierarchy of brokers. The broker hierarchy supports complex relationships between brokers to emulate closer real-world economies. Importantly, the hierarchy enables transitive trust relationships, thus reducing the complexity of the system and the cost to entry. ......................... 150

8.4 A SHARP ticket consists of one or more delegation records. Each delegation transfers control over a given number of units for a specified period of time. Each delegation is signed by its issuer. A ticket contains the full path of delegations from a site authority down to a service manager. Note that each ticket starts with a self-signed delegation used to indicate the total amount of available resources at the site. ......................... 155

8.5 An allocation that causes a problem when using SHARP for divisible resource units. (broker view) .......................................................... 156

8.6 An allocation that causes a problem when using SHARP for divisible resource units. (site view) .......................................................... 156

8.7 Using binpools and split and extract operations for accountable resource delegation. 158

8.8 A resource delegation: 3 units with a resource vector (5,4), and term (30, 60). .... 159

8.9 A complete resource delegation: 3 units with a resource vector (5,4), and term (30, 60). Note that the delegation also includes binpools 1 and 2 from which the delegated binpool 3 has been derived. .......................................................... 160

8.10 To create a SHARP certificate for a given actor, the issuer of the SHARP certificate signs the actor's X.509 certificate and produces a new SHARP certificate, which consists of its own SHARP certificate, the actor's X.509 certificate and the issuer's signature on the actor's X.509 certificate. .......................................................... 161

8.11 A resource ticket consisting of two delegations (SHARP certificate removed). .... 163

8.12 An assignment tree consists of all resource delegations deriving from resources from a given type and site. The tree can be used to detect and prove overcommitment by a broker. .......................................................... 165
8.13 Auditing cycle. A trusted bank service issues initial credit budgets to consumers. Consumers self-recharge their credits and use them to obtain resource tickets through a network of brokers; credit transfers do not involve the bank. Spent credit notes eventually propagate to the originating resource sites, who redeem them to the bank as proof of value offered to the community. The bank or its delegates audit the credit notes to hold participants accountable for their transfers.  .................................................. 171
Chapter 1

Introduction

Computer software today is ubiquitous. Communication, electronic commerce, entertainment, transportation, health care, and numerous other activities are powered by millions of interconnected components. Mission-critical processes in finance, governance, and defense are taken over by dedicated software. Traditional pen-and-paper institutions are rapidly transformed into virtual organizations spanning multiple continents and managing complex workflows in which humans and software interact to deliver faster, better, and more efficient results.

While software is well on its way to take over an increasing number of activities, it is also growing as an attractive target for subversion and abuse. Software’s increasing power to affect the real world is not only enabling rapid advances and improvements in our lives but it is also becoming attractive for individuals and attackers, who may wish to subvert its powers and use them for their own benefit. In a world controlled by software it is critical to ensure that software components are not abused by their users or subverted by attackers; malicious actions, either by software, or by its users must be detectable and provable.

This thesis focuses on addressing the challenges of building trustworthy software systems—systems for which one can verify that both the system and its users faithfully perform their assigned roles and functions. Our primary intent is to empower services to perform important mission-critical tasks, while providing high level of assurance that every operation they perform complies with the rules governing each system and its use. The ultimate goal is to help prevent abuse of existing and emerging technology by making software and its users accountable for their actions.

A system is accountable if it provides to others the means to verify for themselves that the system functions and behaves correctly. When misbehavior is detected, the system produces evidence to identify and punish the responsible party. The stronger the evidence of misbehavior, the stronger the accountability properties of the system. Accountability requires an ability to identify and punish access control violations [69], but also extends beyond it, as even a properly authorized user can request and perform illegal actions.
Accountability is a desirable property of many daily activities. For example, governments must be accountable for their use of power, individuals must be accountable for their behavior in society, and corporations must be accountable to their shareholders. When challenged to do so, ministers, individuals, and executives must be able to demonstrate the correctness of their actions. The checks and balances used to enforce accountability in these contexts, and ultimately the threat of detection and punishment, provide strong disincentives for abuse. They also offer a level of assurance that enables each organization to perform its duties.

As many of the operations people perform on a daily basis are either taken over by software, or require the interaction with software of various types, it is becoming increasingly more urgent to extend the existing accountability techniques to the digital world. This is both an opportunity to improve on and streamline the existing approaches, and also a major requirement to integrate novel techniques intended to address the new means of abuse that modern software can offer.

1.1 Problem

We use the general term service to refer to any software component that offers functionality to its clients. Services are usually networked and interact with each other. The clients of a service can be humans or other services. We use the collective term actor to refer to both humans and services. An action is then defined as either a request from one actor to another or the processing of a request received from an actor. A single action may be local to an actor, but it may also trigger other actions potentially in multiple other actors.

A principal is a real-world certified and verifiable identity. For example, the social security number (SSN) is an identity issued by the U.S. government, that can be verified by the possession of an identity card. Actors and actions may be associated with a principal. Actions with no discernible principal are considered to be anonymous.

The steps one must perform to complete an individual action are defined by its specification. The specification of an action defines the preconditions and the result of the action. For example, the action average has a precondition that there should be at least one number, and the result is defined as the sum of all numbers divided by their count. One way to think of specification is as a collection of rules and laws governing the behavior of each actor.
1.1.1 Law-governed Interactions

In the course of its lifetime an actor may fail to conform to the rules governing its actions. For example, a storage service may return invalid data, unauthorized modifications in an enterprise system may take place, sensitive information may be destroyed without leaving a trace, etc. These example problems are characterized by the fact that they cannot result from any application of valid rules that govern the operation of each system. We refer to violations of this type as semantic faults. When a semantic fault actually takes place, we refer to it as a semantic failure. Semantic faults represent an important class of problems that result from the use of software services of various forms and can be caused not only by the software itself but also by its end users.

We can distinguish two general categories of semantic failures: fail-stop and Byzantine. Fail-stop failures often result in temporary termination of service, and can thus be easily detected. For example, denial of service, is a common fail-stop failure, which is easily identified by the lack of response/service. On the other hand, Byzantine failures [68] represent a class of failures with inconsistent often deliberate/malicious behavior. A Byzantine failure may have subtle and not immediately noticeable effects, and is much harder to detect, e.g., detection may require significant amounts of information and/or processing, e.g., replication of work among multiple participants, and, in some cases, is impossible, e.g., when a majority of the participants collude.

On an abstract level, we can consider Byzantine failures as attempts to conceal the fact that a semantic failure has taken place: the actor responsible for the failure commits it in an attempt to achieve some form of benefit, while acting within the confines of the system. Such actions deliberately try to avoid detection, since once detected the culprit is exposed and the principal associated with it can be punished. The focus of this thesis is semantic failures that result from Byzantine, non-fail-stop behavior. Our goal is to design efficient solutions to help identify and ultimately deter intentional Byzantine semantic failures.

1.1.2 Means and Motives for Abuse

Software systems often offer both the means and the motives for abuse. Increasingly software controls real-world processes and assets. Commands issued by software have direct impact on the real world and the people in it. For example, it may take just a few mouse clicks to reroute traffic,
redirect electricity, or turn off the water supply. Before computer systems took over these tasks, each one would take significant amount of time, and, most importantly, would involve a number of individuals. The involvement of multiple people makes it harder to abuse the system on a massive scale. As a result, by automating workflow and reducing the need for direct face-to-face interaction software systems are threatening existing practices designed to prevent abuse. As software systems become more ubiquitous, they are likely to take this process further and offer even more powerful features within the reach of its users.

With software in control of real world processes and functions, violations and abuse within a software service are likely to bring some tangible benefit to a malicious user or a skillful attacker. Controlling a software service increasingly results in some noticeable effect on the real world and as such may be used to change the course of important processes and activities. If perpetrators manage to disguise their actions and avoid detection, they can continue to enjoy the benefits of subverting a system for indefinite period of time. In essence, electronic services offer new, often very powerful means to achieve personal benefit at the expense of somebody else. It is very difficult to depend on and trust services that do not address this type of problem.

In addition, software systems are owned, operated, and used by self-interested individuals and organizations. Self-interest is a strong incentive for individuals to use their environment and skills to their best advantage. Assuming rationality and no altruism, a self-interested actor has strong incentives to violate its official semantics for personal benefit, as long as these violations can remain undetected—misbehavior in this case is simply the rational thing to do. The ability to identify, prevent, and discourage semantic violations is, therefore, critical for any system with self-interested actors and computer systems in particular, given their powerful concentration of means and motives for abuse.

1.1.3 Interdependence and Scale

Today's software systems consist of a large number of components (services), which are increasingly interconnected. To provide better service and/or expand existing abilities, previously independent systems are integrated and extended with new functionality. Offering these additional features intrinsically depends on data shared among different services, often residing in separate administrative domains. Individual administrative domains are generally autonomous, have their own practices,
and are controlled by self-interested people and organizations. Systems that result from massive integration are inherently decentralized and are based on fragile trust relationships: each component of the system must trust (to some extent) the components it interacts with. The dependencies between individual components, the potential clash of priorities, the relative autonomy of individual participants, and the high stakes (Section 1.1.2), make trusting the resulting systems much more problematic.

One of the major problems of massive integration and large-scale systems is increased vulnerability to subversion and abuse. Semantic failures are likely to be more frequent in a system consisting of a large number of loosely interconnected components that interact across trust domains. The usual lack of centralized control and the autonomy of administrative domains make it hard to enforce consistent rules and practices. As a result, semantic faults are more likely to occur and remain undetected for a longer period of time. Even if a single administrative domain takes extreme measures to prevent semantic faults originating in actors within its own control, semantic faults in actors residing outside of its immediate control can render many of its precautions useless; a system of interdependent components is only as secure as its weakest link.

Another important issue in a massively interconnected system is the fact that interdependence amplifies the effect of semantic failures. In a system with component dependencies the output of one actor is the input of another: semantically correct data exchanges and actions across every actor boundary are necessary to ensure that the semantics of the overall system is not compromised. If a single actor misbehaves, its actions are likely to affect the actions of its neighboring actors: they may commit invalid actions, fail to act, or produce invalid data and send them downstream to other actors. Left uncontrolled this process can spread rapidly over the system and affect every component and user.

1.2 Solution

The threat of abuse is older than computing and it has been the focus of the law enforcement and judicial systems. These institutions impose strong disincentives for abuse and misbehavior. Prosecutors collect evidence demonstrating misbehavior. Verifiable evidence helps convict perpetrators and acquit innocent suspects. Convictions carry a penalty designed to discourage future crimes.
The threat of punishment may not prevent all instances of misbehavior, but it is effective enough to prevent the majority and provide a guarantee of the integrity of day-to-day interactions and activities. Enforcing similar guarantees is essential for the stability, dependability, and success of software systems; trustworthy systems must be able to prevent and tolerate misbehavior and offer assurance that participants behave correctly and consistently. Methodologies and techniques are thus necessary to extend the guarantees of face-to-face interactions to the faceless, anonymous, and automated digital world.

System builders have powerful tools to build secure systems on today’s insecure networks. These tools combine secure sockets and public key cryptography to offer secure perimeters by means of secure communication via authentication, encryption, and message integrity [69]. However, no common perimeter exists across multiple trust domains, and more importantly, a perimeter offers no protection once it is breached, or the attack comes from within it. Byzantine Fault Tolerance [29, 108] uses state replication to tolerate misbehavior, but it comes at a cost and offers no protection when replicas collude or are within the control of a single entity. As completely preventing misbehavior is an elusive goal, it is essential that when misbehavior takes place it be possible to identify the chain of events that led to it and pinpoint the responsible principals. Strong evidence for responsibility is crucial to administer punishment and prevent future abuse.

This particular class of problems requires new solutions, specifically designed to deal with the problems arising from complexity, lack of centralized control, and dependence on fragile trust relationships. This thesis argues for an approach to building dependable trustworthy systems through system support for accountability. A system is accountable if it provides mechanisms to detect and expose misbehavior by its participants. Actions in an accountable system have a known requestor and executor and the correctness of an action can be verified. Most importantly, an accountable system can construct evidence about a principal’s responsibility. Strongly accountable systems go a step further in their ability to construct evidence that does not depend on voting or consensus and can be independently verified by a third party. A strongly accountable system offers high level of assurance that all of its actors operate and behave as expected.
1.2.1 Design Principle of Dependable Network Services

This dissertation elevates accountability to a first-class design principle of dependable network services. The intent is to integrate accountability techniques in the design, internal organization, and protocols of a network service. While it may be possible to apply some of our techniques to existing systems, e.g., by interposing on their communication channels, our primary goal is to study and understand how integrating accountability in the initial design affects the ability of a software component to offer trustworthy service and to ensure that both the component and its users comply with their roles and responsibilities.

Accountability is a means to meeting classical dependability objectives (fault prevention, fault tolerance, and fault removal). It is essential for constructing trustworthy systems and is a building block for dependable systems that offer strong assurance of compliance with specification and tolerance to disruption by malicious or faulty participants [132]:

- Accountability is necessary to support contracts and enforcement in federated distributed services that involve exchanges of goods or services among individuals, organizations, or trust domains. In the context of non-electronic goods accountability is essential for ensuring that such relationships conform to the agreed upon rules. Accountability for electronic services extends this level of assurance to the digital world.

- Designing for accountability can help to facilitate development of services that are resistant to tampering and attack. This is because the internal protocols and organization of accountable services are designed to maintain additional information to protect against tampering and subversion. While an attacker may be able to modify physically messages and data sent or maintained by the service, our approach makes it possible to detect the modifications once data are about to be used. The threat of detection increases the cost of misbehavior and offers additional incentives to conform to the established rules.

- When components are compromised, accountable design can limit the damage. For example, accountability mechanisms may guard against undetected corruption of a service state, e.g., a database, and prevent a compromised service from misrepresenting the actions of its clients. This is important when flawed software systems may run on untrusted infrastructure in a
dangerous world inhabited by attackers and network pathogens. In essence, accountable design constitutes a defense-in-depth to supplement weak perimeter defenses, and to offer some protection once the integrity of the perimeter is violated [115].

1.2.2 Focusing Questions

How to integrate accountability in service design: what is the role of actions and internal state?

Designing for accountability requires augmenting traditional service design to support the collection and verification of evidence of correct operation. Since accountability is tightly coupled with the semantics of a service, accountability techniques are likely to be tailored to the specific functions a service performs. Providing common building blocks to simplify this task is essential if we are to simplify and promote the integration of accountability practices into new systems. In this thesis we identify a common set of operations and provide a number of building blocks to simplify and enable integrating accountability techniques into the design of network services.

Accountability requires an ability to produce evidence about misbehavior. Constructing such evidence is challenging: a service may execute an action and later hide the fact that such an action was ever executed. The internal state a service maintains in the course of its operations is the authoritative repository of the consequences of each action the service performs. Since most service operations depend on the contents of the service’s internal state, the process used by a service to store, organize, and access its state has direct impact on our ability to verify the correctness of individual service operations. The techniques presented in this thesis focus on state management as an essential component in building accountable services.

Evidence of misbehavior is a necessary but not sufficient requirement for accountability: full accountability requires knowledge of the actor (or group of actors) responsible for a specific violation. If a piece of evidence shows both misbehavior and an actor responsible for it, this evidence can be used to administer some form of punishment and ultimately provides strong disincentives for future violations. Being able to associate an action with the actor who requests and/or performs it is thus necessary for accountability. When an action affects the internal state of a service it may
be necessary to maintain additional information to associate the elements from the internal state, which were affected by a given action, with the action and the actor who requested or performed the change. A key component of our approach is to associate undeniably each state change with the action and actor responsible for it.

*Can accountability be enforced in a general way?*

It is tempting to assume that there exist generic approaches to enforcing accountability in any network service. Reusable development platforms and off-the-shelf components for building accountable systems are likely to produce more robust and trustworthy systems, since designers and system builders will be able to apply a common approach and use standard building blocks. However, it is unlikely that one solution can fit all requirements and needs. Sometimes the specific requirements and properties of a service may provide additional resources, which can be leveraged to enforce accountability in more efficient and appropriate ways. While general solutions to the problem of accountability are highly desirable, application-specific solutions have the potential to introduce useful techniques, which may not be generally applicable, but may be suitable for a class of applications with similar characteristics. Based on these arguments, this thesis explores both directions of research: we dedicate significant time to designing and building a generic accountability framework, but we also explore how to leverage application-specific requirements to build accountable services.

*How does service complexity affect accountability? What are the limits of accountability?*

It is reasonable to assume that more complex services will require more sophisticated accountability techniques, which will pose bigger challenges to ensuring correctness and enforcing accountability. Complex actions may require convoluted reasoning about the correctness of each step and may ultimately be unable to verify reliably without being able to re-execute the whole action in a controlled environment. It may also be possible that specific types of actions cannot be made accountable regardless of the investment one is willing to make. It is important to study and understand how action complexity affects accountability. Understanding what aspects of a given
action make its accountability hard may help identify actions with similar properties, which are likely to require similar accountability techniques.

*How expensive is accountability and what are the cost dimensions?*

Accountability requires the addition of new protocols, mechanisms, and data structures alongside the numerous protocols and data structures one can find in today’s systems. Each of these additions can introduce new costs in the operation and maintenance of a service. While it may be possible to mask and amortize some of the costs, others may inadvertently affect the critical path of execution and impact every major service operation. The study of accountability must then classify and quantify the various costs of accountability techniques in an effort to further the knowledge and understanding of accountability and its impact on performance, cost, and reliability. There exists an important relationship between the amount of work dedicated to enforcing accountability and the level of assurance offered by the resulting system. A significant portion of this dissertation is dedicated to building and studying the behavior of the various elements of our accountability framework and quantifying the different aspects of their performance.

### 1.3 Hypothesis and Contributions

This thesis studies and evaluates the following hypothesis: *Integrating accountability into the design of network services is a practical foundation for building trustworthy systems which are resilient to tampering and abuse.* In evaluating our hypothesis, we make the following contributions.

1. Identify the key properties of accountability and the benefits of using accountability as a design principle for dependable software systems. This thesis defines the notion of *strong* accountability, which extends existing uses of the term in requiring that a claim for correctness or misbehavior be independently verifiable by a third party. Strong accountability is absolute, and does not depend on quorum and consensus.

2. Introduce a set of general-purpose design principles for building accountable services. Our *state-based approach* applies to any type of service whose actions depend on its internal state.
The individual principles form a comprehensive methodology that can be applied to a range of services to offer varying levels of assurance about each service's execution.

3. Implement the major elements of the framework as reusable substrate to assist the construction of a range of accountable services. Each component is engineered to maximize reuse and encapsulate complex operations behind intuitive and simple interfaces.

4. Evaluate the costs and versatility of the methodology and the performance of the framework. This thesis reports on each major performance aspect both by using the building blocks in isolation and also from evaluating the prototype of an accountable storage service built using the state-based approach and framework.

5. Show how to leverage and extend the provided building blocks to integrate accountability into other critical infrastructure services, e.g., authorization and lock services. We study analytically the cost of accountability in these contexts and reason about the impact of service complexity on accountability.

6. Study the problem of information leaks in a commodity operating system in relation to the problem of accountability. We design, implement, and evaluate a novel operating system primitive a \textit{doppelgänger process} to help track information flow within the operating system. Our prototype can successfully detect leaks of sensitive information and protect a desktop user from becoming accountable for the actions of software acting on her behalf.

7. Show how to use application-aware techniques to design accountability for an example system—a lease-based virtual resource economy. Unlike the general-purpose state-based approach to accountability, the resource leasing system illustrates how to leverage knowledge of the internal details of a system to integrate various accountability techniques. In particular, we show how to offer non-repudiable interactions, provide fine-grain resource delegation, and use and operate with accountable currency.

\subsection{1.4 Thesis Overview}

The rest of this thesis is organized as follows:
Chapter 2 defines the notions of accountability and strong accountability, describes the basic properties of and requirements for accountable systems, and relates accountability to other approaches for building trustworthy systems.

Chapter 3 describes the internals of the state-based approach to building accountable network services and Chapter 4 presents the design and experimentation results of CATS, a reusable toolkit for building accountable services based on the state-based approach.

We describe our experience using the toolkit to construct an accountable storage service in Chapter 5. Chapter 6 presents extensions to the basic toolkit and how they apply to the design of other important network services.

In Chapter 7 we change our focus to the problem of privacy as it relates to accountability. We present TightLip, an extension to a stock Linux kernel designed to prevent leaks of sensitive data and thus to protect desktop users from applications acting on their behalf, and for whose actions they may become accountable.

Chapter 8 studies the problem of accountability in the context of a virtual resource leasing economy. We identify the accountability requirements for a distributed resource leasing system and present the design and implementation of several building blocks that make key operations within a resource economy accountable. These chapters demonstrates how to leverage the specific requirements of a system and its protocols to design and integrate accountability solutions targeted directly to the system’s needs.

Finally, Chapter 9 concludes and outlines some areas of future research.
Chapter 2

Accountability: Means to an End in Dependable Software Systems

The approach to building trustworthy systems proposed in this dissertation elevates accountability into a first-class design principle—accountability must be integrated in the core protocols and algorithms of dependable system. Our goal is to integrate programmatic checks and balances in the design and implementation of networked systems to enable the verification of systems’ behavior and operation. In this section we further define the notion of accountability, its properties, and requirements. We also discuss how accountability relates to existing and prior solutions to building dependable systems.

2.1 Accountable Actions

We first recall several important terms we introduced in Section 1.1. For simplicity, we focus our discussion to individual services and their clients, rather than complete systems. Since a complete system consists of one or more services, the discussion in this section can easily be applied to reason about the behavior of whole systems. We shall also use the terms service, application, software, and component interchangeably.

For our purposes a service is an instance of a program that executes requests from clients according to a pre-defined protocol carried over an interface. While the service clients can reside in the same process as the service (and communicate using direct function invocation), we are primarily concerned with the case of networked services where the service and its clients reside on different machines and communicate over a network communication channel. For example, the service of interest might expose its functionality over RPC [118], such as a network file service [119], or over SOAP [123], as in the case of XML Web services.

A service encapsulates a set of computational resources such as CPU, storage, network bandwidth, etc., and may span one or more physical servers. Each of those resources may maintain information required by the service to perform its activities and may also perform computations.
We refer to the collection of all information used by the service to perform its actions (and produced as a result of these actions) as the service’s internal state. 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. Since any component may act as a client or a server with respect to other components, we sometimes use the more general term actor.

Services and their clients act on behalf of people and organizations (principals, Section 1.1) who may use the software to enter into and fulfill contractual and legal responsibilities in their various roles. An action is a request or a response that fulfills a request or indicates acceptance of a request. The actions of services and their clients are driven by semantic rules. Semantic rules can be used to determine if an action is correct or not. Each action may have serious consequences in the real world, for which both the requestor and the executor may be held accountable.

**Definition 2.1.1. (Accountable service)** A service is accountable if it provides the means to its clients or external observers to verify for themselves if the service and its clients behave correctly.

Accountability denotes assurance of semantic behavior that extends beyond basic perimeter security and the mechanisms for message authentication, encryption, and integrity (Section 2.3). 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: the sender retains a signed receipt to prove the postal service accepted the package, while the receiver signs a receipt to prove the postal service delivered the package.

An accountable service can provide, upon request, information that can be used as evidence to demonstrate that the service (or its clients) behave (or do not behave) according to the established semantic rules. Depending on the quality of the supplied information, it may serve as a basis for administering some form of sanction when deviation from normal behavior is detected.

The goal of building accountable services is to detect inconsistent or incorrect states or actions. An accountable system could validate an actor’s state as a function of the actions applied to it, enabling audits to identify the original source of a fault. The basic premises of accountability required to enable successful audits are:
• **Actions have well-defined semantics.** Fundamentally, accountability rests on the notion of correctness and the ability to define precisely what constitutes a correct action within a given context. The notion of correctness defines the “rules of the game” and spells out what actors can and cannot do: without it, accountability is undefined and one cannot make any claims about the behavior of actors within a computer system. Well-defined actions enable a service’s clients or peers (such as an auditor) to reason about its states and actions. Unlike actions and situations in real life, electronic actions are likely to have much stricter definitions of correctness, which will not be subject to multiple, potentially conflicting interpretations. In this dissertation we link correctness to the notion of semantics: the meaning of the set of rules that define the behavior of an actor. Semantics encompasses the interpretation and meaning of the *specification* of an actor’s actions: its algorithms, published documentation, or publicly made assertions about its behavior. Section 3.1.1 presents a formal definition of semantic correctness.

• **Every action is bound to the identity of its actor.** In an accountable system it must be possible to link an action to the actor responsible for it. The association of actions and actors is essential for accountability since it makes it possible to assign responsibility, and potentially award or punish actors for their actions. It is critical that the association among actors and actions cannot be forged or misrepresented (Section 2.1.3).

In general, there are at least two parties responsible for a given action: the actor, who directly or indirectly requests the action (*requestor*), and the actor who actually executes the request (*executor*). Each of the involved actors bears a responsibility to perform its part of the process according to the service semantics: requestors should only issue valid requests, while executors should execute only valid requests and should deny any request they find not to conform with the established rules. An ideal accountable system should be able to distinguish among the actions of the requestors and those of the executors. In practice, however, this task may be very complex, e.g., when the requestor is a human and the executor is a piece of software: drawing the distinction between the actions of the human and the actions of the software is not trivial (Chapter 7).

The primary concerns of accountability are actions that directly affect the state of other actors.
Accountable actors can certify the correctness of their actions and cannot deny that they sent a given message or that they committed changes on behalf of other actors. However, accountability alone cannot prevent an actor from sending invalid data to an actor that does not choose to challenge the message’s correctness. In addition, accountability for violations of privacy is a different and complementary problem: any actor can send to any other actor copies of its state or received messages. We address the privacy aspect of accountability in Section 7.

Enforcing accountability, typically, involves performing operations after one or more actions of interest have taken place. Using after-the-fact checks a client, peer, or an auditor attempts to determine if observed behavior conforms to existing rules. Thus, a window of vulnerability exists that spans the time from the moment a misbehavior takes place, until the time it is detected.

The size of the window depends on a number of factors: performance, service organization, willingness to challenge the correctness of actions and data, etc. As a result, the process of using accountability to ensure semantic compliance is primarily applicable to systems which can tolerate some temporary inconsistency and misbehavior.

Accountability alone cannot prevent misbehavior. In particular, our state-based approach (Chapter 3) does not attempt to remove the need for trust among clients and servers; trust is essential for sharing and cooperation. Rather, the philosophy of our approach is: “trust but verify” [133]. Our goal is to hold the service and its clients accountable for their actions, so that a faulty actor is exposed and isolated. Accountability precludes an effective attack to subvert the behavior of the overall system without unavoidable risk of detection. The threat of detection can ultimately discourage misbehavior, but is not sufficient to prevent it.

2.1.1 Basic Properties

Every accountable system shares a set of basic properties, which enable reasoning about the correctness of the system. Namely, an accountable system can ensure that the actions and state of each actor are:

- **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 an 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 specification. Alternatively, majority voting can be used to certify the validity of collected evidence [29].

• **Tamper-evident.** Any attempt to corrupt internal state incurs a high probability of detection. In particular, a client, or an external auditor, may determine if the internal state could or could not result from the sequence of operations issued on the service.

These three properties make it possible to validate the integrity of actions and internal state and assign responsibility when the observed behavior does not comply with a specification. Correctly functioning actors in an accountable system can provide evidence of their integrity, while dishonest or compromised ones cannot hide their misbehavior.

Importantly, an accountable system must take special precautions to prevent false accusations: if the risk of being falsely accused outweighs the benefits derived from being part of the system, then actors will be unwilling to participate. Section 2.1.3 discusses in more details the problems of false accusations and hiding misbehavior.

### 2.1.2 Means to System Dependability

Given the basic properties of accountability, integrating accountability into service design becomes a means to build dependable and trustworthy systems. In fact, accountability can be used to provide three of the four means to system dependability [70]: it can help tolerate, remove, and prevent semantic faults.

• **Fault Tolerance.** Accountable services offer a means to detect and isolate semantic faults, preventing them from spreading to other services and actors. While it may not be possible to mask a fault entirely, incorrect execution can be suppressed so that an attempt to violate the service semantics is not successful. Accountability-targeted service design might also allow recovery by rolling back the service state to a previous, known to be valid state, once a fault is detected.
• **Fault Removal.** By identifying the root source of a fault, accountable systems can exclude misbehaving actors and limit their impact on the rest of the system.

• **Fault Prevention.** The ability to tolerate, exclude, and punish misbehaving actors can ultimately serve as a strong deterrent to any attempts to misbehave. Accountability exposes an attacker to the risk of legal sanction or other damage to its interests. Effective accountability increases the cost of malicious behavior, so that the expected benefit of misbehavior is zero or negative. From a game-theoretic perspective, the dominant actor strategy in a carefully designed accountable system is to comply with the standards of behavior established by the system designers.

2.1.3 Strong Accountability

Accountability requires the means to provide evidence of correct and incorrect actions. The stability of an accountable system depends on its ability to supply evidence that cannot be forged and manipulated: an actor should not be able to hide misbehavior, nor it should be able to fabricate successfully a false accusation. In general, the evidence collection and verification process must:

• **Detect misbehaving actors eventually.** Collected evidence must span all system activities, so that no misbehaving actor remains undetected indefinitely.

• **Prove an actor’s wrongdoings irrefutably.** A piece of evidence is reliable and useful only if no actor can dispute its validity and implications. It must be impossible to acquit guilty actors or fabricate evidence to incriminate well-behaving ones.

There are multiple ways to meet these requirements. *Majority voting* is one way to ensure the quality of collected evidence [29, 59]. As long as a majority of a system’s actors remain uncompromised and follow the agreed-upon rules, this process is guaranteed to result in a stable system. However, as time progresses, systems and majorities within them can change: a statement, which may have been true in the past, may become false, depending on the distribution of votes. Inherently, a majority-based system is vulnerable to “groupthink” (Section 2.3) and, while it may work most of the time, it may not always provide the high level of assurance accountability can offer.
We now refine the notion of accountability to take into consideration the process of evidence collection and verification:

**Definition 2.1.2. (Strongly accountable service)** A service is strongly accountable if it provides the means to its clients or external observers to verify for themselves if the service and its clients behave correctly, without trusting assertions of misbehavior by another participant who may itself be compromised or malicious.

Strong accountability implies a level of assurance that goes beyond the one offered by majority-based systems. Strongly accountable systems are able to provide self-contained evidence, which provides sufficient information so that a client, a peer, or an auditor, can verify it independently, without making any assumptions about the correctness and incentives of the accuser. A strongly accountable system may still not expose all misbehavior, e.g., if peers choose not to question each other’s actions, but, once a peer decides to verify a given action, a strongly accountable system cannot conceal a misbehavior, even if an attacker controls a majority of the system’s components. Strong accountability, however, may not always be attainable.

### 2.1.4 Digital Signatures

Non-repudiation and integrity of communication is a key enabler of accountability. Before a piece of data can be used as evidence of misbehavior, it must be shown, beyond any doubt, that it originated from a given principal and that it has not been altered since in any form. This is an essential requirement for all accountable systems, which can currently be satisfied by using digital signatures based on asymmetric keypairs. Digitally signed communication are a basic prerequisite for strongly accountable systems.

The use of digital signatures introduces certain computation and infrastructure overheads. Generating and verifying digital signatures is a relatively expensive operation, the cost of which is expected to remain constant regardless of improvements in technology; key sizes must grow to compensate for newer and faster hardware. Signature operations depend on the size of the message being signed, the algorithm used, and the key size. Digital signatures also require the existence of an infrastructure to manage public keys and revocation lists. Finally, obtaining certificates and verifying their validity is another cost, somewhat mitigated by caching.
Accountable services add new levels of assurance about their correctness. Increased assurance, however, requires some additional price to be paid. Regardless of their cost, digital signatures are essential to ensure identity and message integrity as these two properties are fundamental for building accountable services. Importantly, symmetric cryptography approaches, e.g., Message Authentication Codes or SSL [15, 48], are not sufficient for accountability. Shared keys cannot guarantee non-repudiation of origin or content: any party in possession of the shared key can forge the actions of another party that uses the same key.

Digital signatures introduce modest overhead, however, any scheme intended to provide non-repudiation and message integrity is likely to be time-consuming if it is to withstand brute force attacks. Should new more efficient asymmetric solutions to the identity and integrity problem come into place, the cost of accountability would likely decrease.

2.2 Toward General Accountability

We are interested in a 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.

Our goal is to provide strong means to detect and isolate faults and to contain them without relying on replication, voting, or secure hardware (see Section 2.3). A key challenge is that designing for accountability is not apparently general: accountability must be “designed in” to application structure, protocols, and specification.

In the remaining of this section we present an overview of two different angles of addressing the problem of accountability, which are at the heart of this thesis work.
2.2.1 State-based Accountability

General solutions to the problem of accountability require a level of abstraction to mask the differences among diverse applications. The model should be general enough to encompass a large number of systems of interest, but it should also be sufficiently realistic so that it can be applied in practice. Our work in this direction is based on a model, which focuses on services with state, whose actions are solely defined as operations on that state. This state-based approach focuses primarily on the operations services perform to manage their state: read, write, and update, and abstracts out the process by which the contents of a write or an update has been derived. The resulting methodology makes it possible to integrate varying levels of accountability within a system, by focusing primarily on updates to internal state as a result of the requests issued against the service.

Services built using this methodology can supply self-certifying evidence that can prove or disprove the correctness of an action. While our approach can provide in a general and uniform way all information needed to make a decision on an action’s correctness, i.e., the sequence of reads and updates to internal state which the action required and produced, the process of verifying the computation that provided the value for a given update is highly service-specific; full verification requires detailed knowledge of the semantics of each action. We describe our work in this direction in the next four chapters.

An important aspect of the state-based approach is that it enforces accountability for state changes. If an action does not produce a state change, it cannot be made accountable using these techniques. For example, reading private information does not involve any state changes, only a copy of existing state. We examine this problem in Chapter 7.

2.2.2 Application-specific Accountability

An alternative to application-independent approaches to building accountable systems is to exploit the specific properties of a service in an attempt to design accountability techniques, which may not apply to other services, but may work very well for the service in question. Such approaches may require significant upfront investment in research and development, but may also result into more efficient and streamlined solutions. Large systems in which semantic violations constitute a significant threat can well justify the cost associated with this approach.
In this thesis work we apply this line of research to the problem of accountability in the context of distributed resource management. We identify several key areas, specific to distributed resource management, and design accountability solutions that leverage their properties. In particular, we present techniques to ensure accountable spending and allocation of virtual currency and accountable fine-grain delegation of distributed resources (Chapter 8).

2.3 Related Work

The concept of accountability appears in various forms in many previous systems. The essential distinction is that we focus on a strong form of accountability in which participants can be challenged to form claims about the semantic correctness of their actions.

Several definitions of accountability exist depending on the target context. Butler Lampson summarizes accountability as follows: “Accountability means that you can punish misbehavior”. Our community often uses the term to refer to a means of enforcing an authorization policy by detecting and punishing violations after the fact, as an alternative to incurring the cost of preventing them in the first place [69]. More recent work seeks to enforce accountability for performance behavior as essential to effective functioning of networking markets [71].

Our notion of “semantic” accountability is different and complementary. We consider actors accountable if they can demonstrate that their actions are semantically correct as well as being properly authorized. An accountable server can be held responsible if its responses violate semantic expectations for the service that it provides. Accountability is “strong” if evidence of misbehavior is independently verifiable by a third party. Clients of a strongly accountable server are also strongly accountable for their actions, since they cannot deny their actions or blame the server.

Subversion is one source of misbehavior in a computer system. Secure perimeters [69], secure hardware [113], and Byzantine Fault Tolerance [29, 108] strive to prevent subversion. But more generally, participants in a cooperative system may have an incentive to behave unfaithfully [3]. Accountability is a tool that complements the above techniques and makes it possible to ensure faithfulness to a specification and semantic norms of interaction.

Designing accountable systems complements existing solutions to building trustworthy systems such as Byzantine Fault Tolerance [29, 108], secure hardware [113], and secure perimeters [69].
2.3.1 Theoretical Approaches

Program correctness has been widely studies in theoretical computer science. In complexity theory the class NP consists of problems that cannot be solved in polynomial time, but the result of each can be verified by using a polynomial time algorithm. The idea of verification can also be found in the different proof systems, e.g., zero-knowledge, interactive, and probabilistically checkable, which can be used to verify the validity of statements and program results [52]. The famous PCP theorem [9] states that any problem in NP can be verified probabilistically by inspecting a constant number of bits from the supplied proof. Aiello et al. build on this result and propose a technique to verify the correctness of any remote procedure call [2]. Their approach uses the PCP theorem and Single Database Private Information Retrieval techniques [66] to guarantee one-round interactive proof with poly-logarithmic communication cost. Secure multi-party computation is yet another related area [53], as it tries to evaluate securely the output of a function requiring inputs held by multiple parties. Cachin et al. is an example of work in this category [27]; their solution ensures correctness by converting computation to encrypted circuit evaluation.

These theoretical approaches have a number of useful properties, but they are difficult to be applied in practice. The most important property of these techniques is that the end result is guaranteed to be correct: clients have explicit mechanisms to verify the correctness of an operation. The most significant limitation is due to the size and complexity of the corresponding proofs—these solutions depend on complex reductions and encoding of information, which make their application virtually impractical. Even if these techniques were practical, they would not be sufficient to ensure accountability, as they do not offer techniques to analyze the sequence of actions that transition a system to a given state. Furthermore, these solutions do not offer mechanisms to associate responsibility for given state and actions. For example, secure multi-party computation evaluates a function correctly given the inputs of multiple parties. However, when one of the parties is a service with state resulting from the interaction of its clients, there is no guarantee that the input supplied by the service is correct. Our proposal is complimentary to work in this area and adds an additional level of correctness guarantees.

Blum introduced the idea of a program checker—a trusted component that mediates the communication between clients and servers [20]. In this model the checker observes all client requests and
server responses and uses some (small) amount of secure memory to maintain specific information to help verify the correctness of server operations. This model has been used to certify operations on linked data structures such as stacks, queues, lists, trees, and graphs [5]. This model, while applicable in some contexts, is not practical—the assumption that the checker can observe all requests and responses does not hold for real applications. Our work can be considered as an attempt to relax Blum’s model by verifying correctness without observing all client requests and responses.

Another stab at the problem of program correctness is the certification-trail technique [22, 117]. The idea of this approach is to run an application twice. During the first execution, the application generates the expected output and a certification trail: a record of some of the application state at different stages of the execution. During the second execution, another program uses the certification trail to determine whether an error has occurred. The main limitation of this approach is that the certification trail is trusted; this technique does not account for maliciously created seemingly correct certification trail.

### 2.3.2 Secure Perimeters

Secure perimeters attempt to prevent misbehavior by means of authorization and authentication. However, when actors reside in different administrative domains, there is no common perimeter to defend. Once the integrity of the perimeter is violated, it offers no protection. More importantly, secure perimeters can be insufficient, as even properly authenticated actors may abuse the system to their own advantage.

Accountability is a defense in-depth that extends the protection of secure perimeters by adding active after the fact challenges to ensure that actor behavior is semantically correct with respect to the rules and responsibilities prescribed by the system. Accountability’s primary mantra is “trust but verify.”

Logging and auditing can also be used to ensure correctness. The key challenges of logging are where to store the logs and how to maintain their integrity. An example solution to the integrity problem can be found in [78, 109, 110]. However, logging by itself is not sufficient to provide accountability as one requires a more rigorous structure of each log record to meet the requirements of accountability.
2.3.3 Secure Hardware

Secure hardware offers foundational mechanisms to ensure untampered execution at the hardware level. Secure hardware can be used to protect program execution on an untrusted platform, to provide copy protection as well as digital cash and post stamps. A number of commercial secure processor systems exist, e.g. IBM Citadel [128], µABYSS [126], etc. A detailed description of the applications of secure hardware can be found in [113, 130]. A significant consequence of using secure hardware is that it requires hardware changes, which can be costly and may take time to apply on a large scale.

The use of secure hardware is not sufficient to provide accountability. While a secure hardware platform can ensure that a program’s code is not modified, ensuring that the program’s actions are semantically correct currently requires verification and certification by a trusted third party. To avoid the need for certification one could design a trusted operating system that can ensure accountability with techniques similar to the ones we describe in this dissertation.

Secure hardware can be used to span the gap between the human user and software, thus providing a trusted path to the user and ensuring that applications a user interacts with issue requests in compliance with the user’s demands. User interface approaches [129] are another alternative to building a trusted path.

Intel’s Trusted Execution Technology [37], formerly known as Intel LaGrande, is a proposal to secure computation and data with hardware support. Intel identifies several areas of particular interest: protected execution, sealed storage, protected input, protected graphics, attestations, and protected launch. The goal of each is to secure different stages of the lifecycle of an application and provide guarantees about the correctness of execution and privacy of information. Once this technology becomes widely available, it could help make systems more accountable by solving one of the fundamental accountability problems: being able to distinguish between the actions of a computer and its user.

Microsoft’s Next Generation Secure Computing Base (NGSCB) [38], formerly known as Palladium, attempts to improve the security and privacy of computer systems. NGSCB intends to combine secure hardware, such as Trusted Platform Module (TPM), and software to provide some guarantees about the security of a running system, i.e, the system has not been tampered with and
no private data has been leaked.

2.3.4 Byzantine Fault Tolerance

N-version programming [10] and Byzantine fault tolerance [29, 108] use multiple implementations and/or state-machine replication to protect against faulty/malicious parties. These techniques are expensive as they perform each computation multiple times and somewhat unrealistic since they assume that replicas cannot communicate. As a result, these techniques are vulnerable to collusion and do not offer accountability as defined in this thesis.

Byzantine fault tolerance (BFT) is a general and powerful technique to tolerate misbehavior by using replication and voting. However, BFT is vulnerable to collusion: a majority of actors may collude to "frame" another. Also, in its pure form, BFT assumes that failures are independent, and thus it is vulnerable to "groupthink" in which an attack succeeds against a quorum of replicas.

BFT offers limited protection when a service (and hence its replica set) is itself acting on behalf of an entity that controls its behavior (typical of web services). Using BFT in this context would require replication on a higher level, which can be significantly more expensive. Importantly, our approach can produce proofs of misbehavior that do not depend on voting or consensus, which makes statements of correctness or misbehavior provable to a third party.

2.3.5 Security Standards

In the Web Services domain, new standards for non-repudiable, certified messages are useful building blocks for accountable systems (e.g., WS-Security). They also facilitate richer trust management schemes through transitive delegation and endorsement of authority by signed security assertions (e.g., SAML).

WS-Security [62] is a joint effort by Microsoft, IBM, VeriSign, and others to create interoperable, platform-independent security mechanisms for XML Web Services. WS-Security defines enhancements to the underlying SOAP [123] transport to protect the integrity and confidentiality of messages. WS-Security makes use of XML Signature [122] and standard security tokens (e.g., X.509 certificates or Kerberos tickets) to ensure message integrity.
2.4 Summary

This chapter defines the notions of accountability and strong accountability. It describes the basic properties of actions in an accountable system and how these properties can be used as a means to building dependable trustworthy systems. We emphasize the importance of well-defined action semantics and the association of actors with their actions. The chapter presents an example of a strongly accountable system and outlines the challenges of extending the example to other systems and our approach to dealing with these challenges by attacking the problem along two different, yet complementary directions. The primary direction of our research attempts to find generic solutions to the problem of accountability, while the secondary, is focused on using application-specific knowledge in designing accountability solutions.

We also position our approach to building trustworthy systems relative to other existing techniques. The primary difference is that we focus on a strong form of accountability, in which individual system participants can be challenged to form claims about the semantic correctness of their actions.
Chapter 3

State-based Accountability

This chapter presents the basics of the state-based approach to building accountable network services. We start by describing an abstract model of stateful network services. Using the model, we present a threat model that captures attacks on the correctness, consistency, and accountability of network services. In the subsequent sections we use the threat model to describe the components of a framework for building accountable services. We conclude the chapter with a discussion of work related to the area of state-based accountability.

3.1 Stateful Network Services

A general-purpose solutions to the problem of accountability must use some form of an abstract model to reason about the behavior and organization of network services. In this section we present our model of network service internal organization and request processing. This model is an attempt to address the problem of accountability in its most fundamental form: ensuring that a service’s actions are consistent with its internal state. In constructing this model we focus on services whose operations depend primarily on their internal state. Such services accept requests to retrieve or transform their internal state according to some well-defined logic. The result of each request is a function of the user input and the internal service state.

Our state-based model presents a general abstraction that applies to a large class of services; it captures the most fundamental operations that state-driven services perform. This model is a step towards establishing a rigorous basis for a key aspect of accountability (internal state management), and provides the means to integrate accountability into a general class of network services.

3.1.1 Service Model

A service is a collection of well-defined logic and state. The state of a service is a set $S$ of state variables with unique names derived from an index set $I$. Each variable’s value belongs to a service value set $V$. The value set of a service obeys various service-specific restrictions, which may constrain
the range and type of each variable’s values.

\[ S = \{ v_i, v_{i2}, v_{i3}, \ldots | i_j \in I, v_{i_j} \in V \} \quad (3.1) \]

The logic of a service is the set \( L \) of action functions, one for each action supported by the service. The set \( L \) is finite, i.e., each service offers a finite number of actions.

\[ L = \{ f_1, f_2, \ldots, f_n | n \in N \}. \quad (3.2) \]

Each action function takes a finite number of client-supplied arguments.

\[ f = f(a_1, a_2, \ldots, a_k | k \in N) \quad (3.3) \]

The evaluation of an action function produces a final result \( O \) and uses two index sets: a read index set \( R_{f(a_1, a_2, \ldots, a_k)} \subseteq I \) and a write index set \( W_{f(a_1, a_2, \ldots, a_k)} \subseteq I \). The read index set specifies the names of state variables that are read, i.e., serve as additional inputs, during function evaluation, while the write index set specifies the names of state variables that are updated (written to) during function evaluation. For a given action function, either \( R \) or \( W \) can be the empty set, but not both. That is, each service action either retrieves a value derived from at least one state variable, or updates the value of at least one state variable. In other words, in our model, no result \( O \) is derived from data not represented in the service state. It is possible that for some action functions neither \( R \) nor \( W \) will be empty. We refer to actions with empty write sets as reads, and actions with non-empty write sets as writes.

Given this model, the execution of an action \( A \) with an action function \( f(a_1, a_2, \ldots, a_k) \), which is requested by a client \( C \), proceeds as follows:

1. The service receives, parses, and verifies the request. That is, the service identifies \( C \) as the requestor of the action, authenticates \( C \), and performs access control checks to determine if \( C \) holds the necessary privileges to request the action.

2. The service verifies each argument to the associated action function, performs additional access control checks, if necessary, and rejects the request if an argument verification fails.
3. The service evaluates the action function \( f(a_1, a_2, ..., a_k) \). In the evaluation process, the service computes the read and write index sets, computes the new values for the state variables referenced by the write index set, updates the variables referenced by the write index set and produces the final result \( O \).

4. The service sends \( O \) back to the client \( C \).

Before we proceed further we need to make the following definitions:

**Definition 3.1.1. (Well-defined action)** An action is well-defined, if its corresponding action function \( f \) is deterministic and explicitly specified over its entire domain. That is, for each combination of its input arguments, there exists a specification, which defines how to calculate the read and write index sets, the values for variables referenced by the write index set, as well as the final result.

In other words, a well-defined action is an action that can be verified. That is, for a well-defined action one can obtain the action specification, the client inputs, and the relevant state variables, and apply the function definition to reproduce the function evaluation and to verify if the action was executed according to the specification. While the full specification may not be publicly available, there must exist channels through which the specification may be obtained when necessary.

**Definition 3.1.2. (Correct action)** An action is correct, if: (1) it is well-defined, (2) the requestor \( C \) holds all required privileges, (3) the client supplied arguments fall in the action function’s domain, and (4) the executor \( E \), has calculated the read index set, the write index set, the values for the variables referenced by the write index set, and the final result of the associated action function according to the action specification. A well-defined action is incorrect if it does not meet at least one of the aforementioned requirements.

**Definition 3.1.3. (Action context)** The state of a service immediately prior to the execution of an action makes the context of the action.

No action exists in isolation: each action either precedes or follows another action. Therefore an action’s result has an impact on the result of future actions, and/or is affected by the actions
that happened before it. In terms of our model, even if an action meets all requirements of Definition 3.1.2, it may still produce unexpected results, because the inputs used to evaluate its action have been produced by an incorrect action. That is, the correctness of action may be affected by the correctness of actions that happened before it.

The recursive nature of correctness implies that even a single incorrect action can have significance consequence over the operation of a network service and its clients: any incorrectly updated state variable may participate in the evaluation of one or more action functions (done by the service or by its clients), and may thus produce incorrect values for other state variables. Left unchecked, this process can impact the state of every actor in an interconnected system.

Definition 3.1.4. (Verification function) The verification function $v_f$ of an action $f$, takes as inputs: (1) $f$’s client inputs $- a_1, a_2, ..., a_k$, (2) $f$’s result $- O$, (3) all state variables referenced by $f$’s read index set $R_f$ (at the time the evaluation of $f$ commenced), and (4) all state variables referenced by $f$’s write index set $W_f$, and outputs 0 if $f$ was evaluated correctly, and a non-zero value otherwise.

$$v_f(a_1, a_2, ..., a_k, S_{R_f}^{Before}, S_{W_f}^{After}, O) = \begin{cases} 0 & \text{if } f \text{ is correctly evaluated;} \\ \neq 0 & \text{otherwise.} \end{cases} \quad (3.4)$$

Each well-defined action has at least one verification function, which equals the difference of the re-evaluation of its action function and the original value produced by the action function. Therefore, assuming the values of state variables in the read set are themselves correct, the evaluation of a verification function is at most as expensive as the evaluation of the original function.

If the validity of the variables in the read set is not known, then the evaluation of a verification function may require the recursive evaluation of the verification functions of the action functions responsible for the variables in the read set. Therefore, in the worst case the evaluation of a verification function may involve re-evaluating every action a service has performed over its lifetime.

Some action functions may have efficient verification functions, i.e., the verification function can be computed using less time and resources as compared to the original function. For example, every problem in NP has a polynomial time verifier [35]. Section 3.2.6 examines action and verification functions in more details.
3.1.2 Separation of State and Logic

Using the service model from Section 3.1.1, we can decompose the execution of every action into two interdependent steps, which may be executed multiple times in the course of a single action.

- **Action function evaluation.** In this step the service evaluates its action function using the supplied inputs from the client and values of state variables retrieved from its internal state. While computation at this stage depends on input arguments and state variables, the performed operations are primarily defined by the specification of the action function. As a result, work performed at this stage is highly service-specific.

- **State variables updates and retrievals.** In this step the service performs operations over its internal state. These operations may retrieve the values of one or more state variable or may update the values of one or more state variables (or create new variables) using values produced by the previous step. While data stored in a state variable may depend on computation performed in the previous stage, the process of storage and retrieval of state variables is similar for all services described by our model: retrievals return the value of a variable in the service state, updates add a new variable or change the value of an existing variable, which is already in the service state.

For a given accountable service one should be able to verify the operations performed in either of the aforementioned steps. A general solution to the problem of accountability, however, must focus on the subset of operations shared among all stateful services, and provide mechanisms and guidelines for dealing with the service-specific aspects. In essence, this is the core of our state-based approach to building accountable services: we separate both stages of action execution and focus on the process of managing internal service state. Accountable internal state operations are then used as building blocks to verify more complex, service-specific operations, e.g., action function evaluations.

3.1.3 Threat Model

A stateful network service may fail to execute a requested action correctly due to a combination of a number of factors and conditions. In this section we describe the range of threats that a solution
to the problem of accountability must address. The threats listed here address both service logic and internal state maintenance issues.

**Authorization and request validation.** A service may fail to validate an incoming request. The service may accept inputs, which do not fall in the domain of the associated action function, or may not perform all required access control checks. For example, the service may authorize a client to execute operations, which require privileges not held by the client, or involve access to state variables, which the client is not allowed to access.

**Action function evaluation.** In the course of executing an action, the service may evaluate the associated action function incorrectly. Such violations can take different forms:

- The service may evaluate correctly a different, but valid, action function $f_1 \in L$ instead of $f \in L$.
- The service may not evaluate correctly the read and/or write index sets $R_f$ and $W_f$.
- Given all client inputs and internal state variables, the service may simply not follow its specification and produce a result, and/or values for the variables in $W_f$, different from what it would have produced if it followed the specification of $f$.

**Internal state maintenance.** The values of state variables stored or read from the service state may represent incorrectly the results of the service operations. In general, a service, which manages its internal state correctly possesses the following properties:

- **Authenticity and undeniability.** The service updates a state variable only as a result of a properly authorized client request. All state updates can be traced back to a valid client request. Importantly, for each state update, the associated client cannot deny responsibility for having requested an action that has impacted the state variable.

- **Freshness and consistency.** A single state variable may be updated multiple times as actions execute. All updates to a variable must be applied in order, so that the service maintains the consistency of its state. Out-of-order execution should be detectable and provable as it may impact correctness. Most importantly, reads from a state variable should always return the value of the state variable that represents the latest update to that variable.
• **Inclusion.** An update to a state variable should be included in the service state and must be visible and accessible to operations performed after the update; the service cannot hide the fact that a state variable with a given value exists. The result of actions affect the service state in a predictable way that can be verified externally; it is impossible for the service, or a client, to deny executing/requesting an operation once the operation is complete.

A faulty service could attempt to violate any of the above properties. For example, it could perform updates to its internal state as a result of requests from unauthorized clients, it can improperly modify existing state variables, or replay valid client requests. In a more subtle attack the service could acknowledge the completion of an action, but attempt to conceal the execution of the action from other clients. The service or a client could attempt to deny that it executed or requested a completed operation.

### 3.1.4 Trust Assumptions

<table>
<thead>
<tr>
<th>Component</th>
<th>Trust Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clients and servers</td>
<td>Trusted but accountable: Cannot subvert the system without risk of provable detection. Incorrect or invalid actions taken with a stolen key are provably detectable.</td>
</tr>
<tr>
<td>Publishing medium</td>
<td>Trusted to render published digests visible to all participants. Accountable for forgeries or alterations to published digests. Cannot subvert the system without risk of provable detection.</td>
</tr>
<tr>
<td>Authorization service</td>
<td>Not required for simple static access control lists. Must be trusted to enforce richer access control policies if needed.</td>
</tr>
<tr>
<td>Trusted platform / trusted path</td>
<td>Necessary for individual user accountability, else no distinction between misbehavior of user and misbehavior of user software.</td>
</tr>
<tr>
<td>Public key infrastructure:</td>
<td>Trusted Computing Base: Compromise of PKI can subvert the system.</td>
</tr>
</tbody>
</table>

Table 3.1 lists the components of an accountable service built using the state-based approach. The accountability properties of the system rest on correct functioning of one core element.

**Asymmetric cryptography.** Each actor can sign its requests using at least one asymmetric key pair bound to a principal; the public key is distributed securely to all actors, e.g., using a Public Key Infrastructure (PKI). Digital signatures ensure integrity, authenticity, and non-repudiation of actions.
Secure authentication is the core trust assumption in our system. If keys are compromised, then an actor may be falsely held accountable for actions it did not take. Note, however, that an attacker cannot misrepresent the actions of any actor whose key it does not possess, and any actions it takes with a stolen key can be traced to that key. Even so, a successful attack against a trusted PKI root would open services to subversion by the attacker; thus a PKI constitutes a Trusted Computing Base in the traditional sense.

The PKI used to certify identities and keys forms the core of the trusted base. No other component is explicitly trusted to behave according to its specifications. The elements of the state-based approach are designed to leverage the trusted base and hold elements outside of the trusted base accountable for their actions. Making the process work end-to-end may require some additional components (authorization services and trusted paths) and we discuss them in Section 3.1.6.

External publishing medium. Each actor has a means to publish a digest of its state periodically to its clients and to an external publishing medium. A digest is a one-way function computed over the actor’s state at a point in time. Digests are digitally signed, so an actor cannot repudiate previous claims about its state. While both clients and servers can publish digests of their states, in general only actors, that act as servers to other actors (services) are required to publish their digests.

Each actor must have independent access to the history of published digests in order to validate proofs independently. Since digests are signed, a faulty publishing medium cannot forge or alter digests unilaterally, but it could mount a denial-of-accountability attack by concealing them. The medium could also collude with a participant to alter the digest history of that participant, but such an attack would be detectable and provable by another participant that caches previously published digests. In essence, a faulty publishing medium can weaken the accountability properties of the system, but it cannot itself subvert the system. The external publishing medium is not part of the trusted computing base.

Due diligence. State-based accountability relies on voluntary actions by each actor to verify the behavior of the others. If the clients of a service choose not to request or check proofs that their requests were executed and persist (see Section 3.1.5), then a faulty service may escape detection. Of course, a lazy client may free-ride on the diligence of others, possibly leading to a classic tragedy
of the commons. In general, due diligence is within an actor’s best interest, especially if this actor provides service to other actors, for which it itself could be held accountable. The important point is that an attacker cannot determine or control its risk of exposure to verification.

3.1.5 Challenges and Audits

An important element of our approach is to incorporate *challenge* (Section 3.2.3) and *audit* (Section 3.2.5) interfaces into service protocols. Challenges force a server to provide a cryptographic proof certifying that its actions are correct and consistent relative to published state digests. An important form of challenge is an *audit* to verify consistent behavior across a sequence of actions or an interval of history. Challenges and audits do not require trust in the auditor; any actor may act as an auditor. If an actor’s challenge or audit reveals misbehavior, the actor can present its case as a *plaintiff* to any other actor, which may verify the validity of the accusation.

Auditing defends against a freshness attack, in which a faulty server discards or reverts valid state updates that it has previously accepted (Section 3.2.5). A client with authority to access a state variable can choose to audit the sequence of updates to that variable through time to ensure freshness and consistency. Actors select a degree of auditing that balances overhead and the probability of detection of misbehavior. The server cannot change its history to conceal its misbehavior from an auditor without being detected.

For example, a service may be challenged to prove that it has incorporated a recently completed state update into its published state digest. It may be audited to prove that its current state resulted from a sequence of valid updates resulting from requests of authorized clients, and that reads from state variables reflect that state. A challenged or audited service must also justify that any accepted operation that causes a state update complies with the existing access control policy (See Section 3.1.6). A faulty server cannot allow unauthorized state updates or execute its own state updates using a fraudulent identity. If an attacker subverts the server, the worst damage it can cause is to deny service or discard data; covert modifications (including reverted state updates) are tamper-evident and can be exposed at any time through challenges and audits. In particular, the server can be held provably accountable for a forking attack [75].

36
3.1.6 Discussion and Limitations

User accountability, identity, and privacy. Ideally, the accountability of a system should extend to its users. For example, a service could hold users accountable for actions they take within a community or organization, possibly exposing them to legal sanction or other social sanction.

However, user accountability requires strong identity bindings. There is a fundamental tension between accountability and privacy or anonymity. Importantly, strong identities exist today within user communities that interact using shared services in areas such as health care, infrastructure control, enterprise work flow, finance, and government, where accountability is particularly important. Many of these areas already use some form of PKI.

User accountability also requires a trusted path to the user so that software running with the user’s identity cannot act without the user’s consent. This requirement ensures end-to-end accountability by removing the possibility of end users avoiding responsibility by blaming a problem to hardware and software that acts on their behalf. This is a very difficult problem and its solution is likely to require some trust in the platform, e.g., a Trusted Platform Module [120]. We defer this question until Chapter 7 where we study the problem of data leaks and confidentiality.

More generally, a signed message or action is not a guarantee of intent by the principal: many security breaches occur when components are subverted or private keys are otherwise stolen [8]. Importantly, when such a breach occurs, our approach ensures that attackers cannot modify existing history, as noted above.

External publishing medium. The external publishing medium is a certified log that is globally visible. The publishing medium is a very simple service: it does not accept updates, does not support dynamic object creation, and does not allow write-sharing. There are several ways to implement an external publishing medium. Timestamping services [25] publish digests using write-once media such as newspapers. An alternative solution may use a trusted web site. The clients of the service can also implement the publishing medium by using a peer-to-peer approach that leverages some form of gossip and secure broadcast.

Access control. As part of its operation a service should demonstrate that processed requests from its clients comply with access control policies, both for the types of requests a client can issue, as well as the state variables that a request may affect. It is easy to hold the service accountable for
enforcing a simple static policy for updates to its internal state; for example, if the client responsible for the creation of a state variable provides an immutable list of identities permitted to modify the state variable. A server may also use an external authorization service to govern access control policy and issue signed assertions endorsing specific clients to access specific state variables. However, such an authorization server must be trusted. Chapter 6 describes how to reduce trust requirements by making authorization services accountable.

**Denial of service through challenges and audits.** One concern is that challenges and audits could be used as the basis for a denial-of-service attack on the server. We view this problem as an issue of performance isolation and quality of service: resources expended to serve audits and challenges may be controlled in the same way as regular service requests. Community standards may define a “statute of limitations” that limits the history subject to audit, and bounds the rate at which each actor may legitimately request audits and challenges. Any limitations and regulations, however, must be verifiable and accountable.

**Violations of Privacy** State-based accountability focuses primarily on operations that update a service’s state. Such operations are subject to strict access control decisions, and each “write” access control check can be audited and verified. However, the state-based approach has an inherent limitation when “read” access control checks are involved; a service may disclose at will any information it stores on behalf of its clients. Thus, using the state-based approach a service cannot be held accountable for breaches of confidentiality. We revisit the problem of privacy and confidential information in Chapter 7.

### 3.2 State-based Approach

The core principle of state-based accountability is to separate state management from state transformation due to application logic. Our approach represents service state as an indexed set of named, typed, elements. The service state store accepts updates annotated with a request—digitally signed by the issuer—and links these action records to the updated state variables. In this section we describe the basic elements of the approach. Chapter 4 describes our implementation of CATS, a reusable toolkit for certified accountable tamper-evident state management based on the state-based approach. The CATS toolkit provides foundational support for building accountable services.
(Chapter 5).

3.2.1 Signed Action Histories

Strong accountability requires a service to represent action histories whose integrity is protected, and provide primitives to retrieve, exchange, and certify those histories. We have to know who said what to whom, and how that information was used.

![Diagram](signed_actions.png)

**Figure 3.1**: Signed action histories are at the heart of accountability. An action history consists of digitally signed action records. The signature on each record helps identify the requestor/executor of the action and protects the integrity of the record's contents.

Every request and response to an accountable service carries a digital signature that uniquely identifies its sender and proves the integrity of the message. Specifically, the content of each request and response is encapsulated in a signed action record. Once issued, an action record cannot be modified without provable detection. Actors may retain a history of action records and transmit them to make provable statements about the behavior of other actors and the validity of their actions. The action history may be integrated into the internal service state. Signed action records are verifiable by any receiver and are non-repudiable by the sender.

For example, the CATS storage service (Chapter 5) links each stored object to the set of client requests that affected it or produced its value. A CATS storage server may then justify its responses by presenting a sequence of cached, signed action records that show how its state resulted from the actions of its clients, starting from its published state at a given point in time, as described below.
Figure 3.2: Each actor commits to its execution history by periodically distributing to other actors a view of its history. To preserve privacy and for improved efficiency, actors perform this step by computing state digests: compact representations of their action histories. Consecutive digests are chained, to prevent against addition/removal of state digests.

3.2.2 State Digests and Commitment

Using the state-based approach an accountable service periodically generates a signed compact representation of its internal state, a state digest. The digest is computed as the result of a collision and second pre-image resistant hash function applied over an atomic snapshot of the state of the service. The resulting digest has the essential property that it is computationally hard to find a state modification that produces the same digest, i.e., it is infeasible for a attacker to perform changes to internal state without making them externally visible. State digests allow to "peek" inside the internal state of a service and detect state changes without exposing private information. Efficient computation and verification of state digests is one of the core enabling mechanisms of the state-based approach. We examine the possible options and the data structures required to support them in Chapter 4.

While digests are logarithmic in size relative to the size of the service state, for all practical purposes we can consider digests to be of constant size, e.g., a SHA-1 [89] digest of 160 bits can be used for services with state approximately up to $2^{160}$ bits. This size limit is likely to be beyond the size of any existing service, considering than the estimated number of atoms in the observable universe is around $10^{80}$.

Servers sign and publish their digests by including them in their responses to clients and publishing them to an external medium (Section 3.1.4). Publishing a digest commits the server to a view
of its state and its history at a specific point in time. The server cannot lie about what its state was at that time without the risk of exposure. Any modification of the state affects the digest; any conflicts with previously published digests are detectable and provable. In particular, the server cannot safely present different views of its state to different clients—a forking attack [75, 81] (Section 5.4). While nothing prevents the server from returning different digest values to different clients, clients can detect such misbehavior by comparing digests among themselves or against the published copy. Two conflicting signed digests constitute an undeniable evidence of the server’s misbehavior.

An epoch is the time interval between generation of two consecutive state digests by some actor. An epoch is committed when the digest of its end-state is published. Epochs are numbered with consecutive timestamp values. For example, epoch numbers may equal the state digest number to be published to the publishing medium. To preserve the integrity of the timeline, the computation of each new state digest incorporates the digest from the previous epoch as an input. This form of chaining prevents the service from removing a committed snapshot or adding a new snapshot in the past; snapshots can only be added at the end of the timeline.

### 3.2.3 Proofs

![Diagram](image)

**Figure 3.3:** Services can make provable statements about the contents of their action records. In particular, it is possible to construct proofs to demonstrate that an action record is a member of the action history set at a particular time instance. Similarly, services can construct proofs showing that an action is not a member of the set. Proofs of inclusion and exclusion can be verified independently against the published state digests of the service.

An important consequence of using state digests to summarize the contents of internal state is that a service can leverage digests to make provable statements about its state and the operations that it has performed. Since a digest is computed over a snapshot of the service state, the digest reflects the presence or absence of state variables in a given snapshot. For each state variable present in the service state, the digest also reflects its value and any of its attributes whose integrity must
be protected. This essential property enables a service to provide two types of proofs relative to a published state digests:

- **Inclusion.** The service claims that a state variable \( v_k \) with a given name and value was recorded in the service state at the end of some epoch: i.e., \( k \in I^T \) and \( v_k \in S^T \). The claim is backed by an unforgeable **membership proof** showing that the state variable’s name and value were used to compute the state digest for that epoch.

- **Exclusion.** The service claims that no state variable \( v_k \) with a given name was recorded in the service state at the end of some epoch: i.e., \( k \notin I^T \) or \( v_k \notin S^T \). The claim is backed by a proof exhibiting a proper computation of the epoch digest that did not include the name and value of \( v_k \).

Efficient proof generation and verification requires specific data structures and algorithms designed to minimize the size of a proof and the time spent generating and verifying it. In addition to performance, one must also consider the privacy and sensitivity of information contained in either of the proof types. Ideally, a client of the service, should not be able to use the proof mechanisms to extract information about unrelated activities of other clients; proofs should expose only the minimum information necessary to substantiate a given claim. We address these and other related design and implementation issues in Chapter 4.

A client may request a proof by issuing a **challenge** on a target state variable. The server responds to a challenge with an inclusion proof if the variable was a part of the state, or an exclusion proof if it was not. Optionally, if required by the client, a challenge may include a signed copy of the request responsible for the state variable’s value.

Challenges enable a client to substantiate any service response. That is, for any state variable in the action’s read set \( S_{R_j} \), the client can issue a challenge to obtain the value of the state variable and to ensure that value was indeed included in the state of the service at the time the request was processed, i.e., the client’s view of the server’s state is consistent with the server’s digests, and therefore (by extension) consistent with the views of other clients.

Similarly, for any variable in the action’s write set \( S_{W_j} \), the service can issue a challenge to obtain the new value (if necessary) and ensure that the variable has been associated with the client’s request.
and included in the service state and included in the published digest in a proper way that precludes the server from denying or safely concealing the update at a later time. Such verification ensures that state updates become visible to other actors.

Challenges alone are insufficient to verify the correctness of an action. A sequence of challenges can supply all arguments needed to evaluate an action’s verification function. Full action verification requires that the verifier has access to the action’s verification function as well to the resources needed to perform its evaluation. Even if a client does not possess all resources to evaluate a given verification function, the client is in possession of data, which could be used to demonstrate a server’s fault at a later time.

### 3.2.4 Request Ordering and Consistency

While services that are trusted unconditionally have a significant flexibility in dealing with request ordering and consistency issues, e.g., to offer higher levels of concurrency and improve performance, an accountable service’s ability to affect order and consistency must be restricted to avoid subtle timing attacks. Since the value of each state variable results from an ordered sequence of updates to the service state, accountability for the values of state variables requires that the server and its clients agree on and certify a serial order of requests and updates to state variables.

![Figure 3.4: Consistency protocol for accountable services.](image)

Each state variable is annotated with a version number. Update requests specify the expected version number for each state variable in the action write set \( W \). The service honors requests with matching version numbers, and must reject requests with stale version numbers. Rejections optionally supply the new values of all affected variables. The client resolves the conflict and issues an updated request (if necessary).

Each state variable is annotated with a version stamp, consisting of a hash of the variable’s value and the epoch number of the update that produced the variable’s value. All version stamps for a
given variable are unique, they differ at least by epoch number, and they can be ordered. A state variable can be updated at most once in a single epoch. If multiple updates to the same variable fall in the same epoch, the service can accept only one and must reject all others until the next epoch.

To ensure a serial order of requests we use a form of client-driven optimistic consistency scheme. The action record associated with an updated state variable must include the variable’s previous version stamp, certifying that the client was aware of the variable’s value at the time of the update. The server must reject a write action, action with a non empty write set \( S_{W_j} \), if the version stamp of any variable in its read \( S_{R_j} \) or write \( S_{W_j} \) sets has changed since the moment the action was issued, e.g., an intervening write has modified the service state and may thus affect the consistency of the requested action. In our current approach, the service is not allowed to perform update conflict resolutions. Clients must retry their requests, possibly after completing one or more read-only actions (actions with empty write sets) and adjusting their requests.

This scheme allows a client or an auditor to determine if the service has executed a given write action in the correct order. For each variable in the action’s write set the version stamp in a variable’s action record should equal the version stamp of the previous version. Similarly, for each variable in the action’s read set, the version stamp of the variable’s action record should equal the version stamp of the current version. By applying the same rules to a sequence of writes, it is possible to determine if any of them was executed out of order. This scheme works equally well for servers and clients: verification reveals any attempts by the server to reorder updates, and it also prevents a client from denying knowledge of a particular update, which took place before issuing its request.

We can relax the requirement that each action record contains the version stamps of all variables in the action’s read and write sets by only including the epoch number and state digests of the epoch of the last service state snapshot the client observed before issuing the action request. This information uniquely identifies the service state known to the client when it decided to issue its request. Given that information one can easily obtain the version stamps of all required state variables. The verification steps described in the previous paragraph remain unchanged.

Our request ordering scheme treats each action as a transaction, whose integrity depends on the integrity of its inputs. If a state variable used by a action A is changed by action B prior to the
execution of A, A is canceled. It is up to the client to determine if it still needs to execute A and with what arguments. This approach prevents a service from interleaving client requests and enforces strong compliance with the specification of each action function. At the same time, it may result in a large number of cancellations. Ideally, a service should be able to perform conflict resolution steps without having to cancel an action and wait for a new request from its client. Conflict resolution by the service, however, opens new attack vectors, defending against which requires more complex models and techniques. We leave this topic to future work.

3.2.5 Freshness and Auditing

It remains to provide a means to hold a server accountable for “freshness”: a state variable’s value should reflect its most recent update prior to the target time of the read. A faulty server might accept a write and revert it later, essentially providing a stale version of the variable’s value. Freshness is a fundamental problem: a server can prove that it does not conceal writes to an object over some interval only by replaying the object’s complete write history over the interval, including proofs that its value did not change during any epoch without a valid write action. Reverted writes are the most difficult form of the “forking attack” to defend against. In a forking attack, a server forks its internal state into multiple versions, presenting each to different clients. In essence, the fork consistency assurance of SUND [75, 81] is that if the server conceals writes, then it must do so consistently or its clients will detect a fault (Section 5.4).

Figure 3.5: At time $t_n$ the malicious operator reverts the effect of the update performed at time $t_m$ and sets the corresponding value to the one valid at time $t_k$. An audit that tracks the value through a sequence of snapshots detects this violation, as long as a read is performed in the interval $[t_m; t_n]$.

A participant may selectively audit a server’s write histories relative to its non-repudiable state digests. The participant chooses some state variables, and some state snapshots out of the execution history of the service and challenges the service to provide inclusion/exclusion properties for the
variables in question. The strong accountability property is that the server cannot conceal any
corruption of a state variable's value from an audit, and any misbehavior detected by an audit is
verifiable by a third party. This approach, however, is probabilistic—it detects misbehavior only
of the selected variables and state snapshots contain a misbehavior. Although reverted writes for
objects or intervals that escape audit may go undetected, a server cannot predict or control what
actions will be audited.

Figure 3.5 illustrates the auditing process. In this example, a state variable receives an update
at time $t_m$, which is correctly applied and the value of the variable becomes $v_4$. However, at a later
time, $t_n$, ($t_k < t_m < t_n$) the server reverts the value of the element to a previous value $v_3$, created
at time $t_k$. As a result of this intervention, subsequent reads to the variable produce an incorrect
result ($v_3$ instead of $v_4$).

To certify freshness, a client must challenge and examine the object's value for every epoch in
the interval since the last reported update to present. We will refer to this interval as the span of
auditing ($(t_k, now)$ in our example). If the audit confirms that the object and its reported value were
incorporated into all digests for the interval, then the client can safely conclude that the reported
value is indeed fresh. On the other hand, if the server cannot provide a membership proof for the
object and its value relative to some subsequent digest, then the server is faulty.

Clients can reduce the overhead of certifying freshness at the expense of a weaker, probabilistic
assurance. Instead of inspecting every snapshot in a span, the client may instead issue challenges
for randomly selected snapshots in the span. We refer to the number of audited epochs as the depth
of the audit. In the above example, the client performs an audit of depth 4. The audit successfully
detects the misbehavior, since it inspects the service state in a snapshot created in the interval
$[t_m, t_n)$, in which the object has the reverted value $v_4$.

The CATS toolkit (Chapter 4) and the CATS accountable storage service (Chapter 5) support
probabilistic audits as described in the above example. Clients can issue audit requests and specify
a list of randomly selected snapshots from a specified auditing span. CATS then constructs a
membership proof for the object and its value in the specified snapshots and returns the result back
to the client.

The probability of detecting a reverted update for a given state element depends on the lifetime
of the reverted update relative to the span of auditing. Using the above example, as the number of snapshots in \([t_n, \text{now}]\) increases, the probability of selecting a snapshot in \([t_m, t_n]\) decreases. Thus, detecting an offense with a fixed probability becomes more expensive as more time passes since the offense took place. If an incorrect update is not noticed earlier, it may be difficult and expensive to detect it later.

One way to deal with this problem is to examine more snapshots in the span (increase the depth of auditing) when verifying older objects. More formally, if \(p\) is the probability of selecting a snapshot that reveals misbehavior, the probability of detecting misbehavior after \(d\) snapshot inspections is \(P(p, d) = 1 - (1 - p)^d\). If the lifetime of a reverted update decreases relative to the span, then \(p\) decreases. Figure 3.6 shows the probability that audits of fixed depth detect that the value of an element is incorrect for different values of \(p\) and \(d\).

![Figure 3.6](image)

**Figure 3.6**: The longer a reverted update remains undetected (decreasing \(p\)), the larger the number of snapshots that must be inspected to ensure that an audit detects misbehavior with a given fixed probability.

Another possible solution is to have trusted auditors periodically inspect every object and ensure its freshness. The auditors publish commit records visible to all clients, certifying that the service state until the point of inspection is consistent.

An alternative, less expensive form of auditing can impose a probabilistic bound on the number of reverted updates. The core idea is to choose objects at random and inspect their values in randomly selected state snapshots. We make use of the following result:

**Proposition 3.2.1.** Conducting \(1/e\) successful audits of actions selected at random from an action set ensures that with probability at least \((1 - 1/e)\) the fraction of action records with a violation is
not more than $\epsilon$ ($0 < \epsilon < 1$).

**Proof.** If the fraction of actions with a violation is more than $\epsilon$, then the probability of selecting a single action without a violation is less then $1 - \epsilon$. Let $P$ be the probability of selecting at random $1/\epsilon$ actions without a violation. We have:

$$P < (1 - \epsilon)^{1/\epsilon}$$

We have from [86] that: $(1 + t/n)^n \leq e^t$ for $t, n \in R$, such that $n \geq 1$ and $|t| \leq n$. Substituting $t = -1$ we get that $(1 - 1/n)^n \leq 1/e$ for $n \geq 1$. If we substitute $n$ with $1/\epsilon$ (this is a valid substitution since $1/\epsilon \geq 1$), we get that:

$$P < (1 - \epsilon)^{1/\epsilon} \leq 1/e < 1/2$$

That is, if the fraction of actions with violation is more than $\epsilon$ then the probability of not detecting a violation after $1/\epsilon$ audits is $1/e$. Therefore, if we do not detect a violation after $1/\epsilon$ trials, than the probability of having no more than $\epsilon$ fraction of actions with violations is at least $1 - 1/e > 1/2$.

To give a concrete example, examining 3 action records at random and not detecting a violation, ensures that with probability 0.63 no more than $1/3$ of all action records have a violation in them. Importantly, we can increase the probability arbitrarily by repeating the procedure multiple times. It takes $O(\log 1/\delta)$ repetitions to boost the probability to $1 - \delta$, for $\delta \in (0, 1)$.

We use Proposition 3.2.1 to derive some bounds on the correctness of a service’s history:

**Theorem 3.2.1.** Examining the complete execution history of $n$ ($n > 0$) objects chosen uniformly at random from all state objects present in a service’s state, and ensuring the correctness of each update operation to these objects, guarantees that with probability at least $1 - 1/e$ the service has maintained correctly at least $(1 - 1/n)$ fraction of all objects over its complete execution history.

**Proof.** Follows directly from applying Proposition 3.2.1.
Theorem 3.2.2. Examining \( n \) \((n > 0)\) state update operations and ensuring that none of them reverts a previous update, ensures that with probability at least \( 1 - 1/e \), no more than an \( 1/n \) fraction of all state updates have been reverted.

Proof. Follows directly from applying Proposition 3.2.1.

Finally, another complementary approach is for each writer to challenge periodically objects it has written to ensure that its updates persist until another authorized client overwrites them.

3.2.6 Action Correctness

The previous sections describe a general approach to preserving the execution history of a network service. This approach forces a service and its clients to commit to the actions that they request and perform. Each action performed by the service leaves an imprint on the service’s internal state. The impact on the service state enables clients to verify that the service modified its state as a result of their requests and that data the service returns to them have been derived from the service’s internal state. Attempts by the service to conceal or reorder actions have a high probability of being detected. These steps, essentially, ensure that any violations committed by the service or its clients are detectable: the read and write sets of each action function are preserved in the internal state and cannot be modified without detection.

What remains is to verify that for a given action function the values the service produces for each variable in the action function’s write set are correct: that is, the service has correctly combined all its inputs and has produced a result that is correct relative to the semantic definition of the action. In essence, full verification requires us to apply the action’s verification function using the information preserved in the service’s internal state. While all previously described steps are generic and do not depend on the specific action a service performs, verification functions represent the service-specific component of our methodology. Since a verification function is directly related to the semantic definition of the corresponding action, it is unlikely that this step can be generalized. We come back to this question in Chapter 6.
3.3 “Fresh” Untrusted Data Structures

Section 3.2.5 presented an auditing approach to ensure the freshness of data items a service’s internal state. The freshness property is essential for state-based accountability as it ensures that no tempering has taken place. It is important to ask if it is possible to ensure freshness without relying on auditing. Can we construct a server-based data structure that always returns the latest value for each item stored into it? This section presents a preliminary treatment and some intuition as to why such a data structure most likely does not exist.

We present a preliminary theoretical treatment of the problem of building a data structure that always returns the latest version of each stored element. We define the problem, prove two basic lemmas, and state a proposition with some intuition as to why we believe it holds true. The material in this section is not complete and presents our current understanding of the problem. It is left to future work to provide a complete proof. Our solution strategy involves experimenting with different problem models and deriving information-theoretic minimal sizes for “authentication” information needed to ensure freshness.

Let \( D \) be a data structure used to organize key-value pairs. We can think of each key-value pair as a variable \( V_k \) with name \( k \) and a value \( v \). \( D \) resides on a remote machine and is maintained by an untrusted entity \( \mathcal{E} \). The clients of \( \mathcal{E} \) issue requests for operations to be performed on \( D \). Each request can introduce a variable, update a variable’s value, or retrieve the value of a variable. The clients of \( D \) do not trust that \( \mathcal{E} \) manages honestly and correctly all variables under its control. However, we assume that given a variable’s value, it is always possible to determine its originator, thus it is impossible for \( \mathcal{E} \) to introduce new variables and values at random.

**Definition 3.3.1.** \( D \) is **persistent** if any update to the data structure preserves the current version. Updates to a persistent data structure introduce a new version and preserve all previous versions. A persistent data structure consists of a series of snapshots and each represents the state of the data structure at a given point in time.

When \( \mathcal{E} \) is untrusted, persistence provides another level of protection if \( \mathcal{E} \) misbehaves. If \( D \) is persistent it will maintain all values ever associated with a given key. Inspecting the old values can help reason about the correctness of past actions. This is impossible if the data structure is ephemeral.
We assume that $\mathcal{E}$ and its clients share a common notion of time and that clients timestamp a variable’s value when they send it to $\mathcal{E}$. For each snapshot $S$, we have the following two invariants:

- The timestamp of any variables $v \in S$ is smaller or equal to the time stamp of $S$.
- If the timestamp of a variable is equal to the timestamp of the snapshot, this variable was created/updated in that snapshot.

**Definition 3.3.2.** Let $\mathcal{D}$ be a persistent data structure. $\mathcal{D}$ provides instance guarantees if each snapshot $S$ is:

- **Immutable** - the values of all variables that are part of $S$ are guaranteed to remain unchanged. Any attempt to modify $S$, remove a variable or change its value, breaks the integrity of $S$.

- **Certifiable** - it is possible to construct an undeniable proof that $v \in S$ (or $v \notin S$)

**Definition 3.3.3.** $\mathcal{E}$ can publish to an external medium $O(\log n)$ information about each of its snapshots. We will refer to this data as the authenticator of a snapshot.

The main role of an authenticator is to help reason about the correctness of operations performed on $\mathcal{D}$.

We assume that given the time of the snapshot, it is always possible to retrieve the authenticator of the snapshot. Moreover, we also assume that $\mathcal{E}$ cannot make conflicting claims about an authenticator; any attempt to publish different authenticators for the same snapshot can be detected.

**Definition 3.3.4.** $\mathcal{D}$ provides interval guarantees if an element $V_k$ that is created/updated in snapshot $S_i$ is guaranteed to retain its value in all subsequent snapshots until the first snapshot $S_{k'} (i < k)$ that results from the execution of a valid client update to $V_k$. Attempts to remove $V_k$ or to modify its value in any snapshot $S_j (i < j < k)$ destroy the integrity of the data structure and are undeniably detected.

**Lemma 3.3.1.** There exists a $\mathcal{D}$ that can provide instance guarantees such that in case the guarantee is violated one can always detect the violation.
Proof. By example. Let $D$ be a persistent hash tree. In this case, the authenticator for a snapshot is the label of the root of the tree. If we assume that the underlying hash function is collision resistant and second pre-image resistant, then any change to a variable in an existing snapshot will modify the label of the root node and will introduce a new snapshot authenticator (Immutable). For each variable $V_k$ in a given snapshot, $D$ can be used to construct a proof that it belongs to the snapshot and has a given value $v$ (Certifiable). To construct this proof, we provide the labels of the siblings of all nodes on the path from the root to the given variable. To verify the proof, one recursively recomputes the root authenticator using the provided information and compares the result to the officially reported authenticator. \hfill \Box

**Lemma 3.3.2.** There exists a $D$ that can provide interval guarantees such that in case the guarantee is violated one can always detect the violation.

Proof. From Lemma 3.3.1 we know that there exists a $D$ that provides instance guarantees. Earlier we noted that if the timestamp of a variable’s value equals the time stamp of its containing snapshot, then the value of the variable was created in the same snapshot. Since $D$ provides instance guarantees, reading a variable with a time stamp equal to the time stamp of the containing snapshot is always guaranteed to return the correct value of the variable or result in detection of tampering.

Problems occur when the timestamp $t_v$ of the variable $V_k$ we read is smaller than the timestamp $t_s$ of the snapshot from which we are reading. To verify that $D$ has not violated the interval guarantee, we need to go back to the snapshot with timestamp $t_v$ and examine all snapshots with timestamps between $t_v$ and $t_s$. Examination involves reading the value of $V_k$ from each snapshot. Because $D$ can provide instance guarantees, each read returns a proof to certify the validity of the result. Assuming that all proofs are valid, then all values of $V_k$ are actual values generated by clients of $D$. If the timestamp of all values we read is equal to $t_v$, then $D$ has not violated the interval guarantee. If there exists at least one value with a timestamp $t > t_v$, then $D$ has violated the interval guarantee and we have detected the violation. \hfill \Box

In Lemma 3.3.2 we showed that if a data structure satisfies the instance guarantee, then it is always possible to verify whether the data structure satisfies the interval guarantee. However, the verification process used in the proof of the lemma has cost linear in the number of snapshots.
between two consecutive updates to a single variable. A trusted auditor can employ this mechanism to ensure the long term compliance of a data structure, however, this technique is not suitable to be used by each client when it performs a read from the data structure.

**Proposition 3.3.1.** There is no data structure $D$ for which one can determine that it provides/violates the interval guarantee by only performing $O(1)$ verification operations.

**Intuition.** If such a data structure exists, verification cannot be based on examining snapshots. There can be an unlimited number of snapshots between two consecutive updates to a given variable. Inspecting a constant size of them will have to ensure that no illegal operation has taken place in the remaining unbounded number of snapshots. The other possible alternative is to make use of the information contained in each snapshot authenticator and try to detect inconsistencies at the time new snapshots are created. However, there are two main problems with this approach: (1) the size of an authenticator is logarithmic, while a snapshot can contain an unlimited number of elements and (2) $E$ can introduce arbitrary intermediary snapshots making it hard to audit each two consecutive authenticators.

### 3.4 Related Work

A number of fundamental techniques exist to provide various forms of accountability in different contexts. Some systems organize digitally signed data [50, 79] or require digitally signed communication [81]. Storage systems often reference data using cryptographic hashes to ensure tamper-evidence [101]. State digests find application as proof of authenticity in file systems [49], applications running on untrusted environments [77], and time-stamping services [25]. Long term historic state trails help establish provable causality in distributed systems [80]. Our state-based approach generalizes many of these techniques into toolkit that can potentially enable the construction of a range of accountable service.

KASTS [79] is an archival storage system that enables the long time storage of digitally signed documents. One of its building blocks, KAS, maintains a secure archive of public key certificates. KAS provides guarantees for provable correctness of its operations and is the main inspiration for our work. The framework proposed in this thesis build upon and extends the techniques used in KAS.
Authenticated data structures [7, 24, 79, 84, 88] are at the core of the implementation of the state-based approach and we examine them in details in Chapter 4.

Secure logging and auditing can also be used to preserve a tamper-evident record of events [109, 110]. In addition to maintaining a tamper-evident record of service events, we address the question of representing service state to make state management operations verifiable. The key challenges of logging are how to ensure all relevant events are logged, where to store the logs, and how to maintain their integrity. Our approach provides a general methodology to address these challenges.

The BAR model [3] addresses the problem of building cooperative peer-to-peer services across multiple administrative domains. This is a Byzantine Fault Tolerance approach optimized for the presence of rational peers. Since the underlying primitives depend on voting and replication, BAR-based services are vulnerable to collusion. However, BAR does not maintain history and does not use auditing to deal with problems that may arise if peers collude. Our approach shares similar goals with BAR and is complementary. We address the problem of building an accountable service within the control of a single authority that is accessed by multiple clients, who may reside in different administrative domains. In this context, the service implements all logic, and voting and replication are not sufficient. Our methodology forces the service to preserve a faithful record of its history to enable auditing in the future. BAR-based services may be able to enforce performance guarantees, something we currently do not support, although accusations are not provable outside the system.

PeerReview [50] extends parts of our approach to the context of a peer-to-peer systems. Unlike our methodology, which advocates a new approach to building applications, PeerReview focuses on making existing applications accountable by interposing on their communication. Each node in the system is periodically verified by a set of other nodes from the system (witnesses), at least one of which is assumed to be correct. Each witness replays the messages received by the node being verified and compares the result with the responses of the verified node. PeerReview uses an approach similar to the methodology described in this chapter to ensure the integrity and consistency of messages sent and received by a given node.

SUNDR [75, 81] and Plutus [65] are two recent network storage systems that defend against attacks mounted by a faulty storage server. We compare our approach to these systems in Section 5.4.
3.5 Summary

This chapter presented the core of the state-based approach to building accountable network services. We described the abstract service model that underlies the approach and the range of threats our methodology is intended to protect against. We also discussed the relationship of our approach to prior research. In the next chapters we present an implementation of the methodology and the results of several experiments intended to evaluate the costs and benefits of the approach.
Chapter 4

CATS Toolkit

This chapter presents the CATS toolkit for building accountable network services. The toolkit implements the state-based methodology described in Chapter 3 in a set of reusable components that can be leveraged to integrate accountability into network services. We present the challenges of implementing the state-based approach and describe a set of experiments intended to demonstrate the cost and performance implications of our approach.

4.1 Overview

The CATS toolkit allows a network service to organize and manage its internal state accountably. The toolkit enables a service to demonstrate that the operations that it performs on its internal state comply with the sequence of requests that it processes on behalf of its clients. In particular, the clients of the service can receive a concise proof that any data returned by the service is derived from its internal state, and that any updates to the service’s internal state are visible to other clients. The toolkit supports the full range of auditing and challenges described in Chapter 3 and provides the means to ensure that a service and its clients behave according to their specifications.

The primary component of CATS is a data structure that enables a service to index its internal state and to compute efficiently a digest over its entire internal state. As a service’s internal state evolves over time, the data structure makes it possible for the service to maintain multiple snapshots of its state, so that it can recreate its state as it existed at a given point in time. The data structure also makes it possible for the service to prove efficiently that a given state variable is or is not present in its internal state. These features enable a service to implement and support the state-based approach to building accountable services. Our implementation of this set of essential functionality in the form of a reusable toolkit effectively reduces the complexity of applying the state-based approach.
4.2 Data Structures

We considered two ways to compute a compact digest over a dynamic set of elements and demonstrate that an element with a given key and value is or is not part of the set: cryptographic accumulators [17, 56, 94, 106] and authenticated dictionaries [7, 24, 79, 84, 88].

4.2.1 Cryptographic Accumulators

Cryptographic accumulators [17] can be used to compute a digest over a set of elements and can also generate proofs of inclusion and exclusion. Such schemes offer constant size membership proofs. However, the proof update cost may be polynomial [56]. Often these schemes rely on trapdoor information, known to a coalition of trusted parties [94]. It is possible to avoid the trapdoor requirement [106], however, the size of the proof, although constant, is quite significant for most practical data set sizes. Moreover, exclusion proofs generated using cryptographic accumulators have the size of the whole data set. Thus, while cryptographic accumulators can be used to implement the state-based approach, their use in this context is unlikely to be practical.

4.2.2 Authenticated Dictionaries

An authenticated dictionary is an index over a set of elements with unique identifiers. The dictionary makes it possible to find efficiently an element with a given key as well as to add and remove new mappings as needed. Unlike a traditional indexing structure, which simply acknowledges update operations or returns requested elements, an authenticated dictionary can annotate each query and update response with an explicit proof. Query and update proofs demonstrate cryptographically that a requested operation is indeed performed over the data structure, i.e., updates modify the data structures, and queries return elements indexed by the structure. Importantly, proofs are compact, logarithmic in the size of the data set, and incur a modes cost (also logarithmic in the size of the data set) to generate and verify. Since proofs are based on cryptographic primitives—secure hashes and digital signatures—it is computationally infeasible to fabricate an incorrect proof.

Merkle [84] proposed the first example of a static authenticated dictionary using a simple hash tree scheme. A Merkle tree is a binary tree with data elements stored in its leaves (Figure 4.1). Each node of the tree is augmented with an authentication label—the output of a pseudo one-way,
**collision-resistant** hash function applied to the contents on the node’s sub-tree. More specifically, the label of a leaf node is equal to the hash of its data, and the label of an internal node is equal to the hash of the concatenation of the labels of its left and right children. The recursive Merkle hash labels the root of the tree with an **authenticator** covering the entire tree—it's value effectively represents a hash computed over every element.

![Merkle Tree Diagram](image)

**Figure 4.1**: The leaves of a Merkle tree organize the elements of a given set. Each internal tree node contains an authentication label computed over its subtree. The tree can construct a proof to demonstrate the existence of a given element. For example, the proof for element 25 includes the labels stored at the shaded nodes. Proofs of non-existence, however, are equal to the whole tree.

Merkle’s simple scheme makes it possible to construct a concise proof for the existence of an element with a known hash in a tree with a known root authenticator. The proof consists of the authentication labels of the siblings of all nodes along the path from the tree root to the leaf containing the element. Any actor can verify the proof by computing the hash for elements along the path from the leaf to the root, and comparing the computed root label to the known authenticator. It is computationally infeasible to fabricate a set of sibling labels that combine with the element hash to yield the known root authenticator.

Naor and Nissim extended Merkle’s original scheme to collections of **dynamically** evolving data, and showed how to update authentication information incrementally [88]. Buldas et al. [24] replaced the original binary tree with a binary search tree for efficient indexing, and enforced a sorted order. The resulting **authenticated search tree** (Figure 4.2) can construct efficient proofs that an element with a given hash is not present in the tree.

Anagnostopous et al. [7] introduced the idea of a Persistent Authenticated Dictionary (PAD). A PAD extends the authenticated search tree of Buldas et al. [24] to preserve a sequence of authenticated snapshots of the data structure taken after each update. A PAD can construct membership proofs for the state of the structure at any particular snapshot. The PAD prevents incorrect addition and removal of snapshots by chaining the root labels of the snapshots.
Maniatis [78, 79] describes an extension of the PAD design in the context of a certificate archive service. It uses secondary storage for large or long-lived collections, and shows how to reduce overhead by taking snapshots periodically with a configurable round size. This persistent authenticated search tree (PAST) structure is the starting point for our work. We discuss our relationship to this body of work in Section 4.6.

4.3 Design Issues

A number of organizational issues impact the performance of authenticated dictionaries. In this subsection we describe the cost dimensions, the way they impact performance, and the specific choices we made in our design.

4.3.1 Data Layout

Data placement impacts the performance of authenticated search trees. Search trees provide two options for storing data. Node-oriented organizations store data at all nodes, while leaf-oriented designs store data only at the leaves. These two choices impact the space requirement, size of membership proof, and authentication cost.

Leaf-oriented implementations are easier to remove data from and typically offer efficient sequential data scans that can be performed by traversing only leaf tree nodes. However, these designs are more wasteful in their using the internal tree nodes only as an index. For example, a binary search tree with data stored at the leaves has twice as many tree nodes than an equivalent binary search tree storing data at all nodes. As tree degree increases, the effect of data placement on tree size decreases as the fraction of tree leaves relative to all tree nodes increases. Extarnal memory search
trees, such as a B-Tree (Section 4.3.3) use fixed size disk blocks to store keys and data and tend to be leaf-oriented, since this approach maximizes the number of keys stored in a tree block, and hence reduces the tree's height, and the I/O cost.

\[ e = H(\|5\| \|5\|) \]

**Figure 4.3:** Data layout affects authentication label computation. Depending on data layout, label computation takes a different number of arguments: leaf-oriented designs use only the child labels and the internal key, while node-oriented design require, in addition, the hash of the data stored at each internal tree node.

Authenticated search trees are affected by data layout in yet another way. The computation of authentication labels depends on where data are located. With data at leaves, the computation of an internal node label requires the split keys and the labels of its children. When internal nodes also store data, this computation also requires the hash of each data value stored at the tree node (Figure 4.3). Therefore, node-oriented designs may require more data to compute a single authentication label, and consequently may have larger membership proof components. At the same time, a membership proof for a node-oriented design will likely contain less components relative to the leaf-oriented equivalent (different tree heights), which could potentially offset the effect of increased component size.

To quantify the difference in membership proof sizes, we simulated an authenticated binary search tree organized as a Red-Black Tree [13]. Each simulation inserts 2,000,000 unique randomly generated keys, and we observe the resulting membership proof size. Note that we exclude the actual data associated with a given key from the final proof size: we only use a constant hash for each data item. In this particular scenario each data value has a 4 byte key and the size of an authentication label is 20 bytes. As a result of this organization the data needed to compute an authentication label are respectively 44 (leaf-oriented) and 64 (node-oriented) bytes. Figure 4.4 shows the averaged results with one standard deviation obtained from 100 simulations. All in all, data placement affects
the size of a membership proof, with data at leaves implementations having smaller membership proof size. However, since membership proofs are generally small, this difference is insignificant. The time needed to generate a membership proof using either implementation is approximately the same. Note that as tree degree increases, the advantage of the leaf-oriented approach becomes more pronounced.

![Diagram](image.png)

**Figure 4.4**: Red-black tree average proof size depending on data layout. The leaf-oriented design results in smaller proofs.

Our calculations so far ignored the actual values associated with keys. In a real usage scenario, to verify the result of a query or update operation one may also want to obtain all referenced values, compute their hashes and verify that they are indeed the ones reported by the proof. Leaf-oriented designs ensure that each proof references exactly one data item, while node-oriented designs require at least one data item for each tree level. Therefore, leaf-oriented designs reveal less information about the contents of the data structure. Based on the consideration’s presented here and in the following subsections, our design adopts the leaf-oriented approach.

### 4.3.2 Tree Degree

The degree of a tree node defines the maximum number of children that the node can have. For each child, the tree node maintains a pointer and a split key obeying the rules of the specific algorithm. Higher node degree is used primarily by external memory trees since it helps amortize the cost of external memory access and results in a tree of smaller height. Similarly to data location, tree degree affects authenticated search trees in a unique way.
It is possible to authenticate any search tree simply by allocating extra space within each node to store its authentication label. A naive application of the same idea to a high-degree tree such as a B-Tree [12], however, can have an adverse impact on authentication performance. The high degree used by a B-Tree increases the size of the sibling set needed to compute an authentication label: a membership proof for a B-Tree storing 1,000,000 four-byte identifiers with out-degree of 100 is more than 7 times larger than a proof generated by a binary tree storing the same data.

![Figure 4.5](image)

**Figure 4.5.** Computing an authentication label. Each computation takes as input one child label component, and \((k - 1)\) label and split key components.

We can describe the computation of a tree node label as having an input consisting of a number of components. For a tree of degree \(k\), label computation takes one child label only component, and \((k - 1)\) child label and split key components (Figure 4.5). Therefore, the larger the tree degree, the larger the number of inputs necessary for the computation of a single authentication label. Consequently, label computation time increases with tree degree. The increase in input size could be compensated by the smaller tree height that is due to the larger tree degree. It turns out that the effect of the smaller tree height is not sufficient to offset the effect of the increased number of child label and split key components.

**Proposition 4.3.1.** A binary search tree provides membership proofs of minimal size.

**Proof.** We define the size of a membership proof to be equal to the number of child label and split key components needed to recompute the root label of the tree starting from the label of the data the proof refers to. Let \(T\) be a perfectly balanced tree of degree \(k\) storing a total of \(N\) values. As described earlier, label computation requires one label only component and \((k-1)\) label and split key components. During proof verification the label-only component is initially provided by the hash of the returned value and propagates recursively. Therefore, omitting the label of the returned data, a membership proof will consist of \((k-1)\lceil \log_k N \rceil\) proof components. Since the size of a component is
independent on the tree’s degree, we want to find a $k$ that minimizes the above expression. Without loss of generality, we can assume that $N$ is a power of $k$. Let $f(k) = (k - 1) \log_k N$, where $k \geq 2$.

We have:

$$f'(k) = \log_k N - \frac{k - 1}{k \ln k} \log_k N$$

$$= \log_k N (1 - \frac{k - 1}{k \ln k})$$

$$= \log_k N(\frac{k \ln k - k + 1}{k \ln k})$$

We solve:

$$f'(k) = 0 \Rightarrow k \ln k - k + 1 = 0 \Rightarrow k = 1$$

For $k = 2$, $f'(k) = 2 \ln 2 - 1 = 0.39 > 0$. Therefore $f(k)$ is an increasing function for $k \geq 1$. $\Rightarrow k = 2$ minimizes the number of proof components and hence the size of the membership proof. $\square$

4.3.3 Data Size

The versatility of a toolkit for building large scale services depends on its ability to handle large volumes of data, which rarely fit in the available main memory. External memory can help address the size limitation, however, the several orders of magnitude difference in access time may significantly impact performance. An external memory design should be tuned to minimize the number of I/O operations, increase locality, and amortize the cost of a disc access to a number of operations.

B-Trees and their variants offer an efficient mechanism to build external memory search trees with optimal I/O performance [12]. B-Trees achieve their optimality by using a high out-degree and a carefully designed balancing algorithm. A B-Tree node occupies a number of consecutive disk blocks and maintains a split key and a pointer to each of its children. When the available space within a block is exhausted, the tree is balanced using block splits: half of the block’s keys and pointers are migrated to a new block. Split operations may propagate recursively and increase the height of the tree. This organization guarantees that a tree of degree $B$ indexing $N$ elements has height $O(\log_B N)$.

The result of Proposition 4.3.1 exposes a dilemma for building scalable authenticated data structures: indexing large collections of data requires the use of an I/O efficient underlying data structure,
e.g., a B-Tree, however, it will be costly to compute digests and to generate, transmit, and verify proofs in such a data structure. Alternatively, one can optimize for smaller proof sizes, but such an approach is likely to result in poor I/O performance. We address this challenge with a hybrid approach consisting of a balanced binary tree mapped to the nodes of a B-Tree for efficient disk storage (Figure 4.6). While this approach ensures the optimality of search operations, updates can trigger a large number of I/O operations to balance the binary tree. We relax the requirement of global binary tree balance and trade it for improved I/O performance: only the part of the tree that is stored within a single B-Tree block should be balanced. This restriction may increase the height of the binary tree but offers an acceptable tradeoff between membership proof size and I/O efficiency. This solution is similar to the one of Maniatis [79]. Proposition 4.3.1 shows that his choice of a binary tree is indeed optimal.

![Figure 4.6: Mapping a binary search tree to disk blocks for efficient storage. The idea is to use a large degree external memory tree, such as a btree, and to organize the keys within each block into a binary search tree. This organization achieves good I/O performance and produces membership proofs of reasonable size.](image)

### 4.3.4 Persistence

Persistent search trees avoid data overwrite during updates and retain multiple versions of each stored data item, effectively preserving multiple *snapshots* of the data structure at different points in time. Data structure snapshots enables important application features such as historic analysis and data recovery. These features are of great interest when dealing with potentially malicious environments and system participants because they empower some of the basic mechanisms needed to provide accountability. However, this added benefit comes at the expense of increased space requirements.

The number of snapshots and the granularity at which they are maintained is an important design
question with profound impact on performance and accountability. To facilitate the discussion in this subsection, we refer to the period between two consecutive snapshots as an \textit{epoch}, and \textit{epoch length} measures the number of updates within an epoch.

The increased space demand of persistence can be reduced by using \textit{copy-on-write} to share data structure elements among multiple snapshots: to update the data structure one makes a copy of all elements about to be affected by the update and applies the update to the copies rather than the originals; all other elements remain the same. Driscoll et al. [45] show that a variant of this technique, when applied to certain data structures can provide \textit{constant} amortized space overhead per operation. Constant space overhead, however, is unattainable for authenticated search trees because the update of a single tree node (or the introduction of a new one) invalidates the authentication labels of all nodes along the path from the affected node to the tree root. Therefore, each authenticated search tree update triggers modifications to a logarithmic number of tree nodes and consequently causes \textit{logarithmic} space overhead.

Using coarser persistence granularity, longer epoch length, helps decrease the space overhead due to persistence. Instead of applying \textit{copy-on-write} for each update operation, it is possible to execute a sequence of updates to the same view of the data structure. All these updates will be considered as having taken place in the same data structure epoch. This approach avoids making copies of already duplicated tree nodes, however only one update to a given key can be correctly processed within a single epoch, which may reduce the concurrency of operations depending on the write conflict rate. The epoch size, therefore, poses a trade-off between space overhead, and precision of historic analysis and data recovery.

\textbf{Proposition 4.3.2.} The expected number of copied/duplicated tree nodes for an \textit{m-way} persistent search tree with a total of \( n \) nodes during an epoch of \( k \) updates is equal to:

\[
\sum_{i=0}^{\log_m n} m^i (1 - (1 - \frac{1}{m}))^k
\]

\textit{Proof.} Let \( T \) be an \textit{m-way} tree with a total of \( n \) nodes. Let’s assume that \( T \) is complete. Performing \( k \) update operations will require the duplication of some tree nodes. Let \( X \) be a random variable equal to the number of duplicated tree nodes during an epoch of size \( k \). Let \( p_i \) be the probability that a single update operation will affect a given node on level \( i \) of the tree. The probability that a

65
Using the above expression, we can compute the expected tree size for a given epoch size at any time instance. To help reason about space overhead, we use the notion of relative space overhead or stretch factor. The stretch factor is the ratio of the size of the persistent tree with a given epoch length to the size of the corresponding ephemeral tree. Thus, the stretch factor is a measure of the additional space required to provide persistence. Figure 4.7 shows the results for epochs of different length. We observe that the overhead is logarithmic, with larger epoch length having smaller constant factors, due to the increased level of sharing among a larger number of operations.

Figure 4.7: Expected stretch factor. The stretch factor of a persistent dictionary is logarithmic and depends on the epoch length. The smaller the epoch, the larger the stretch factor. Larger epoch size at any time instance.

The above expression shows that for a complete m-way tree, $n_i = m^i$, and $p_i = \frac{1}{i}$ (assuming random sequence of update operations). Therefore:

$$E[X] = \sum_{i=0}^{\log_m n} n_i (1 - (1 - \frac{1}{i})^k)$$
Using longer epochs reduces a constant fraction from this overhead by amortizing the cost of tree maintenance (including regenerating the authenticators) across multiple update operations. For example, since labels close to the root of the tree are affected by every update operation, delaying label regeneration to the end of the epoch can ensure that each label is computed exactly once (Section 4.3.6). Processing more updates within a single epoch also decreases space utilization and the disk write rate. Importantly, longer epoch sizes reduce the number of state digests that must be published to the external medium.

However, there are limits to the practical epoch length. Longer epochs include more writes and modify more disk blocks; the number of dirty blocks for a write set also grows logarithmically with the size of the tree. If the set of dirty blocks does not fit within the I/O cache, then the system may incur an additional I/O cost to regenerate the labels at the end of the epoch (Section 4.3.6). Longer epochs also increase the time to create a state digest and acknowledge the completion of an operation. Epoch length is an adjustable parameter in our design, in order to reflect these complexities.

4.3.5 Concurrency

Any data structure intended for use in high performance computing environments should be able to support multiple concurrent readers and writers. Concurrent data access requires synchronization mechanism to avoid corrupting data or deadlocks. Persistent data structures make the problem of concurrency somewhat easier—reads from earlier epochs do not require synchronization. However, multiple operations in the currently active epoch still require mechanisms to avoid corrupting the data structure.

Path locking is one way to deal with concurrency in a tree data structure. Using this technique one obtains a lock on every node on the path from the root to the location where the update needs to take place. Since an update operation will affect at most the tree nodes along the path from the root to the update location, path locking is always guaranteed to be correct. However this technique is too restrictive because each update operation will try to lock the root of the tree, which will force update operations to execute sequentially. As a result path locking offers primarily integrity protection rather than performance enhancement.
Lehman and Yao introduced a powerful synchronization technique based on lock coupling [72]. Their solution uses reader-writer locks [18] to reduce contention—when reading a tree node one obtains a shared reader lock, similarly, before writing to a node one obtains an exclusive writer lock. To perform an update one recursively obtains and releases reader locks until reaching the tree node where the update should take place. Then the writer obtains a writer lock and makes sure that the node is still the correct place for the update operation. If in the meantime some other thread invalidated the tree node, the writer lock is released and the client continues searching for the proper location. Once the client has the writer lock over the correct block, it can perform the update. Should the update require propagation up the tree path, the client obtains a writer lock on the parent node and then releases the writer lock on the child.

Lehman and Yao also present an addition to the above technique when dealing for trees that grow bottom up by using node splits instead of rotation: $(\alpha, \beta)$-trees, B-Trees, etc. The idea is to introduce one more key and pointer element to each tree node. When performing a split this key is set to the split key and the pointer refers to the new node. These additions come to use when one is moving down or up the tree hierarchy: if a locked node does not contain the key it is expected to, one follows the chain of extra pointers and uses the split keys to navigate to the correct tree node.

Lock coupling always obtains writer locks in a bottom-up fashion and is guaranteed to be correct. It also offers maximum concurrency: at any time a thread holds at most two writer locks. However, lock coupling adds significant complexity and is prone to errors that are hard to detect. All in all, the choice of a synchronization mechanism depends primarily on whether the search tree is stored in main memory, or resides on an external storage medium. The benefit of locking with main memory implementations is negligible and even negative. Yet, when external memory is involved, the benefit of using multiple threads far outweighs the complexity and hardness of implementation. Our external memory design used Lehman and Yao’s algorithm.

4.3.6 Finalization

When an epoch consists of more than one update operation there are two alternatives for computing authentication labels: immediate label computation as a part of each update operation and delayed label computation at the end of each epoch (finalization). Immediate computation has the advantage
that all necessary labels are computed at the time an update operation completes. However, these labels cannot be used for authentication because other updates during the same epoch may invalidate (and recompute) some of them (e.g., each update operation within a single epoch invalidates the label of the root node). Finalization avoids label computation and allows update operations to complete faster. The computation of labels at the end of each epoch, though, makes it harder to satisfy new update operations while finalization is taking place.

Immediate label computation works best when the underlying data structure uses path-locking for concurrency: to compute the labels invalidated during an update one “bubbles-up” towards the root of the tree, obtaining and releasing locks on the siblings of each node along the path. This approach to label computation does not work directly with Lehman and Yao’s lock coupling approach since it requires that a write holds both the write lock on a parent node and read lock on a child node. This condition will eventually result in a deadlock. It is possible to work around this limitation by making sure that an update does not need data from sibling blocks, i.e., in addition to split keys, each node also stores the authentication label of the corresponding subtree. While our design can be adapted to support either way of label computation, the current external memory prototype uses finalization to avoid recomputing labels and to reduce the implementation difficulty of an already complex data structure. We also offer a single-threaded implementation, which uses immediate label computation.

Finalization performance and main memory block cache capacity are directly correlated. During finalization the data structure must access all blocks that have been affected during the current epoch. If blocks do not fit in the main memory block cache, finalization is likely to incur additional I/O operations. This relationship provides an additional constraint on the epoch length.

4.3.7 Recovery

CATS operations may be interrupted by a system failure: power, disk, or other type of outage. While it is generally desirable for a data structure of this type to recover successfully when failures occur, CATS usage for building accountable services poses additional requirements on recovery. If a service using CATS commits to an action, which has uncommitted data in a CATS data structure, crash failures resulting in data loss would actually make the service accountable for the lost data.
An important consequence of this result is that a service cannot respond to a client if one or more CATS update operations pertaining to that client’s request are still in progress.

Since it may be impractical to delay a response until all relevant state is committed to permanent storage, or it may be too expensive to commit frequently, CATS uses its recovery mechanisms to ensure that sufficient information about a pending operation is committed to disk, before acknowledging the operation. If the pending operation completes normally, no additional work is necessary. Upon failure, however, the additional information is used to restart and complete the pending operation. This feature is currently supported using write-ahead-logging.

4.4 Implementation

We implemented CATS using C# and the Microsoft .NET framework 1.1. The current toolkit prototype consists of approximately 20,000 lines of code and offers both a main-memory and an external memory implementation. The main memory implementation is more efficient, but is limited in the amount of data that it can index and the number of snapshots that it can maintain. While the main memory component has limited application, it offers a baseline for the external memory prototype. The experience with the main memory version helped us understand the intricacies of authenticated data structures and prepared us for the challenges of external memory.

4.4.1 Main Memory

The main memory part of the toolkit provides a library implementing different tree algorithms in multiple variants. The library is designed to work with most tree algorithms and currently provides support for Read-Black trees [13] and AVL trees [1]. The tree variants supported by the library are: ephemeral, persistent, authenticated, and persistent authenticated. Each variant builds on top of the previous and new tree logic algorithms can easily be plugged-in. The resulting trees can be serialized/deserialized to/from and external memory medium, but these operations cannot be used as a replacement of a data structure explicitly designed for external memory.
Figure 4.8: The CATS toolkit consists of several layers. At the bottom, an I/O abstraction layer provides access to the underlying storage medium. A block cache resides above the I/O layer and provides caching and consistent access to disk blocks. The dictionary module uses the block cache to store and organize its contents. The write-ahead logger helps record state about each dictionary update to enable recovery after a crash.

4.4.2 External Memory

The primary component of CATS is an external memory persistent authenticated search tree (Figure 4.8). Unlike the main memory component, this one is explicitly designed to handle large volumes of data and snapshots. The persistent authenticated search tree is based on a B-Tree with each of its nodes organized as a persistent balanced binary search tree (see 4.3.3). Each B-Tree block is identified by a unique identifier and consists of a region for metadata and a region for storing the nodes of the embedded binary tree. Each binary tree node has a constant size and contains its key, authentication label, and the location of its left and right children. A location identifier is a concatenation of a B-Tree block identifier and an offset within the block.

A B-Tree block has fixed configurable size (64KB is the default) and can accommodate a constant number of binary tree nodes (leaf tree blocks contain a slightly larger number of binary tree nodes). Each block maintains in its metadata region an allocation bitmap to keep track of which binary tree nodes are currently in use. The metadata section also tracks the number of binary tree nodes, the location of the current root of the binary tree, and other state useful for persistence and synchronization, e.g., a link pointer to the tree block’s right sibling, as per Lehman and Yao’s algorithm, a log sequence number of the last update operation to the block, a reader/writer lock, and a reference count, etc.

An intermediary block cache module controls access to storage, buffering, and synchronization. An asynchronous background thread monitors the block cache and drains the cache to keep the number of blocks in use between configured high-water and low-water marks. The block cache man-
ager provides interfaces to pin and unpin blocks in memory, as well as to lock blocks in different lock modes (shared and exclusive). The current implementation does not perform deadlock detection.

The block cache is layered on top of a generic storage layer, which abstracts the actual storage implementation. This layer exports a block-level abstraction with a unified interface to pluggable underlying storage implementations. The current prototype stores blocks in a file on the host file system, so it can easily use parallel or replicated storage supported by the file system. Support for other storage media can, thus, be easily integrated.

Tree operations proceed in epochs. The epoch length is not fixed and can be tailored to application requirements, e.g., two snapshots can have different epoch lengths. Cats supports two main operations: query and update. Queries can be performed over any snapshot, however, they return a membership proof only when applied to a committed snapshot. Updates are always applied to the current snapshot and do not return a membership proof immediately. Optionally, Cats can block an update and return a membership proof after the current snapshot has been committed. The implementation offers two modes of computing authentication labels: immediate, as part of each update, and delayed, at the end of a round. When using delayed computation, updates mark each affected binary tree node: at the end of an epoch the algorithm computes the authentication labels in a sweep up from the bottom of the tree. We use root chaining to prevent the inclusion/removal of epochs.

All tree operations use write-ahead-logging to ensure successful crash recovery without data loss. The write-ahead-log can be configured to flush log records immediately (default) or to buffer them in memory and write them periodically to disk (e.g., assuming the host server has access to NVRAM). After a crash, Cats scans its log file searching for the most recently committed snapshot and verifies if any of the operations after it need to be restarted. Once recovery is complete, the current snapshot is committed to disk, and the data structure resumes normal operation.

Persistence in Cats builds on the approach of Becker et al. [14]. B-Tree blocks are engineered to have 20% more space for binary tree nodes than their fill factor. This approach makes it possible to co-locate binary tree nodes from different epochs in the same block. Each binary tree node contains a field that denotes the epoch to which it belongs. During an update the algorithm makes a copy of all binary tree nodes along the update path that do not belong to the current epoch and applies
the update to the copies, not the originals. The metadata region of the B-Tree block also maintains a variable that counts the number of binary tree nodes that are reachable from the current binary tree node and belong to the current epoch.

If there is not enough space to perform an update in a B-Tree block, the implementation rolls back all changes (any copies made as part of the update operation) and performs a split operation. There are two types of split operations: node splits and version splits. A node split moves approximately half of a block's binary tree nodes to a newly allocated B-Tree block. The current implementation moves the right subtree of the embedded binary tree. A version split migrates all nodes that are reachable from the current binary tree root to a newly allocated block and marks them as belonging to the current epoch. Node split is used when all reachable nodes from the root of the binary tree belong to the same epoch, and a version split otherwise.

The current prototype does not offer a delete operation: it maintains all values for a given element. While this approach may have a potentially high storage cost, it is designed to enable fine-grain historic analysis and support reasoning about accountability over any interval of time. In practice, however, it may be necessary to remove old data. Since data removal may be leveraged to conceal a misbehavior in the past, it must be carefully controlled and regulated. One possible approach is to define “a statute of limitations”, which will impose a time limit on the validity of old data, and will enable services to discard data, which is no longer relevant. We leave the details of this mechanism to future work.

Finally, the toolkit exports interfaces to customize various operating parameters to the needs of applications. These parameters include: the type of search key, the hash function for computing authentication labels, etc. CATS also supports pluggable algorithms to balance the binary tree within each B-Tree block, which affects update cost and proof size. Currently CATS offers a Red-Black tree [13, 58], AVL [1] and a weight-balanced [83] trees can be added with a minimal effort.

4.5 Evaluation

In this section we present an evaluation study of our prototype. Our goal is to understand how the core data structure required by the state-based approach performs. In particular, we study membership proofs (size and time to verify), and the impact of epoch length on update and storage
costs. We also compare the behavior of the external memory implementation to the one of the main memory prototype.

4.5.1 Methodology

We run all tests on IBM x335 servers running Windows 2003 Server Standard Edition operating system. Each machine has 2 Pentium IV Xeon processors running at 2.8GHz, 2GB of RAM, 1Gb Ethernet, and two IBM SCSI hard disks with rotation speed of 10,000 RPM, average latency and seek time of 3 ms and 4.7 ms respectively.

We use SHA-1 to compute authentication labels. The key size we use is 16 bytes, and object keys are assigned randomly in the identifier space. All tests are single-threaded. We create new epoch at fixed controlled rates, depending on the experiment. External memory tests use a sequence of 3 million keys, while main-memory tests range from 1 to 2 million keys depending on epoch length. All tests organize keys using the Red-Black tree algorithm using a leaf-oriented layout. We report averages and one standard deviation (where appropriate) from 100 and 50 test runs for main memory and external memory experiments respectively.

We store the external tree in blocks of size 64KB. The tree block cache has capacity 2000 blocks and is chosen so that the tree does not fit in main memory. For storage medium we use files residing on the local NTFS file system. The files are accessed using the standard .NET I/O API. For improved performance we place the write-ahead log and the index on separate disks. We commit dirty tree blocks to disk every 2 seconds.

4.5.2 Main Memory

Cost of hashing. The first test we performed evaluated the cost of hashing data on our experimental testbed. We compared the cost of two popular hash functions: SHA-1 [89] and MD5 [103]. Figure 4.9 illustrates the result. Each data point represents an average over 1000 iterations with one standard deviation. As expected, for relatively small data sizes both hash functions take approximately the same time. However, as data size increases SHA-1 becomes more expensive.

Membership proofs. Generating and verifying membership proofs is an essential authenticated search tree operation. We simulated the generation and verification of membership proofs with
different number of components varying the hash function being used. Figure 4.10 shows the proof size and Figure 4.11 shows the time to verify a membership proof as a function of the number of membership proof components. Since the number of membership proof components is logarithmic relative to the number of unique elements in the data set, for most practical purposes membership proofs are expected to occupy less than 2KB and take less than 1ms to verify.

Figure 4.12 also reports the average path length observed in the experiments with our main memory implementation, while Figure 4.13 shows the corresponding membership proof size and Figure 4.14 shows the corresponding verification times.

**Space overhead.** Figure 4.15 shows the relative space overhead observed during our tests. As expected, increasing the granularity of persistence increases the space demands of the data structure, with epoch size 100 taking as much as 10 times more space than the corresponding ephemeral tree. Overall, these results agree with the analytical model derived in Section 4.3.4.

**Basic tree operations** Figure 4.16 shows the time to perform an insert operation without computing authentication labels. This is the time needed by the Red-Black tree algorithm to locate the insertion point and maintain the structure of the tree. We observe that as epoch size decreases, the time needed to perform an insert operation increases. This result is due to the fact, that smaller epoch sizes trigger the creation of more node copies. The periodic fluctuation at the beginning of each epoch is due to the increased number of node copy operations. As the epoch progresses, less nodes need to be copied, and the time to locate the insertion point stabilizes. The results of epoch
Figure 4.10: Membership proof size as a function of the number of proof components (SHA-1, data at leaves). Single proof components have constant size. The number of proof components is logarithmic relative to the number of elements of the data set. For most practical purposes, proofs are less than 2KB long.

Figure 4.11: Membership proof verification time as a function of the number of proof components. For most practical purposes, proof verification takes less than 1ms. Note that the times on the graph do not include the time spent to obtain/verify snapshot authenticators and validate signing certificates.
**Figure 4.12:** Average path length for a main memory search tree.

**Figure 4.13:** Membership proof size for a main memory persistent authenticated search tree. As expected, proofs are less than 2K bytes large.

**Figure 4.14:** Membership proof verification time for a main memory persistent authenticated search tree. As expected, verification takes less than 1ms.
Figure 4.15: Main memory persistent tree stretch factor. As epoch size decreases the stretch factor increases, making the resulting data structure larger.

Figure 4.16: Main memory tree insertion time without computing authentication labels. Smaller epochs take longer to perform a tree update since a smaller epoch requires making copies of a larger number of nodes during each update operation.

of size $10^6$ closely approximate the cost of query operations. Query operations are not affected by epoch length, and depend only on the height of the tree.

The cost of authentication. Label computation is an essential part of an authenticated search tree update operation. In Section 4.3.6 we described two approaches for computing authentication labels: immediate and delayed (finalization). We implemented both mechanisms and Figure 4.17 shows the time needed to perform an insert operation performing immediate label computation. Comparing these results to the ones presented on Figure 4.16, we can calculate the cost of computing authentication labels. As tree height stabilizes, authentication label computation takes between 420
Figure 4.17: Main memory insertion time with immediate authentication label computation. The cost of hashing is independent on epoch length and is the dominant cost of an update operation.

and 430 microseconds.

Finalization provides an alternative to immediate label computation. To compute the amortized time to perform an insert operation we charge the time needed to finalize an epoch to all insert operations that take place during the epoch. Figure 4.18 shows the resulting amortized insertion time. The gain of finalization is striking for epochs of larger sizes, with the cost of overall insertion time being 25% of the cost of the corresponding immediate operation. As the epoch size decreases, the benefit of finalization decreases and for an epoch size of 100, it is relatively close to the corresponding non-finalization time. We explain this phenomenon, with the fact that using smaller epochs triggers smaller number of label re-computations and consequently finalization can offer limited improvement.

Summary In this section we presented an evaluation of the main memory component of Cats. The evaluation of our prototype shows that membership proof verification is efficient and takes less than 0.5 milliseconds. Insertion operations can be as costly as 450 microseconds per operation. The way authentication labels are computed plays an important part; delaying label computation can reduce the cost of an insertion operation by up to 75%. Queries and membership proof generation operations are fast: approximately 20 microseconds. The choice of an epoch size is crucial for the performance of the data structures. Smaller epoch sizes result in larger trees and costlier insert and update operations.
Figure 4.18: Main memory insertion time using finalization. Delaying the computation of authentication labels until the end of an epoch can significantly reduce the cost of authentication for large epochs. As epoch length decreases, the benefit of finalization decreases.

4.5.3 External Memory

Path length and membership proofs. Figure 4.19 shows the average path length observed in our experiments. As expected, the path length is slightly larger than the corresponding main memory tree’s path length (the global embedded binary tree is not completely balanced), but it is still logarithmic and provides acceptable performance. Figure 4.20 shows the size of membership proofs and the time to verify a membership proof as a function of the number of unique keys in the index. Importantly, these metrics do not depend on epoch size. As expected, membership proof size is logarithmic relative to the unique keys in the index. For our configuration, a proof is less than 2KB and it takes less 1 millisecond to verify. These results are comparable to the results for our main memory prototype.

Space overhead. Figure 4.21 shows the stretch factor for epochs of different size. As expected the stretch factor increases as epoch length decreases since each update triggers an increasing number of node copies. The observed stretch factor is comparable to the main memory tree’s stretch factor.

Basic tree operations. Figure 4.22 shows the time to perform an insert operation without computing authentication labels as a function of the number of unique keys in the tree. Performance degrades as epoch length decreases, since the tree’s size increases and a smaller number of blocks can fit in the block cache.

Authentication cost. Figure 4.23 shows the cost of update operations when authentication
Figure 4.19: External memory tree average path length.

Figure 4.20: External memory membership proof size is logarithmic relative to the number of unique keys in the index and is on the order of 1KB. The time to verify a membership proof is on the order of 500 μseconds.

labels are computed immediately during the update operation. Similarly to our main memory prototype this mode of operation introduces between 400 and 450 μsec overhead to each update. Unlike the main memory implementation, update cost still depends on epoch length, since the working set no longer fits in main memory. In Figure 4.24 we calculate the amortized cost per update of computing authentication labels at the end of each epoch instead of as part of each update operation (finalization). As expected, the benefits are more pronounced for larger epochs.

Figure 4.25 shows the time to apply an update to the authenticated index using finalization instead of immediate label computation. We observe that the update cost is logarithmic with respect to the number of unique keys. The cost also grows as epoch length decreases. There are two
Figure 4.21: The external memory tree’s stretch factor is logarithmic relative to the epoch length. Smaller epochs have significantly higher stretch factors resulting in larger indexes.

Figure 4.22: External memory tree update time without authentication depends on the epoch size.

primary reasons to explain this behavior. First, smaller epochs amortize the cost of maintaining the index and computing authentication labels among a smaller number of updates. Second, smaller epochs produce larger indexes and cause higher write rate per update operation.

Figure 4.24 shows an unusual behavior for epoch size of 1,000,000 updates—the amortized cost increases unexpectedly. We account for this behavior by studying the number of I/O operations generated during label computation (Figure 4.26). While for the remaining epoch sizes all nodes that require label computation fit in the block cache, for this particular epoch size, the cache cannot accommodate all nodes. As a result, label computation incurs a number of cache misses which result in additional disk operations. This result shows that there is an interesting dependence between
the epoch size and cache capacity. It is possible that for some configurations increased cache misses during label computation can offset the benefit of delaying label computation.

**Summary.** The important point to take away from these experiments is that membership proofs are compact and relatively cheap to verify. Update operations in our unoptimized prototype have acceptable performance. Epoch length has direct impact on space overhead and update cost and is the primary factor that determines performance.

Our evaluation suggests that it is possible to build external memory persistent authenticated dictionaries with acceptable performance. The toolkit implementation produces membership proofs of manageable size (less than 2KB when using 16 byte keys and SHA-1). For most practical index
**Figure 4.25:** The time to apply an update to the authenticated dictionary is logarithmic relative to the number of unique keys in the index and increases as epoch length decreases.

**Figure 4.26:** Cache misses during delayed label computation can affect performance.

sizes, the verification of a proof requires less than 1 millisecond, excluding the initial data element hashing. The amount with which authentication label computation increases the latency of index operations depends on the nature in which labels are computed. Immediate label computation incurs an overhead between 400 and 450 microseconds per operation when using SHA-1 and 16 byte keys. For the same scenario, delaying label computation can reduce the overhead significantly depending on epoch size, e.g., for epoch size of 1,000,000 updates, each update operation incurs on the average 100 microseconds additional overhead. The overhead increases as epoch size decreases. The configuration of the index can impact performance as cache misses during delayed label computation trigger additional disk accesses.
4.6 Related Work

The implementation of Cats is inspired by earlier work by Maniatis [78, 79]. While the focus of Maniatis was to show that the design was suitable to build an accountable certificate store, we generalize the original design as the basis for a reusable state storage layer. In particular, our implementation is highly concurrent and recovers after failures. We also integrate interfaces to allow auditing the state store and tracking state updates through time. Our general implementation permits a detailed evaluation of the performance costs of authenticated dictionaries, which is critical to understanding when and where it is practical to use them. We analyze design issues that affect these primary costs: space requirements, proof size, access latency as a function of epoch length, and update cost.

We also explore the systems context for use of Cats stores as a basis for accountable network services. In particular, we expose the freshness problem that affects all authenticated data structures in the literature. Although an authenticated dictionary can generate a proof that a snapshot reflects the state change from an update operation, it cannot guarantee that the service does not falsely discard the update in a subsequent epoch. This is because the application controls the submission of new updates; thus a malicious service could execute unauthorized writes or replay prior authorized writes. We show how to address these problems with write timestamps and primitives for auditing by an arbitrary third party (Chapter 3).

Cats makes use of authenticated data structures to enforce undeniable and tamper-evident commitment, as well as to demonstrate inclusion in or absence from the committed set. There are two general classes of authenticated data structures. The first class is more common and makes use of a search tree. This class was introduced by Merkle [84] and a number of authors have improved on the original proposal: dynamic trees [88], enforced sorted order [24], count-certified [93], persistent [7], external memory friendly [78], and linked (resilient to snapshot removal/insertion) [25]. The second class of data structures is based on skip lists [100], and [7, 55, 78] present work in this category. Authenticated data structures are widely used in certificate revocation [23, 24, 84], timestamping [16, 24, 25] data publication [43, 41, 93], database queries [26], securing memories [19, 33, 34], file systems [29, 49, 81], databases [77], authenticating XML documents [42], authenticating distributed data [98], and archival of signed documents [79].
CATS implements the state-based approach to building accountable services and incorporates and generalizes techniques used by prior systems into a reusable toolkit that can act as a substrate for a range of accountable services. The elements of CATS have been used in many previous systems for similar goals of semantic accountability. Many systems require digitally signed communication and some maintain some form of signed action histories [50, 79]. Long-term historic state trails help establish provable causality in distributed systems [80]. Storage systems often reference data using cryptographic hashes to ensure tamper-evidence [101]. State digests are used to prove authenticity in file systems [49], applications running on untrusted environments [77], and time-stamping services [25].

Other systems have used the approaches in this section to implement application services with various accountability properties. Examples include file systems [81, 49], time-stamping services [25], and certificate management systems [79]. A state store toolkit with integrated support for accountability can significantly reduce the complexity of these services, and enable construction of a range of similar services with strong correctness guarantees.

The CATS toolkit is based on the storage abstraction of a binary search tree. The Boxwood Project [76] also provides a toolkit for building storage infrastructures from foundational search tree abstractions. Boxwood does not address the problem of accountability. In comparison, CATS offers a toolkit for building accountable services; storage services are a subset of the possible applications of the toolkit.

4.7 Summary

This chapter presented the design and implementation of the CATS toolkit. We describe the different design choices and how they impact the performance of the data structures used in the toolkit. We conduct extensive micro benchmarks to evaluate the cost of our approach. Overall, the making the data structures authenticated and persistent incurs modest costs, which are likely to be acceptable for a range of applications in which correctness and accountability are primary concern. We continue our evaluation of the approach in the next chapter in the context of a storage service built using the CATS toolkit.
Chapter 5

CATS Storage Service

In this chapter we describe the design, implementation, and evaluation of the CATS Storage Service: a simple storage service with strong accountability properties. The service is built using the state-based approach from Chapter 3 and the CATS toolkit from Chapter 4. We show how to leverage the CATS toolkit to integrate support for strong accountability into a specific network service. We also study the impact of accountability on the performance and storage costs of the service.

5.1 Overview

CATS is a simple network storage service: it enables clients to read and write a shared directory of objects maintained by a CATS server. Similar storage services are now being offered commercially, e.g., Amazon’s Simple Storage Service [6]. Such services offer storage hosted by a service provider and make it possible for users and organizations to share data.

An important challenge of offering commercial hosted storage services is to ensure the end users that the service maintains a consistent view of their data, e.g., the service only applies updates from authorized users and users can see each other’s updates. Organizations using the storage service may need access to secure audit trails to audit updates to their shared data, for example, to aid internal investigations and attribute responsibility.

CATS addresses this challenge by providing clients with the means to verify that the server executes writes correctly, and that read responses are correct given the sequence of valid writes received at the server. Crucially, strong accountability of the server also extends to the clients: a correct server can prove that its state resulted from actions taken by specific clients. Clients cannot deny or repudiate their operations on a strongly accountable server, or the impact of those operations on the shared state. The CATS network storage service is based on the generic CATS state storage toolkit (Chapter 3).

The CATS network storage service is a building block for distributed applications or services. Its purpose is to act as a safe repository of state shared among the participants (actors) of a distributed
**Figure 5.1:** Overview of a CATS-based service: the accountable storage service.

The CATS-based service is designed to ensure accountability in a system where clients interact with a shared state. The shared state guides the behavior of the clients, and updates to the shared state may affect the overall group. The clients can represent a community of users responsible for their updates.

CATS supports the core functions of a network file system. Clients can create, read, and write opaque objects identified by unique identifiers (oids). Each write request generates a new version of an object. Versions are named by monotonically increasing timestamps of the writes that created them. Reads retrieve the most recent version of the object. A client may also request a prior version of an object that was valid at a specified time. We have limited our focus to essential object storage functions: for example, in its present form, CATS does not support nested directories, symbolic names or links, renaming, partial reads or writes, truncates, or appends.

The CATS storage service supports simple static access control lists: creators of an object attach a list of identities of actors allowed to modify the object. The list is static and access rights cannot be granted and revoked once an object has been created. We revisit the subject of access control in Chapter 6.

### 5.1.1 Threat Model

The CATS storage service shares the generic threat model of accountable network services described in Chapter 3. Table 5.1 summarizes the attacks covered by our threat model and the defenses used to protect against them.
Table 5.1: Summary of attacks and defenses.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server fails to execute write or object create</td>
<td>provable detection</td>
</tr>
<tr>
<td>Client repudiates object create or write</td>
<td>provable detection</td>
</tr>
<tr>
<td>Server denies write or object create</td>
<td>provable detection</td>
</tr>
<tr>
<td>Client writes out of order</td>
<td>rejected by server</td>
</tr>
<tr>
<td>Server returns invalid read response</td>
<td>provable detection</td>
</tr>
<tr>
<td>Fraudulent or unauthorized writes</td>
<td>provable detection with simple static ACLs; relies on trusted external authorization server for richer policies</td>
</tr>
<tr>
<td>Server replays or reverts valid writes, destroys objects, reorders writes, or accepts out-of-order writes</td>
<td>verifiable detection by challenge or audit</td>
</tr>
<tr>
<td>Tampering &amp; forking, or other variants of above attacks on data integrity</td>
<td>verifiable detection by challenge or audit</td>
</tr>
<tr>
<td>Violation of privacy or read access policy</td>
<td>no defense</td>
</tr>
</tbody>
</table>

5.1.2 Toolkit-based Design

Figure 5.1 presents a high-level view of the CATS storage service and its components. The service is designed following strictly the state-based methodology described in Chapter 3. Here we provide the high-level points about the storage service design. Please consult Chapter 3 for further details.

The service and its clients communicate using the Simple Object Access Protocol (SOAP). Each message consists of a digitally signed header and an optional payload. We assume the existence of a Public Key Infrastructure (PKI), which is responsible for distributing keypairs to all involved parties, and which can be used to discover or verify the key for a given actor. We also assume that there exists an external publishing medium, to which the service publishes periodically its digitally signed state digests. The Public Key Infrastructure constitutes the trusted base. Misbehavior of any other actor can be detected and proven. The publishing medium does not have to be trusted. See Chapter 3 for details.

The storage service has simple and well-defined operations (Table 5.2), which allows us to design it as a thin layer on top of the CATS generic state store toolkit. We define the internal state of the storage service to be the collection of all objects that it stores. Since each object has a unique identifier (oid), we can leverage the CATS toolkit directly to represent the service state as an indexed collection of unique elements. With the help of the toolkit, the service computes periodically digests of its state, which it digitally signs and publishes to the external medium and also returns them
Table 5.2: Accountable storage service operations. The service accepts reads and writes to a set of named versioned objects. All client writes and all server responses are digitally signed so that actors can be held accountable for their actions. Challenges and audits provide the means for clients or third-party auditors to verify the consistency of a server’s actions relative to periodic non-repudiable digests generated by that server and visible to all actors.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(oid)</td>
<td>Returns the most recent version of the object with the specified oid. In our prototype, every read response includes a proof of correctness relative to the digest for the most recent epoch (implicit challenge).</td>
</tr>
<tr>
<td>read(oid, time)</td>
<td>Returns a signed response with the version of the specified object at given time (epoch). In our prototype, every read response includes a proof of correctness relative to the digest for the epoch (implicit challenge).</td>
</tr>
<tr>
<td>write(oid, value, current)</td>
<td>Overwrites the specified object with the given value. The server accepts the request only if at the time of the write the version stamp of the object is equal to current. Otherwise, returns the object’s version stamp.</td>
</tr>
<tr>
<td>challenge(oid, epoch)</td>
<td>Requests that the server provide a proof that the object with the given oid is or is not included in the service state, relative to the published digest for the given time (epoch). The response includes a membership proof for the object and its value, or an exclusion proof if the oid was not valid.</td>
</tr>
<tr>
<td>audit(oid, epoch, span, depth)</td>
<td>Requests that the service substantiate the write history for the object named by oid over the time interval [epoch, epoch + span]. The request is equivalent to depth randomly selected challenge requests over the interval. The depth parameter controls the degree of assurance.</td>
</tr>
</tbody>
</table>
back in its responses to clients. As a result of this organization, the storage service can leverage the toolkit mechanisms to prove to clients the correctness of read and write operations: i.e., reads are satisfied from state visible to all clients, and writes update the service state, so that other clients see the same changes.

Following the state-based methodology, the storage service processes client requests in epochs. The length of an epoch is variable. Its length depends on the write conflict rate in the client workload and has a configurable upper bound (several minutes). If a write conflict occurs in an epoch, the service rejects the conflicting write and commits the current epoch. The client whose write has been rejected, can issue a read for the new version of the conflicted object. After inspecting the new value, the client may issue another write request.

Since the service logic is rather simple, clients can verify on their own the correctness of an individual operation: every write overwrites exactly one object in its entirety, and every read retrieves the entire contents of exactly one object. The header of each write request contains information about the assumed object version at the time the request was made, the time of the request, and the hash of the payload. The service preserves the headers of write requests and uses them to compute the authentication labels for a given object. Thus, to verify an operation, a client obtains a membership proof and the corresponding write header and verifies the proof.

To verify a write operation, a client simply ensures that its write request was used in the calculation of the new state digest (a simple membership proof verification). This can be done either by indicating that the service should delay the acknowledgment of the write operation until the completion of the epoch, or, alternatively, the client can issue a challenge operation for the object once the epoch is committed.

Verification of a read operation is more complicated. First the client must ensure that the returned result represents data written by another authorized client. This can be done by inspecting the returned write header and querying the existing access control policy. The client must also compare the hash stored in the write header to the hash of the returned data. Next, the client must verify the supplied membership proof to ensure that the returned data indeed came from the current service state and it is the same reply that another client would have received. These steps ensure all but the freshness of the returned object value: i.e., the service may have hidden one or
more recent updates from the client.

To ensure freshness, the client can optionally issue an audit request. An audit request specifies a span of time over which depth number of random challenges must be issued. To process an audit request, the service makes the random selection (subject to verification by the client) and returns a set of membership proofs (one for each challenge) in a single response. The client verifies the result of an audit by making sure that the audited object is present (with its current version) in all inspected epochs. Failure to verify an audit constitutes an undeniable proof that the service did not provide a fresh response. Since the service offers strong accountability properties, any evidence of misbehavior can be independently verified by a third party, without having to rely on trust or consensus voting.

5.2 Implementation

We implemented the CATS storage service using C#, .NET framework 1.1., and Web Services Enhancements (WSE) 2.0 for SOAP/DIME communication. The service implementation consists of approximately 6,000 lines of code.

The storage service is log-structured [104]. It consists of an append only log and an index for locating data on the log. We use the CATS toolkit for the index, and a slight modification of the toolkit write ahead log for the append-only log. Each log entry consists of two portions: the XML request header, and the request payload. Clients identify objects using 16 byte key identifiers. The label of each object is a function of its key, the hash of the data, and the hash of the request that generated the data. In this way we achieve tamper-evident linking of the three key elements that determine the contents of an object.

The service is structured as a collection of stages using a custom staging framework inspired by
SEDÁ [127]. A pool of worker threads services requests for each stage. The size of the pool changes dynamically in response to offered load. Stages have queues of fixed capacity to absorb variations in load. Once queues fill up, the service starts rejecting requests: either by dropping them, or by notifying the client to resubmit the request at a later time.

The service and its clients communicate via SOAP over TCP. To limit the overhead of XML, we transfer binary data using DIME [91] attachments. Requests and responses are digitally signed. The current implementation uses a custom XML canonicalization scheme and RSA for signatures. Future releases will integrate the service with web services standards such as XML-Signatures and WS-Security [62].

A write request carries the key of the object to be modified, contains the known version token, and the hash of the data to replace the current object. The request carries a digital signature, which is appended as a custom SOAP header. The signature section lists the hash of the public key that should be used to verify the request. The public key identifies the sender of the message and can be used for access control decisions.

The service interprets an object’s key as a capability and will allow clients that know the key to retrieve and update the corresponding object. It is possible to layer more complex access control schemes if necessary. For example, clients may be required to present signed authorization assertions to allow them to perform an operation. Alternatively, the service can obtain the necessary assertions and integrate them in the state store. In either way, clients and the service are accountable for any write that they request/process. The service is not accountable for allowing illegal reads. Encrypting object contents can prevent possible data leaks.

The service processes update requests in rounds. The length of a round is determined by the incidence of valid (not conflicting) update requests for the same object. If the service receives a valid write request for an object that has already been modified in the current epoch, the service terminates the epoch, creates a new one, and applies the write to it. This approach allows for the best granularity of accountability as it preserves every version of an object. However, it has impact on performance if workload has high contention (Section 5.3.3). An alternative design choice is to limit the rate at which new snapshots are created by imposing a minimum bound on the length of a round. To enforce this bound, the service will have to reject a write if it will generate an epoch.
ahead of its due time. Clients will have to buffer and coalesce their writes, which will result in decreased data sharing.

5.3 Evaluation

In this section we present an evaluation study of our prototype. Our goal is to understand how integrating support for accountability impacts the performance of the storage service. In particular we study the effects of digital signatures, write conflict rate, object size, and auditing and challenges.

5.3.1 Methodology

We run all tests on IBM x335 servers running Windows 2003 Server Standard Edition operating system. Each machine has 2 Pentium IV Xeon processors running at 2.8GHz, 2GB of RAM, 1Gb/s Ethernet, and IBM SCSI hard disks with rotation speed of 10,000 RPM, average latency and seek time of 3 ms and 4.7 ms respectively.

We use SHA-1 to compute authentication labels. The key size we use is 16 bytes, and object keys are assigned randomly in the identifier space. This is a conservative approach intended to limit spatial locality and stress our implementation. In practice, spatial locality can reduce the overhead of persistence and the cost of state digest computation. We store the index and the log in blocks of size 64KB. The index cache has capacity 2000 blocks and the log cache has capacity 8000 blocks. The cache sizes are chosen so that neither the index, nor the append-only log fit in memory. For storage medium we use files residing on the local NTFS file system. The files are accessed using the standard .NET I/O API. For improved performance we place the append-only log and the index on separate disks. We commit dirty blocks to disk every 2 seconds.

We use a population of 1,000,000 unique objects and pre-populate the server to have one version of each object. Since the cost of index operations grows logarithmically with the size of the state store, a state store with 1,000,000 objects has already reached a steady state in which increasing size has minor impact on performance (doubling the size of the store will increase the cost of operations by approximately 3%).

We use a community of clients to issue requests to the service using the SOAP interface. Our workloads consist of synthetic reads/writes of a controlled size. We vary the “heat” of the workload
by biasing the probability of selecting a given object as the target of a read/write operation. We denote heat using the notation $X : Y$, which we interpret to mean: $X\%$ of the requests go to $Y\%$ of the objects. For example 100:100 is a uniform workload, and 80:20 is the typical hot/cold workload. All tests last 3 minutes with initial 30 seconds for warming the caches. We report averages and standard deviations from 10 runs. Our test workloads are a rough approximation of real storage workload and the magnitude of the performance results we report may differ from that of real systems. However, the focus of our study is the cost of accountability and our workloads allow us to vary the key parameters to quantify those effects.

5.3.2 Saturation Throughput

We now measure the saturation throughput of the storage service prototype. The first experiment evaluates the maximum bandwidth a storage service using SOAP/DIME based on WSE 2.0 can achieve. We issue write requests to the service to store objects of controlled sizes. The client follows the full protocol, while the server only extracts the object from the SOAP message, validates it (when requests are signed) and sends a response to acknowledge the operation. No storage takes place. The server's CPU is saturated. Figure 5.3 shows the saturation request rate and the resulting data bandwidth for objects of different size with and without using digital signatures. Although we are using 1Gbps LAN, the observed data bandwidth peaks at approximately 15MB/s. This is the best possible request rate that our storage service can achieve. Importantly, the results show that
**Figure 5.4:** Write saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. As object size increases, the relative cost of digital signatures decreases.

**Figure 5.5:** Read saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. Performance is seek limited due to the log-structured design.

digital signatures are expensive, however, the relative cost of signing a request decreases as object size increases.

In the next experiments we measure the read and write saturation throughput of our implementation under a traditional hot/cold workload (80:20). We use the load generator to fully saturate the server and measure the sustained throughput. We vary object size to study its impact. Figure 5.4 shows the performance for write and Figure 5.5 for read operations. As object size increases, the request rate decreases and the transfer rate increases. With the increase of object size, write performance increases relatively to the maximum achievable request rate. The service offers better write than read performance, which is due to its log-structured design.

As object size increases the relative penalty of using digitally signed communication decreases.
Figure 5.6: Rate of epoch creation as a function of request rate and workload contention. Higher request rates and contention create new epochs faster and can affect service longevity.

Figure 5.7: Epoch length as a function of request rate and workload contention. Epoch length is independent on request rate and is determined by workload contention.

4KB signed writes achieve 42% of the request rate of writes without signatures. This number increases to 86% for object of size 256KB. The ratios are better for reads: 81% for 4KB, and 90% for 64KB. The log-structured design explains this observation: the read workload is seek-dominated and the time spent signing a request is smaller relative to the time spent obtaining the data from the storage medium.

Overall, the experimental results show that our implementation offers reasonable performance for both read and write operations and that the cost of digital signatures is not prohibitive.

5.3.3 Workload Contention

In Section 4.5 we concluded that epoch length has significant impact on performance. The storage service creates a new epoch if it receives more than one update operation to the same object in the
same epoch. As a result of this design choice, the performance of the service will be dependent on
the contention present in its workload. In the next experiments we study this relationship.

We use workloads of different heat, ranging from uniform (100:100) to highly skewed (80:1)
and generate requests at controlled rates below the server saturation level. Figure 5.6 shows the
observed epoch creation rate. For a given workload heat, processing requests at a higher rate creates
new epochs more frequently. As a result, well-provisioned services processing high volumes of client
requests will produce a large number of state snapshots. Similarly, more skewed workloads create
new epochs at a higher rate. Clearly, mechanisms are needed to allow services to discard state
snapshots to keep space overhead and auditing costs at an acceptable level. Figure 5.7 shows the
resulting epoch lengths. Importantly, epoch length does not depend on request rate, however, it
can vary from 1400 updates for a uniform workload, to 200 updates for a highly skewed workload.
These results suggest that for typical usage scenarios index update operations (the primary cost
component of accountability) will be in the range from 300 to 600 μseconds (Figure 4.25).

### 5.3.4 Challenges and Audits

Challenges and audits are the primary mechanisms to ensure a service behaves correctly over time.
In the next experiments we study the impact of these mechanisms on service performance. The
metric of concern is saturation throughput. For these experiments we populate the service so that
the 1,000,000 objects are distributed among 100 epochs of length 10,000 updates. The resulting
index is approximately 15 times larger than the index with all objects stored in an epoch of length
1,000,000. To isolate the impact of storage access on auditing, communication is not signed.

The cost of auditing operations depends on their depth, the number of inspected snapshots, span,
the total number of snapshots in the examined interval, and scope, the fraction of objects that are
likely to be audited, e.g., in our framework objects are audited only on access, and stale objects
may never be audited. Using this terminology, a challenge is an audit of depth 1 and span 1.

Figure 5.8 shows the auditing saturation throughput for different combinations of span, scope,
and depth. For a fixed scope, auditing becomes more expensive as span increases. Similarly, higher
auditing depths are more expensive. Larger scope is also more expensive, however, its impact is
less pronounced. We can explain the above results with the fact that the span and scope determine
the locality of auditing operations: smaller spans and scopes concentrate audits on a portion of the index. Increasing the depth queries more snapshots and reduces the total audits/seconds rate.

Controlling the rate of auditing is an important problem as it can be used to mount denial of service attacks. This is a resource control problem and solutions from the resource management domains are applicable. For example, clients can be allocated auditing budgets and the service can reject audits from a client if the client has exhausted her budget. Importantly, the services can be challenged for such rejections, to which it can reply with a collection of signed client requests to show that the client has exhausted its budget.

5.4 Related Work

SUNDR [75, 81] and Plutus [65] are two recent network storage systems that defend against attacks mounted by a faulty storage server. 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 attacks on data integrity, the most difficult of which is a “forking attack”. In both Plutus and SUNDR file system logic is implemented by clients: the server only stores opaque blocks of data. In SUNDR the server also participates in the consistency protocol.

Both SUNDR and Plutus can detect various attempts by the server to tamper with the contents of stored blocks. However, the papers do not attempt to define precise accountability properties. In general, they blame any inconsistencies on the server: it is not clear to what extent clients are
accountable for their actions or for managing file system metadata correctly. For example, SUNDIG does not show how to defend against a client that covertly deletes a file created by another user in the same group.

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. This is a subtle change of focus as compared to SUNDIR, where the primary concern is to build a tamper-evident system and terminate its use once misbehavior is detected. CATS file semantics are implemented and enforced by the server in the conventional way, rather than implemented entirely by the clients. Clients can independently verify correct execution relative to published digests, rather than “comparing notes” to detect inconsistencies. CATS uses similar techniques to SUNDIR and Plutus to make the stored data tamper-evident, but it also defends against false accusations against a server, e.g., a malicious client that “frames” a server by claiming falsely that it 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.

5.5 Summary

This section presented the application of the state-based approach to accountability. The CATS storage service is built using the methodology and toolkit developed in the previous two chapters. Experiments with the prototype help understand the costs of the approach when applied to a real-world service. We observe that performance depends to a large extent on the write conflict ratio: the higher the conflict ratio, the shorter epochs are and the more expensive data and action authentication becomes. Digital signatures have a noticeable impact on the service’s performance, however, their cost decreases as data size increases. Auditing performance is reasonable and can be used to detect semantic violations. All-in-all, the CATS storage service offers reasonable performance and high level of assurance about its correctness.
Chapter 6

State-based Approach Extensions

The state-based approach to building accountable services and the CATS toolkit offer a set of essential primitives that can be used to construct more complex accountable services. In this chapter we describe several extensions that build upon the basic accountable object storage primitives. We first describe how to extend accountability to update operations that affect several objects, thus, in essence, providing support for accountable transactions. Next, we describe how to implement a dynamic accountable access control service. The service can be used by a CATS storage service to provide accountable attestations about the access permissions of a principal to a given object. We also present the design of an accountable lock server. The lock server design exposes an important problem common to many services: denial of service. Finally, we describe a class of action functions which lend themselves to fully automated verification.

6.1 Updates to Multiple Objects

The CATS protocol, as described in the previous chapters, ensures the accountability of update operations that affect a single object. While, this is an essential primitive, many services can benefit from the ability to perform accountable updates that affect multiple objects. As an example, consider uses of the CATS storage service (Chapter 5) to store a very large object. Since the CATS toolkit supports only full object overwrites, object updates in this case are expensive. A client of the service can benefit significantly by splitting the object into multiple blocks of manageable size: the first block can be used to describe all other blocks that comprise the object. To perform an update to the object, the client issues one update that consists of several full-block overwrites. If the server can ensure the accountability of the compound update operation, then the client is assured that its update is safe.

Ensuring the correctness of multi-object updates requires that the server either apply all individual updates or none at all. In essence, multi-object updates are transactions; they succeed only if each individual write is successful. To ensure the accountability of multi-object updates special
care must be taken to prevent a fraudulent server from interleaving the updates of multiple users and thus presenting an inconsistent view.

We can ensure the correctness of multi-object updates using a version number vector. The vector has an entry for each object affected by the update operation and it contains the version number of that object prior to the update. Each individual update is annotated with the version vector and is signed, thus the version vector is used in the calculation of each object’s hash, and gets represented in the server state digest. When processing a multi-object update, the server must reject any update with an invalid version vector, e.g., an object to be affected by the update has already received another update.

As with single object updates, we verify the correctness of multi-object updates by issuing a challenge. However, challenges to an object whose value has been the result of a multi-object update must be followed by challenges to all objects that were part of the multi-object update. The additional challenges are necessary to ensure that the service applied the update as a whole instead of selectively choosing to update some of the affected variables.

Verifying the correctness of a multi-object update is a potentially expensive operation. We can reduce the cost by selecting for verification only a random subset of the objects at the expense of reduced misbehavior detection probability. If the detection probability must remain at a fixed level both for single and multi-object updates, the benefits of supporting multi-object updates come at the expense of increased verification costs: the larger the number of objects inside a single multi-object update, the more expensive the audits required to bound the probability of misbehavior.

We can calculate the probability of a successful audit for multi-object update as follows. Let $K \in N$ be the number of objects in a multi-object update. A misbehaving server tampers with the contents of $L$ of them, $0 \leq L \leq K$. An audit of $m$ out of the $K$ objects will fail to select a tampered object with probability $\prod_{i=0}^{m-1} \frac{K-i}{K}$. For each object selected for auditing we need to verify that its version number, as represented within the update’s version vector, matches the version number of the object before the update; this step ensures that the multi-object update was applied correctly.

If some time has elapsed from the moment the update was applied, we also need to ensure freshness to rule out the probability of reverted writes (Section 3.2.5). If freshness is a concern, then we will have to resort to probabilistic audits over the period from the time the update was
applied to the time of the query. Let this period span $A$ snapshots. If for each object chosen for audit, we select $n$ randomly chosen snapshots over the suspected interval, then the probability of missing a reverted update is at most: $\prod_{i=0}^{n-1} \frac{1}{A-i}$. Therefore, the probability that we will miss a misbehavior in a multi-object update after choosing $m$ objects for audit and auditing each of them over $n$ snapshots is at most:

$$\prod_{i=0}^{m-1} \frac{L}{K-i} + \prod_{i=0}^{m-1} \frac{L}{K-i} \times \prod_{i=0}^{n-1} \frac{1}{A-i}$$

Figure 6.1 plots the probability of not detecting a violation in a multi-object update of age 10 snapshots spanning 10 objects. The fact that now there are more objects that need to be checked for freshness adds a multiplicative factor to the probability of failed detection. Similarly to object age, as the number of objects in a multi-object update increases, it takes significantly more samples to ensure a given probability of detection.

In certain cases we can accept a lower detection probability. For example, if an object is frequently accessed, we can relax the requirements for individual audits and still get a significantly high probability of detection when considering all accesses: it takes a single successful detection to show that a server misbehaves.

![Figure 6.1: Probability of not detecting a violation in a multi-object update of age 10 snapshots spanning 10 objects.](image)
6.2 Accountable Access Control

Verifying the correctness of access control decisions is an important part of enforcing accountability. The CATS storage service uses simple static ACLs provided by an object’s creator to guide its access control decisions. While these ACLs allow the system to demonstrate that it accepts updates only from authorized principals, their static nature restricts flexibility: one cannot change the permissions of an object dynamically. Our design makes it possible to alleviate this problem by using a separate authorization service. The service manages dynamic access control lists and provides signed attestations about the access permissions (or lack of) of a given principal. While the authorization service can be a trusted component, there is an opportunity to reduce the level of trust by making the authorization service itself accountable. Since authorization is an essential component of many systems, accountable authorization has the potential to enable accountability in a large class of applications.

As a starting point, we can formalize the internal state of an authorization service as a set of boolean variables, one for every possible (object, principal, right) triple. Authorization queries and updates to access control policy reduce to reads and writes to one or more boolean variables. Note that this state representation does not allow for groups of principals. Groups are not supported by our design and we discuss this limitation at the end of this section.

Now, let us assume that the set of boolean variables is static and each variable has been signed by a trusted third party to certify its validity. We can make the authorization service accountable for its responses to queries by storing all boolean variables using the CATS toolkit. Each variable is represented by an object with key (object, principal, right). For simplicity, we can consider each key to be a concatenation of three fixed size keys and comparison is lexicographic, i.e., all variables involving object X appear before variables involving object Y. This organization enables us to verify the correctness of query responses: each affirmative response will produce a signed variable that is also a part of the current state. Similarly, each negative response will demonstrate that the variable is not part of the state. Challenges and audits can be used to ensure the freshness of responses.

Now, let us consider making the service dynamic, i.e., we want to allow the service to accept updates to the contents of one or more boolean variables. The problem with updates is that each update itself must be properly authorized! We can deal with this circular dependency using the
following rules:

1. Each update is signed by the principal making the update. We will denote such update as \((object_j, principal_i, right, 0|1)_{principal_k}\). This notation represents an update to variable \((object_j, principal_i, right)\) by \(principal_k\) setting the variable’s value to be either 0 or 1; 0 means access denied, 1 means access granted.

2. A principal \(i\) indicates ownership of an object \(j\) by issuing the following update: 
\((object_j, principal_i, owner', 1)_{principal_i}\). The service accepts such updates only if at the time of the update there are no boolean variables of the form \((object_j, *, *)\). This fact can later be demonstrated by a challenge for \((object_j, 0, 0)\) in the service state snapshot at the time before the update. The challenge must demonstrate that \((object_j, 0, 0)\) is not in the service state and that the keys immediately before and after \((object_j, 0, 0)\) (in sorted order) in the service state do not involve \(object_j\) as a component.

3. Principals that possess the owner role for a given object can delegate (or revoke) rights to that object. More formally, 
\((object_j, principal_k, right, 0|1)_{principal_i}\) is accepted only if \((object_j, principal_i, owner')\) exists and is equal to 1.

To enforce accountability, compliance with the above rules must be verified during query operations. To verify a query one must not only ensure that the response is fresh, but must also verify that the creator of the variable’s content was itself authorized to perform the update. Therefore, if the service responds that \((object_j, principal_i, right)\) exists and it was signed by \(principal_k\) we must ensure that \((object_j, principal_k, owner')\) exists and is equal to 1. If \((object_j, principal_k, owner')\) was signed by \(principal_k\), we must verify that \((object_j, 0, 0)\) did not exist prior to the creation of \((object_j, principal_k, owner')\). On the other hand, if \((object_j, principal_k, owner)\) was signed by a different principal, we must follow the delegation chain recursively.

The above paragraphs describe the high-level details about implementing a simple accountable access control system. The access control system uses the basic toolkit abstractions of read, write, challenge, and audit to ensure the accountability of access control queries and delegations. This service can serve as a building block of many distributed systems, where access to shared resources
must be controlled, e.g., the Grid [97]. The service described here, however, is not a full-fledged Role-based Access Control Service [47] (RBAC).

A key challenge in extending the design to an RBAC service is that a generic RBAC service includes the notion of subject and object groups, which significantly complicate the design of an accountable authorization service. While it is possible to verify positive statements, e.g., “principal A has right B over object C”, verifying negative statements, e.g., “principal A does not have right B over object C” is not trivial. To verify a positive statement it is sufficient to return a set of signed attestations representing group membership and group role assignments. By applying the rules of group membership it is possible to demonstrate that the attestations justify the positive statement. However, to verify the correctness of a negative statement one must demonstrate that no rules stored in the authorization service can be combined to contradict the statement. Presenting the whole service state as evidence solves this problem, however, the cost of performing verification in this way is prohibitive.

One approach to provide support for groups can involve translating all group relationships to direct ones. For example, if A is a member of B, and B is a member of C, then the fact that A is a member of C must be explicitly represented in the state rather than derived. Updates in this case become more complex and may involve multiple variables (Section 6.1). For example, changing a group’s membership may trigger a large number of updates as new group/object/role relationships become valid and prior ones become invalidated. This approach scales badly as the size and number of groups increases.

6.3 Accountable Lock Service

Lock servers offer another important service for distributed systems. A lock service enables its clients to obtain and release locks over shared resources and as such serves as a synchronization point in a distributed system ensuring correctness and event ordering. Lock servers are, generally, trusted, but we can leverage the state-based approach to make the operations of a lock service accountable.

A lock service supports the following operations:

- $acquire(X)$ - The caller expresses an intention to acquire exclusive control over resource $X$.

If no other client currently owns $X$, the caller is granted $X$. If another client owns $X$, the call
is blocked until the owner releases $X$.

- $\text{release}(X)$ - The caller releases exclusive control over resource $X$. The client must have been previously granted exclusive control over $X$ by a call to $\text{acquire}(X)$. After the call completes, the client no longer has exclusive control over $X$.

We now describe how to make both operations accountable. We represent resource $X$ by a state-variable with $\text{oid} X$. The value of the variable can be either 0 (the lock is free), or 1 (the lock has been taken). Each value is digitally signed by a client of the service. The first $\text{acquire}$ request for a given resource creates the corresponding state variable.

Each acquire request carries the value 1 signed by the client. The service processes $\text{acquire}(X)$ as follows:

1. Rejects the request if the client supplied a value different from 1 and/or did not sign it properly.
2. Retrieves $X$ from its internal state, together with a membership proof $P$.
3. If $X$ exists and its value is 0, or if $X$ does not exist, the service updates $X$ with the value supplied by the client. The service uses the client request as the action record to justify the write. Go to step 5.
4. If $X$ exists and its value is 1, the service blocks the request until the next $\text{release}$ request for the same resource.
5. The service obtains a membership proof for the new value of $X$ that it just wrote, $P'$
6. The service sends the new signed value of $X$, $P'$, and $P$ to the caller.

Each $\text{release}(X)$ request carries the signed value and membership proof that the service returned to the client in the preceding $\text{acquire}(X)$ call ($P'$). In addition, the client also sends the value 0 signed by its key as the new value for $X$. The service processes $\text{release}(X)$ as follows:

1. Rejects the request if the client supplied a value different from 0 and/or did not sign it properly.
2. Retrieves $X$ from its internal state and ensures that its value is 1 and that its version number matches the version number reflected in $P'$.
3. If the values do not match the service sends an error response to the client.

4. If the values match, the service updates the value of \( X \) with the value supplied by the client.

5. The service supplies a membership proof \( P'' \) for the new value of \( X \).

6. The service sends \( P'' \) back to the caller.

Defined in this way, acquire and release operations become verifiable and accountable. Since both operations translate into updates to the corresponding state variable and the new values for the state variable are created by the caller, the caller has sufficient information to determine if:

- the service applied the update to its state
- the previous value for an acquire operation was 0 and for a release operation: 1

These steps prevent the service from granting the lock simultaneously to more than one client. If the lock is currently in use, the service cannot grant the lock to another client without risking detection. Similarly if a client does not own the lock, the service cannot honor this client’s release request without risking detection.

Note that a client does not need to obey any locking protocol when using a shared resource. However, if that resource uses an accountable update protocol, it should be possible to audit the client’s access to the resource and determine if the client was holding the required locks.

The protocol we described ensures that the lock service is accountable for any updates to its internal state. However, this protocol does not prevent the service from refusing to apply the updates of a client: a client may issue an acquire request, which could either not be honored at all, or be satisfied at a much later time. In essence, the design does not ensure lock fairness and does not protect against starvation. This is an instance of the more general denial of service problem, which is the subject of the next section in this chapter.

One possible approach to dealing with the problem of fairness is to have clients timestamp each request. If the source of timestamps can be trusted, or held accountable, e.g., it is a timestamping service \([16, 25]\), and if the service is required to timestamp its epoch digests, we could verify the basic fairness property: the service should respond to an acquire request immediately if the lock variable has value 0 in the epoch corresponding to the request. We can relax the need for a
trusted/accountable time source, by providing for a window within which a service must process a request. The service should never honor an acquire request outside of the allowable window. This approach makes violations of fairness detectable, but it may also cause the service to starve forever requests that have been delayed for too long.

6.4 Denial of Service

The state-based approach presented in this thesis intends to ensure the accountability of any update to the internal state of a service. Any update a service applies to its state can be audited: if the service, or the requesting client, committed a violation of the established service semantics, this violation can be detected and proven. However, this methodology applies only to a subset of all requests that a service receives, namely, the requests the service chooses to honor. If the service deliberately avoids to honor a given request and does not commit any changes to its state as a result of this request, the service cannot be held accountable. In effect, a malicious service may mount denial of service attacks, by refusing to honor specific requests. For example, if a request is likely to expose a misbehavior by the service or by another client, the service may choose to ignore this request and never respond back to the client.

A client can detect that its requests are not processed by the service with relative ease, however, proving that the service deliberately denies its requests is difficult. A client, who has been refused access cannot prove undeniably that it did not receive a response without using help from a third party. Since the client controls its own machine, the client can falsify any evidence that it may use to justify its case. A third party can corroborate the client’s case, but to use a third party for this purpose, that party must be trusted by both the client and the service.

Denial of service represents an important class of problems for which it is difficult and maybe impossible to enforce accountability without making additional assumptions and restrictions. The underlying problem in many cases is the inability to prove undeniably, without requiring trust in a third party, that a given message was (not) sent or was (not) received at a given time. There is no way to verify claims about message initiation or receipt using only evidence from the server or receiver.

Our best strategy in such cases is to find a common trust point and require that any statements
about sending or receiving messages involve that common trust point. Examples of common trust points include secure hardware and escrow services. An alternative solution may be to modify the communication protocol between clients and services so that a service would not have information about the actual identity of the sender (or the actual contents of the message) at the time it receives a message. Such an approach can reduce the problem to an instance of the fair-exchange problem [96], which requires a trusted third party [95, 107].

While it may be impossible to hold a server accountable for denying service to a client, the methodology described in this thesis ensures that when a server does determine to process a request the server will be accountable for the way it processes the request. A server which is selectively ignoring client requests is already suspect by the client and, sooner or later, the client will find enough evidence to support its claim. For example, the client can have another client issue the same operation. In general, a server must process more requests than it rejects. Making the server accountable for the way it handles acknowledged requests covers a large fraction of the possible attack space. If higher degree of assurance is required, then it may be unavoidable to introduce a trusted component.

6.5 Accountable State Machines

We now revisit the problem of ensuring the correctness of an action using the state-based approach. The essence of the state-based approach is to preserve all inputs used by the service to produce the result of an action function. To verify the correctness of the calculations performed by the service an auditor needs to obtain the inputs and feed them to the action’s verification function. The complexity of the action’s verification function determines the frequency with which an auditor or the clients of a given service can validate the correctness of computations made by the service. Ideally, verification functions are simple enough so that clients can perform that verification themselves.

If a client is able to verify the result of an action function independently without requiring significant resources, the methodology described in this thesis becomes more attractive. More frequent verification increases the probability of detecting a misbehavior and reduces the window of vulnerability (Chapter 2) during which the service state is inconsistent with the actions applied to it. The more complex a verification function is, the higher the cost clients have to pay to verify the
function, and the larger the window of vulnerability.

\[ \text{Figure 6.2: Some state variables can be defined as state machines with well-known states and transition functions. Action functions involving such variables simply select the correct transition and produce the new value. Such action functions have well-defined verification functions, which can be generalized and automated.} \]

To understand better the applicability of the approach, we focus on a class of action functions each of which represents a transition in a finite state machine (Figure 6.2). For simplicity we can consider functions whose input set consists of a single state variable. The value set of the state variable is known in advance and the changes to the value of the state variable can be described as transitions in a state machine. The transition function of the state machine is part of the specification of the service and is well-known and accessible to clients and auditors.

It is straightforward to verify the correctness of action functions defined in this way: the verifier (client or auditor) can obtain the function's specification and check that the resulting state corresponds to the chosen transition. Importantly, this process is generic and can be fully automated, as long as it is possible to formalize the changes to a state variable as transitions in a state machine.

State machines offer a powerful abstraction that can be leveraged to automate the process of verifying the correctness of an action. Since the state-space of a function increases rapidly with the complexity of the task being performed, this approach is likely to be applicable mostly to services that perform relatively simple operations. Ideally, such services can serve as the building blocks for other more complex services.

\section{6.6 Summary}

This section presented several extensions and applications of the state-based approach. We described an approach to making transactional updates accountable. In addition to the age of an update,
the cost of accountability for transactional updates also depends on the number of state variables affected by the transaction. We also described how to make an authorization server accountable by reorganizing its internal state and protocols. We applied the same approach to a lock server and described the denial of service problem. Finally, we presented a class of applications, which may benefit from an automated verification of their action functions.
Chapter 7

Privacy Protection and Accountability

In the previous chapters we developed and evaluated a methodology for making network services and their clients accountable for actions that affect shared state. The techniques we described ensure that changes to the shared state can be associated with the actor who requested or performed them. If a change represents a violation of established semantics, then the violation can be demonstrated and the actor is accountable. However, not all actions modify state, and the methodology we described so far does not provide accountability for actions that do not result in state changes. In this section we address the problem of privacy as it relates to the subject of accountability.

7.1 Accountability for Data Dissemination

Holding someone accountable for disclosing private or sensitive information is a different and complementary problem to the one addressed in the previous chapters of this dissertation. The core of our state-based approach is to augment the state-change protocol of a service to ensure that state changes are performed and visible to all clients of a service. Any time a state change is performed, the change must be accompanied by an action record and a state digest is eventually published to the publishing medium. Whenever a client wants to use the result of a state update operation, the client can use the challenge and audit interfaces to ensure the correctness of the provided data.

These interfaces and protocols work because the actions in question either make changes to shared state or clients want to make sure that data fetched from shared state are authentic. If an action does not make changes to shared state and a client does not want to ensure the authenticity of data, then we need to supplement the state-based methodology with additional techniques to ensure that private information is not disseminated in violation of existing access control policies. There are two major problems that we need to address

113
7.1.1 Problem I: No State Changes

Reading information does not modify the shared state of a service: anyone can make a copy of a byte stream and pass it to another actor. If the byte stream is not encrypted, anyone can extract whatever information is encoded in it. Encryption can ensure the privacy of the stream’s contents but is not a solution; the stream’s endpoints usually have access to the plaintext.

In the context of a shared service, for encryption to protect sensitive data, the service should operate only on encrypted data. Any operation the service performs should be applicable to encrypted data, to avoid the possibility of the service disclosing data in plaintext. This requirement, while feasible for some arithmetic operations, greatly reduces the functionality of the shared service, transforming it mostly into a storage server, that performs, none, or very limited computation.

When encryption is used, only the owner of a piece of data should be allowed to decrypt its contents and get access to the plaintext. However, if the owner must share data with another actor, the owner must provide a copy of the data, which the other actor must be able to decrypt. Digital watermarking solutions exist to ensure that any copies of decrypted data can be traced back to the original recipient. Digital watermarks, however, do not protect against using the original copy as a guide and synthesizing new piece of data, e.g., someone creating a new Excel document and manually entering the numbers stored in the original balance sheet. Most importantly, without requiring additional trusted entities, the owner of a piece of data can deliberately disseminate it to frame another actor.

7.1.2 Problem II: No Trusted Path

Information leaks are further complicated by the fact that data are typically managed not directly by their owner, but by software acting on behalf of the owner. If a piece of software is allowed to access some piece of information, it may be possible for that software to use the information in a way that the user did not intend. In essence, a piece of software can perform actions that can be attributed to the operator/owner of the software and she could be held accountable for them. This in essence is the trusted path problem, which we first described in Chapter 3. In this section we focus on an instance of the problem as it applies to data dissemination.

The mere possibility of having software behave differently from what its end user intended,
provides a convenient loophole for avoiding responsibility when misbehavior is detected. An end user always has the option of blaming the problem on its hardware and software. Before we can accuse actors for disclosing information, we must ensure that we provide the means to detect when such disclosures take place and prevent them from happening. Most actors are generally not malicious and are well-intended and would prevent accidental disclosure when alerted.

7.1.3 Detect Now or After the Fact?

Private information has an important property: private information is important and useful only until it is revealed. Once revealed, private information is no longer secret and it is likely to be impossible to restore its secrecy. In many cases, being able to trace disclosed information back to its originator, is a necessary, but not sufficient requirement to ensure the stability of a system. In environments where data confidentiality is a concern preventing information leaks is paramount. For such environments accountability may not provide the incentives to prevent dissemination completely, and even a small probability of a leak can jeopardize the system.

The uncertainty of being able to hold actors accountable for disseminating confidential information and the potential damage from disclosed information speak in favor of approaches that focus on preventing information leaks rather than relying solely on after-the-fact checks to expose leaked information and the actor responsible for it. We argue that, while useful, accountability for disseminating confidential information may not be possible and it may not even be sufficient to ensure the integrity of a system with shared state. As far as data privacy is concerned, an approach that intends to prevent disclosure at the moment confidential information is about to be leaked is likely to be more useful than one intended to detect disclosure at a later time.

To this end, we describe our work on a privacy management system called TightLip which intends to prevent leaks of sensitive data in the context of a commodity desktop operating system. TightLip identifies and tracks a sensitive data item as it moves within the operating system and is used by the various software running on it. TightLip's primary goal is to detect when potentially sensitive information is about to leave the boundaries of the user's system and prevent the user from becoming accountable for something she did not intend. TightLip does not depend on the existence of trusted hardware, it only assumes that the operating system is trusted and secure.
7.2 Data Leaks and Access Control Misconfigurations

Email, the web, and peer-to-peer file sharing have created countless opportunities for users to exchange data with each other. However, managing the permissions of the shared spaces that these applications create is challenging, even for highly skilled system administrators [73]. For untrained PC users, access control errors are routine and can lead to damaging privacy leaks. A 2003 usability study of the Kazaa peer-to-peer file-sharing network found that many users share their entire hard drive with the rest of the Internet, including email inboxes and credit card information [54]. Over 12 hours, the study found 156 distinct users who were sharing their email inboxes. Not only were these files available for download, but other users could be observed downloading them. Examples of similar leaks abound [74, 82, 99, 131].

Secure communication channels [28, 44] and intrusion detection systems [39, 51] would not have prevented these exposures. Furthermore, the impact of these leaks extends beyond the negligent users themselves since leaked sensitive data is often previous communication and transaction records involving others. No matter how careful any individual is, her privacy will only be as secure as her least competent confidant. Prior approaches to similar problems are either incompatible with legacy code [46, 67, 87, 112], rely on expensive binary rewriting and emulation [31, 90], or require changes to the underlying architecture [40, 116, 121].

We are exploring new approaches to preventing leaks due to access control misconfigurations through a privacy management system called TightLip. TightLip’s goal is to allow organizations and users to better manage their shared spaces by helping them define what data is important and who is trusted, rather than requiring an understanding of the complex dynamics of how data flows among software components. Realizing this goal requires addressing three challenges: 1) creating file and host meta-data to identify sensitive files and trusted hosts, 2) tracking the propagation of sensitive data through a system and identifying potential leaks, and 3) developing policies for dealing with potential leaks. This chapter focuses a new operating system object we have developed to deal with the second challenge: Doppelganger processes.

Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an original process. In TightLip, doppelgangers are spawned when a process tries to read sensitive data. The kernel returns sensitive data to the original and scrubbed data to the doppelganger.
doppelganger and original then run in parallel while the operating system monitors the sequence and arguments of their system calls. As long as the outputs for both processes are the same, then the original’s output does not depend on the sensitive input with very high probability. However, if the operating system detects divergent outputs, then the original’s output is likely descended from the sensitive input.

A breach arises when such an output is destined for an endpoint that falls outside of TightLip’s control, such as a socket connected to an untrusted host. When potential breaches are detected, TightLip invokes a policy module, which can direct the operating system to fail the output, ignore the alert, or even swap in the doppelganger for the original. Using doppelgangers to infer the sensitivity of processes’ outputs is attractive because it requires only minor changes to existing operating systems and no modifications to the underlying architecture or legacy applications.

We have added support for doppelgangers to the Linux kernel and currently support their use of the file system, UNIX domain sockets, pipes, network sockets, and GUIs. Early experience with this prototype has shown that doppelgangers are useful for an important subset of applications: servers which read files, encode the files’ content, and then write the resulting data to the network. Micro-benchmarks of several common file transfer applications as well as the SpecWeb99 benchmark demonstrate that doppelgangers impose negligible performance overhead under moderate server workloads. For example, SpecWeb99 results show that Apache running on TightLip exhibits only a 5% slowdown in request rate and response time compared to an unmodified server environment.

7.3 Overview

Access control misconfigurations are common and potentially damaging: peer-to-peer users often inadvertently share emails and credit card information [54], computer science department faculty have been found to set the permissions of their email files to all-readable [131], professors have inadvertently left students’ grade information in their public web space [99], a database of 20,000 Hong Kong police complainants’ personal information was accidentally published on the web and ended up in Google’s cache [74], and UK employees unintentionally copied sensitive internal documents to remote Google servers via Google Desktop [82]. Because these breaches were not the result of buggy or malicious software, they present a different threat model than is normally assumed by the
privacy and security literature.

TightLip addresses this problem in three phases: 1) help users identify sensitive files, 2) track the propagation of sensitivity through a running system and detect when sensitive data may leave the system, and 3) enable policies for handling potential breaches. The focus of this chapter is on the mechanisms used in phase two, but the rest of this section provides an overview of all three.

7.3.1 Identifying Sensitive Files

To identify sensitive data, TightLip periodically scans each file in a file system and applies a series of diagnostics, each corresponding to a different sensitive data type. These diagnostics use heuristics about a file’s path, name, and content to infer whether or not it is of a particular sensitive type. For example, the email diagnostic checks for a “.pst” file extension, placement below a “mail” directory, and the ASCII string “Message-ID” in the file.

This scanning process is similar to anti-virus software that uses a periodically updated library of definitions to scan for infected files. The difference is that rather than prompting users when they find a positive match, diagnostics silently mark the file as sensitive and invoke the type’s associated scrubber.

Scrubbers use a file’s content to produce a non-sensitive shadow version of the file. For example, the email scrubber outputs a properly formatted shadow email file of the same size as the input file, but marks out each message’s sender, recipient, subject, and body fields. Attachments are handled by recursively invoking other format-preserving, MIME-specific scrubbers. When the system cannot determine a data source’s type it reverts to the default scrubber, which replaces each character from the sensitive data source with the “x” character.

7.3.2 Sensitivity Tracking and Breach Detection

Once files have been labeled, TightLip must track how sensitive information propagates through executing processes and prevent it from being copied to an untrusted destination. This problem is an instance of information-flow tracking, which has most commonly been used to protect systems from malicious exploits such as buffer overflows and format string attacks. Unfortunately, these solutions either suffer from incompatibility with legacy applications [46, 67, 87, 112, 134], require expensive binary rewriting [31, 32, 40, 90, 116], or rely on hardware support [121].
Instead, TightLip offers a new point in the design space of information-flow secure systems based on doppelganger processes. Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an original process. Figure 7.1 shows a simple example of how doppelgangers can be used to track sensitive information. Initially, an original process runs without touching sensitive data. At some point, the original attempts to read a sensitive file, which prompts the TightLip kernel to spawn a doppelganger. The kernel returns the sensitive file’s content to the original and the scrubbed content of the shadow file to the doppelganger.

Once the reads have been satisfied, the original and doppelganger are both placed on the CPU ready queue and, when scheduled, modify their private memory objects. The operating system subsequently tracks sensitivity at the granularity of a system call. If the doppelganger and original generate the same system call sequence with the same arguments, then these outputs do not depend on either the sensitive or scrubbed input with high probability and the operating system does nothing. This might happen when an application such as a virus scanner handles sensitive files, but
does not act on their content.

However, if the doppelganger and original make the same system call with different arguments, then the original’s output likely depends on sensitive data and the objects the call modifies are marked as sensitive. As long as updated objects are within the operating system’s control, such as files and pipes, then they can be transitively labeled. However, if the system call modifies an object that is outside the control of the system, such as a socket connected to an untrusted host, then allowing the original’s system call may compromise confidentiality.

By tracking information-flow at a relatively coarse granularity, TightLip avoids many of the drawbacks of previous approaches. First, because TightLip does not depend on any language-level mechanisms, it is compatible with legacy applications. Second, comparing the sequence and arguments of system calls does not require hardware support and needs only minor changes to existing operating systems. Third, the performance penalty of introducing doppelgangers is modest; the overhead of scheduling an additional process is negligible for most workloads.

Finally, an important limitation of existing information-flow tracking solutions is that they cannot gracefully transition a process from a tainted (i.e., having accessed sensitive data) and to an untainted state. A list of tainted memory locations or variables is not enough to infer what a clean alternative would look like. Bridging this semantic gap requires understanding a process’s execution logic and data structure invariants. Because of this, once a breach has been detected, prior solutions require all tainted processes associated with the breach to be rebooted. While rebooting purges taint, it also wipes out untainted connections and data structures.

Doppelgangers provide TightLip with an internally consistent, clean alternative to the tainted process; as long as shadow files are generated properly, doppelgangers will not contain any sensitive information. This allows TightLip to swap doppelgangers in for their original processes—preserving continuous execution without compromising confidentiality.

### 7.3.3 Disclosure Policies

Once the operating system detects that sensitive data is about to be copied onto a destination outside of TightLip’s control, it invokes the disclosure policy module. Module policies specify how the kernel should handle attempts to copy sensitive data to untrusted destinations. Our current
prototype supports actions such as disabling a process’s write permissions, terminating the process, scrubbing output buffers, or swapping the doppelganger in for the original.

TightLip provides default policies, but also notifies users of potential breaches so that they can define their own policies. Query answers can be delivered synchronously or asynchronously (e.g. via pop-up windows or emails). Answers can also be cached to minimize future interactions with the user.

7.4 Limitations

Though TightLip is attractive for its low overhead, compatibility with legacy applications and hardware, and support for continuous execution, it is not without its limitations. First, in TightLip the operating system is completely trusted. TightLip is helpless to stop the kernel from maliciously or unintentionally compromising confidentiality. For example, TightLip cannot prevent an in-kernel NFS server from leaking sensitive data.

Second, TightLip relies on scrubbers to produce valid data. An incorrectly formatted shadow file could crash the doppelganger. In addition, swapping in a doppelganger is only safe if scrubbers can remove all sensitive information. While feasible for many data types, it may not be possible to meet these requirements for all data sources.

Third, scrubbed data can lead to false negatives in some pathological cases. For example, an original process may accept a network query asking whether a sensitive variable is even or odd. TightLip could generate a scrubbed value that is different from the sensitive variable, but of the same parity. The output for the doppelganger and original would be the same, despite the fact that the original is leaking information. The problem is that it is possible to generate “unlucky” scrubbed data that can lead to a false negative. Such false negatives are unlikely to arise in practice since the probability of a collision decreases geometrically with the number of bits required to encode a response.

Fourth, TightLip avoids the overhead of previous approaches by focusing on system calls, rather than individual memory locations. Unfortunately, if a process reads sensitive data from multiple sources, TightLip cannot compute the exact provenance of a sensitive output. While this loss of information makes more fine-grained confidentiality policies impossible, it allows us to provide
practical application performance.

Fifth, TightLip does not address the problem of covert channels. An application can use a variety of covert channels to transmit sensitive information [105, 121]. Since it is unlikely that systems can close all possible covert channels [121], dealing with covert channels is beyond the scope of this chapter.

Finally, TightLip relies on comparisons of process outputs to track sensitivity and transitively label objects. If the doppelganger generates a different system call than its original, it has entered a different execution state and may no longer provide information about the relationship between the sensitive input and the original’s output. Such divergence might happen if scrubbed input induced a different control flow in the doppelganger. Without the doppelganger as a point of comparison, any object subsequently modified by the original must be marked sensitive; this can lead to incorrectly labeled objects and false positives.

This limitation of doppelgangers is similar to those faced by taint-flow analysis of “implicit flow.” Consider the following code fragment, in which variable x is tainted: `if(x) { y=1; } else { y=0; }`. Variable y should be flagged since its value depends on the value of x. Tainting each variable written inside a conditional block captures all dependencies, but can also implicate innocent variables and raise false positives. In practice, following dependencies across conditionals is extremely difficult without carefully-placed programmer annotations [134]. Every taint-checker for legacy code that we are aware of ignores implicit flow to avoid false positives.

Despite the challenges of conditionals, for an important subset of applications, it is reasonable to assume that scrubbed input will not affect control flow. Web servers, peer-to-peer clients, distributed file systems, and the sharing features of Google Desktop blindly copy data into buffers without interpreting it. Early experience with our prototype confirms such behavior and the rest of this chapter is focused on scenarios in which scrubbed data does not affect control flow.

Much of the rest of our discussion of TightLip describes how to eliminate sources of divergence between an original and doppelganger process so that differences only emerge from the initial scrubbed input. If any other input or interaction with the system causes a doppelganger to enter an alternate execution state, TightLip may generate additional false positives.
Table 7.1: Doppelganger-kernel interactions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Processing description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel update</td>
<td>bind</td>
<td>Apply original, return result to both.</td>
</tr>
<tr>
<td>Kernel read</td>
<td>getpid</td>
<td>Verify identical system call sequences.</td>
</tr>
<tr>
<td>Non-kernel update</td>
<td>send</td>
<td>Synchronize, compare buffers.</td>
</tr>
<tr>
<td>Non-kernel read</td>
<td>gettimeofday</td>
<td>Buffer original results, return to both.</td>
</tr>
</tbody>
</table>

7.5 Design

There are two primary challenges in designing support for doppelganger processes. First, because doppelgangers may run for extended periods and compete with other processes for CPU time and physical memory, they must be as resource-efficient as possible. Second, since TightLip relies on divergence to detect breaches, all doppelganger inputs and outputs must be carefully regulated to minimize false positives.

7.5.1 Reducing Doppelganger Overhead

Our first challenge was limiting the resources consumed by doppelgangers. A doppelganger can be spawned at any point in the original’s execution. One option is to create the doppelganger concurrently with the original, but doing so would incur the cost of monitoring in the common case when taint is absent.

Instead, TightLip only creates a doppelganger when a process attempts to read from a sensitive file. For the vast majority of processes, reading sensitive files will occur rarely, if ever. However, some long-lived processes that frequently handle sensitive data such as virus scanners and file search tools may require a doppelganger throughout their execution. For these applications, it is important that doppelgangers be as resource-efficient as possible.

Once created, doppelgangers are inserted into the same CPU ready queue as other processes. This imposes a modest scheduling overhead and adds processor load. However, unlike taint-checkers, the fact that doppelgangers have a separate execution context enables a degree of parallelization with other processes, including the original. Though we assume a uni-processor environment throughout this chapter, TightLip should be able to take advantage of emerging multi-core architectures.

To limit memory consumption, doppelgangers are forked from the original with their memory marked copy-on-write. In addition, nearly all of the doppelganger’s kernel-maintained process state
is shared read-only with the original, including its file object table and associated file descriptor namespace. The only separate, writable objects maintained for the doppelganger are its execution context, file offsets, and modified memory pages.

### 7.5.2 Doppelganger Inputs and Outputs

In TightLip the kernel must manage doppelganger inputs and outputs to perform three functions: prevent external effects, limit the sources of divergence to the initial scrubbed input, and contain sensitive data. To perform these functions, the kernel must regulate information that passes between the doppelganger and kernel through system calls, signals, and thread schedules.

Kernel-doppelganger interactions fall into one of the following categories: kernel updates, kernel reads, non-kernel updates, and non-kernel reads. Table 7.1 lists each type, provides an example system call, and briefly describes how TightLip regulates the interaction.

#### Updates to Kernel State

As with speculative execution environments [30, 92], TightLip must prevent doppelgangers from producing any external effects so that it remains intrusive. As long as an application does not try to leak sensitive information, it should behave no differently than in the case when there is no doppelganger.

This is why original processes must share their kernel state with the doppelganger read-only. If the doppelganger were allowed to update the original’s objects, it could alter its execution. Thus, system calls that modify the shared kernel state must be strictly ordered so that only the original process can apply updates.

System calls that update kernel state include, but are not limited to, exit, fork, time, lseek, alarm, sigaction, gettimeofday, settimeofday, select, poll, read, fcntl, bind, connect, listen, accept, shutdown, and setsockopt.

TightLip uses barriers and condition variables to implement these system calls. A barrier is placed at the entry of each kernel modifying call. After both processes have entered, TightLip checks their call arguments to verify that they are the same. If the arguments match, then the original process executes the update, while the doppelganger waits. Once the original finishes,
TightLip notifies the doppelganger of the result before allowing it to continue executing.

If the processes generate different updates and the modified objects are under the kernel’s control, TightLip applies the original’s update and records a transfer of sensitivity. For example, the kernel transitively marks as sensitive objects such as pipes, UNIX domain sockets, and files. Subsequent reads of these objects by other processes may spawn doppelgangers.

It is important to note that processes will never block indefinitely. If one process times out waiting for the other to reach the barrier, TightLip assumes that the processes have diverged and discards the doppelganger. The kernel will then have to either mark any subsequently modified objects sensitive or invoke the policy module.

Signals are a special kernel-doppelganger interaction since they involve two phases: signal handler registration, which modifies kernel data, and signal delivery, which injects data into the process. Handler registration is managed using barriers and condition variables as other kernel state updates are; only requests from the original are actually registered. However, whenever signals are delivered, both processes must receive the same signals in the same order at the same points in their execution. We discuss signal delivery in Section 7.5.2.

Of course, doppelgangers must also be prevented from modifying non-kernel state such as writing to files or network sockets. Because it may not be possible to proceed with these writes without invoking a disclosure policy and potentially involving the user, modifications of non-kernel state are treated differently. We discuss updates to non-kernel state in Section 7.5.2.

**Doppelganger Inputs**

To reduce the false positive rate TightLip must ensure that sources of divergence are limited to the scrubbed input. For example, both processes must receive the same values for time-of-day requests, receive the same network data, and experience the same signals. Ensuring that reads from kernel state are the same is trivial, given that updates are synchronized. However, preventing non-kernel reads, signal delivery, and thread-interleavings from generating divergence is more challenging.

**Non-kernel reads**

The values returned by non-kernel reads, such as from a file, a network socket, and the processor’s clock, can change over time. For example, consecutive calls to gettimeofday or consecutive reads...
from a socket will each return different data. TightLip must ensure that paired accesses to non-
kernel state return the same value to both the original and doppelganger. This requirement is similar
to the Environment Instruction Assumption ensured by hypervisor-based fault-tolerance [21].

To prevent the original from getting one input and the doppelganger another, TightLip assigns a
producer-consumer buffer to each data source. For each buffer, the original process is the producer
and the doppelganger is the consumer. System calls that use such queues include read, ready, recv,
recvfrom, and gettimeofday.

If the original (producer) makes a read request first, it is satisfied by the external source and the
result is copied into the buffer. If the buffer is full, the original must block until the doppelganger
(consumer) performs a read from the same non-kernel source and consumes the same amount of
data from the buffer. Similarly, if the doppelganger attempts to read from a non-kernel source and
the buffer is empty, it must wait for the original to add data.

The mechanism is altered slightly if the read is from another sensitive source. In this case, the
kernel returns scrubbed buffers to the doppelganger and updates a list of sensitive inputs to the
process. Otherwise, the producer-consumer queue is handled exactly the same as for a non-sensitive
source. As before, neither process will block indefinitely.

Signals

In Section 7.5.2, we explained that signals are a two-phase interaction: a process registers a handler
and the kernel may later deliver a signal. We treat the first phase as a kernel update. Since
modifications to kernel state are synchronized, any signal handler that the original successfully
registers is also registered for the doppelganger.

The TightLip kernel delivers signals to a process as it transitions into user mode. Any signals
intended for a process are added to its signal queue and then moved from the queue to the process’s
stack as it exits kernel space. A process can exit kernel space either because it has finished a system
call or because it had been pre-empted and is scheduled to start executing again.

To prevent divergence, any signals delivered to the doppelganger and original must have the
same content, be delivered in the same order, and must be delivered to the same point in their
execution. If any of these conditions are violated, the processes could stray. TightLip ensures that
signal content and order is identical by copying any signal intended for the original to both the original’s and doppelganger’s signal queue.

Before jumping back into user space, the kernel places pending signals on the first process’s stack. Conceptually, when the process re-enters user space, it handles the signals in order before returning from its system call. The same is true when the second process (whether the doppelganger or original) re-enters user space. For the second process, a further check is needed to ensure that only signals that were delivered to the first are delivered to the second.

In previous sections, we have described how the original and doppelganger must be synchronized when entering system call code in the kernel so that TightLip can detect divergence. Unfortunately, simply synchronizing the entry to system calls between processes is insufficient to ensure that signals are delivered to the same execution state.

This is because some system calls can be interrupted by a signal arriving while the kernel is blocked waiting for an external event to complete the call. In such cases, the kernel delivers the signal to the process and returns an “interrupted” error code (e.g. EINTR in Linux). Interrupting the system call allows the kernel to deliver signals without waiting (potentially forever) for the external event to occur.

Properly written user code that receives an interrupted error code will retry the system call. If TightLip only synchronizes on system call entry-points, retrying an interrupted system call can lead to different system call sequences. Consider the following example taken from the execution of the SSH daemon, sshd, where Process 1 and 2 could be either the doppelganger or original:

- Process 1 (P1) calls write and waits for Process 2 (P2).
- P2 calls write, wakes up P1, completes write, returns to user-mode, calls select, and waits for P1 to call select.
- P1 wakes up and begins to complete write.
- A signal arrives for the original process.
- The kernel puts the signal handler on P1’s stack and sets the return value of P1’s write to EINTR.
Figure 7.2: Signaling that leads to divergence.

- P1 handles the signal, sees a return code of EINTR for write, retries write, and waits for P2 to call write.

In this example, divergence arose because P1’s and P2’s calls to write generated different return values, which led P1 to call write twice. To prevent this, TightLip must ensure that paired system calls generate the same return values. Thus, system call exit-points must be synchronized as well as entry-points. In our example, paired exit-points prevent P2’s write from returning a different return value than P1’s: both P1 and P2 are returned either EINTR or the number of written bytes.

Parallel control flows as well as lock-step system call entry and exit points make it likely that signals will be delivered to the same point in processes’ execution, but they are still not a guarantee. To see why, consider the processes in Figure 7.2. In the example, a user-level thread library uses an alarm signal to pre-empt an application’s threads. When the signal is handled determines how much progress the user-level thread makes. In this case, it determines the order in which threads acquire a lock. The problem is that the doppelganger and original have been pre-empted at different instructions, which forces them to handle the same signal in different states. Ideally, the processor would provide a recovery register, which can be decremented each time an instruction is retired; the processor then generates an interrupt once it becomes negative. Unfortunately, the x86 architecture does not support such a register.

Even without a recovery register, TightLip can still limit the likelihood of divergence by deferring signal delivery until the processes reach a synchronization point. Most programs make system calls throughout their execution, providing many opportunities to handle signals. However, for the rare program that does not make any system calls, the kernel cannot wait indefinitely without
compromising program correctness. Thus, the kernel can defer delivering signals on pre-emption re-entry only a finite number of times. In our limited experience with our prototype kernel, we have not seen process divergence due to signal delivery.

**Threads**

Managing multi-threaded processes requires two additional mechanisms. First, the kernel must pair doppelganger and original threads entering and exiting the kernel. Second, the kernel must ensure that synchronization resources are acquired in the same order for both processes. Assuming parallel control flows, if control is transferred between threads along system calls and thread primitives such as lock/unlock pairs, then TightLip can guarantee that the original and doppelganger threads will enter and exit the kernel at the same points.

**Updates to Non-kernel State**

The last process interactions to be regulated are updates to non-kernel state. As with other system calls, these updates are synchronized between the processes using barriers and condition variables. The difference between these modifications and those to kernel state is that TightLip does not automatically apply the original’s update and return the result to both processes. TightLip’s behavior depends on whether the original and the doppelganger have generated the same updates.

**Handling Potential Leaks**

If both processes generate the same update, then TightLip assumes that the update does not depend on the sensitive input and that releasing it will not compromise confidentiality. The kernel applies the update, returns the result, and takes no further action.

If the updates differ and are to an object outside of the kernel’s control, TightLip assumes that a breach is about to occur and queries the disclosure policy module. Our prototype currently supports several disclosure policies: do nothing (allow the potentially sensitive data to pass), disable writes to the network (the system call returns an error), send the doppelganger output instead of the original’s, terminate the process, and swap the doppelganger for the original process.
Swapping

If the user chooses to swap in the doppelganger, the kernel sets the original’s child processes’ parent to the doppelganger, discards the original, and associates the original’s process identifier with the doppelganger’s process state. While the swap is in-progress, both processes must be removed from the CPU ready queue. This allows related helper processes to make more progress than they might have otherwise, which can affect the execution of the swapped-in process in subtle but not incorrect ways. We will describe an example of such behavior in Section 7.7.1.

Swapped-in processes require an extra mechanism to run the doppelganger efficiently and safely. For each swapped-in process, TightLip maintains a fixed-size list of open files inherited from the doppelganger. Anytime the swapped-in process attempts to read from a sensitive file, the kernel checks whether the file is on the list. If it is, TightLip knows that the process had previously received scrubbed data from the file and returns more scrubbed data. If the file is not on the list and the file is sensitive, TightLip spawns a new doppelganger.

These lists are an optimization to avoid spawning doppelgangers unnecessarily. Particularly for large files that require multiple reads, spawning a new doppelganger for every sensitive read can lead to poor performance. Importantly, leaving files off of the list can only hurt performance and will never affect correctness or compromise confidentiality. Because of this guarantee, TightLip can remove any write restrictions on the swapped-in process since its internal state is guaranteed to be untainted.

Unfortunately, swapping is not without risk. In some cases, writing the doppelganger’s buffer to the network and keeping the doppelganger around to monitor the original may be the best option. For example, the user may want the original to write sensitive data to a local file even if it should not write it to the network. However, maintaining both processes incurs some overhead and non-sensitive writes would still be identical for both the original and the swapped-in process with very high probability.

Furthermore, the doppelganger can stray from the original in unpredictable ways. This is similar to the uncertainty generated by failure-oblivious computing [102]. To reduce this risk, TightLip can monitor internal divergence in addition to external divergence. External symptoms of straying are obvious—when the doppelganger generates different sequences of system calls or uses different
arguments. Less obvious may be if the scrubbed data or some other input silently shifts the process's control flow. Straying of this form may not generate external symptoms, but can still leave the doppelganger in a different execution state than the original.

We believe that this kind of divergence will be rare for applications such as file servers, web servers, and peer-to-peer clients; these processes will read a sensitive file, encode its contents, and write the result to the network. Afterward, the doppelganger and original will return to the same state after the network write. In other cases, divergence will likely manifest itself as a different sequence of system calls or a crash [85].

For additional safety, TightLip can take advantage of common processor performance counters, such as those offered by the Pentium4 [114] to detect internal divergence. If the number of instructions, number of branches taken and mix of loads and stores are sufficiently similar, then it is unlikely that the scrubbed input affected the doppelganger’s control flow. TightLip can use these values to measure the likelihood that the doppelganger and original are in the same execution state and relay this information to the user.

7.5.3 Example: Secure Copy (scp)

To demonstrate the design of the TightLip kernel, it is useful to step through an example of copying a sensitive file from a TightLip-enabled remote host via the secure copy utility, scp.

Secure copy requests are accepted by an SSH daemon, ssdh, running on the remote host. After authenticating the requester, ssdh forks a child process, shell, which runs under the uid of the authenticated user and will transfer encrypted file data directly to the requester via a network socket, nsck. shell creates a child process of its own, worker, which reads the requested data from the file system and writes it to a UNIX domain socket, dsock, connecting shell and worker.

As soon as worker attempts to read a sensitive file, the kernel spawns a doppelganger, D(worker). Once worker and D(worker) have returned from their respective reads, they both try to write to dsock. Since dsock is under the kernel’s control, the actual file data from worker is buffered and dsock is transitively marked sensitive. shell, meanwhile, selects on dsock and is woken up when there is data available for reading.

When shell attempts to read from dsock (which is now sensitive), the kernel forks another doppel-
ganger, D(shell), and returns the actual buffer content (sensitive file data) to shell and scrubbed data to D(shell). shell and D(shell) both encrypt their data and attempt to write the result to nsock. Since their output buffers are different, the breach is detected. By default, the kernel writes D(shell)’s encrypted scrubbed data to nsock, sets the parent process of worker and D(worker) to D(shell), and swaps in D(shell) for shell.

7.5.4 Future Work

Though TightLip supports most interactions between doppelgangers and the operating system, there is still some work to be done. For example, we currently do not support communication over shared memory. TightLip could intercept on individual loads and stores to shared memory by setting the page permissions to read-only. Though this prevents sensitive data from passing freely, it also generate a page fault on every access of the shared pages.

In addition, TightLip currently lacks a mechanism to prevent a misconfigured process from overwriting sensitive data. Our design targets data confidentiality, but does not address data integrity. However, it is easy to imagine integrating integrity checks with our current design. For example, anytime a process attempts to write to a sensitive file, TightLip could invoke the policy module, as it currently does for network socket writes.

Finally, we believe that it will be possible to reduce the memory consumed by a long-lived doppelganger by periodically comparing its memory pages to the original’s. This would make using the doppelganger solely to generate untainted network traffic—as opposed to swapping it in for the original—more attractive.

Though doppelgangers will copy-on-write memory pages as they execute, many of those pages may still be identical to the original’s. This would be true for pages that only receive updates that are independent of the scrubbed input. These pages could be remarked copy-on-write and shared anew by the two processes.

Furthermore, even if a page initially contained bytes that depended on the scrubbed input, over time those bytes may be overwritten with non-sensitive values. These pages could also be recovered. Carried to its logical conclusion, if all memory pages of the original and doppelganger converged, then the doppelganger could be discarded altogether. We may be able to apply the
memory consolidation techniques used in the VMware hypervisor [124] to this problem and intend to explore these mechanisms and others in our future work.

7.6 Implementation

Our TightLip prototype consists of several hundred lines of C code scattered throughout the Linux 2.6.13 kernel. We currently support signals, inter-process communication via pipes, UNIX domain sockets, and graphical user interfaces. Most of the code deals with monitoring doppelgänger execution, but we also made minor modifications to the ext3 file system to store persistent sensitivity labels.

7.6.1 File Systems

Sensitivity is currently represented as a single bit co-located on-disk with file objects. If more complex classifications become necessary, using one bit could be extended to multiple bits. To query sensitivity, we added a predicate to the kernel file object that returns the sensitivity status of any file, socket, and pipe. TightLip currently only supports sensitivity in the ext3 file system, though this implementation is backwards-compatible with existing ext3 partitions. Adding sensitivity to future file systems should be straightforward since manipulating the sensitivity bit in on-disk ext3 inodes only required an extra three lines of code.

Our prototype also provides a new privileged system call to manage sensitivity from user-space. The system call can be used to read, set, or clear the sensitivity of a given file. This is used by TightLip diagnostics and by a utility for setting sensitivity by hand.

7.6.2 Data Structures

Our prototype augments several existing Linux data structures and adds one new one, called a completion structure. Completion structures buffer the results of an invoked kernel function. This allows TightLip to apply an update or receive a value from a non-kernel source once, but pass on the result to both the original and doppelganger. Minimally, completion structures consist of arguments to a function and its return value. They may also contain instructions for the receiving process, such as a divergence notification or instructions to terminate.
TightLiP also required several modifications to the Linux task structure. These additions allow the kernel to map doppelgangers to and from originals, synchronize their actions, and pass messages between them. The task structure of the original process also stores a list of buffers corresponding to kernel function calls such as bind, accept, and read. Finally, all process structures contain a list of at most 10 open sensitive files from which scrubbed data should be returned. Once a sensitive file is closed, it is removed from this list.

### 7.6.3 System Calls

System call entry and exit barriers are crucial for detecting and preventing divergence. For example, correctly implementing the exit system call requires that peers synchronize in the kernel to atomically remove any mutual dependencies between them. We have inserted barriers in almost all implemented system calls. In the future, we may be able to relax these constraints and eliminate some unnecessary barriers.

We began implementing TightLiP by modifying read system calls for files and network sockets. Next, we modified the write system call to compare the outputs of the original and the doppelganger. The prototype allows invocation of a custom policy module when TightLiP determines that a process is attempting to write sensitive data. Supported policies include allowing the sensitive data to be written, killing the process, closing the file/socket, writing the output of the doppelganger, and swapping the doppelganger for the original process.

After read and write calls, we added support for reads and modifications of kernel state, including all of the socket system calls. We have instrumented most, but not all relevant system calls. Linux currently offers more than 290 system calls, of which we have modified 28.

### 7.6.4 Process Swapping

TightLiP implements process swapping in several stages. First, it synchronizes the processes using a barrier. Then the original process notifies the doppelganger that swapping should take place. The doppelganger receives the message and exchanges its process identifier with the original’s. To do this requires unregistering both processes from the global process table and then re-registering them under the exchanged identifiers. The doppelganger must then purge any pointers to the original
process's task structure.

Once the doppelganger has finished cleaning up, it acknowledges the original's notification. After receiving this acknowledgment, the original removes any of its state that depends on the doppelganger and sets its parent to the `init` process. This avoids a child death signal from being delivered to its actual parent. The original also re-parents all of its children to the swapped-in doppelganger. Once these updates are in place, the original safely exits.

### 7.6.5 Future Implementation Work

There are still several features of our design that remain unimplemented. The major goal of the current prototype has been to evaluate our design by running several key applications such as a web server, NFS server, and sshd server. We are currently working on support for multi-threaded applications. Our focus on single-threaded applications, pipes, UNIX domain sockets, files, and network sockets has given us valuable experience with many of the core mechanisms of TightLip and we look forward to a complete environment in the very near future.

### 7.7 Evaluation

In this section we describe an evaluation of our TightLip prototype using a set of data transfer micro-benchmarks and SpecWeb99. Our goal was to examine how TightLip affects data transfer time, resource requirements, and application saturation throughput.

We used several unmodified server applications: Apache-1.3.34, NFS server 2.2beta47-20, and sshd-3.8. Each of these applications is structured differently, leading to unique interactions with the kernel. Apache runs as a collective of worker processes that are created on demand and destroyed when idle for a given period. The NFS server is a single-threaded, event-driven process that uses signals to handle concurrent requests.

sshd forks a shell process to represent the user requesting a connection. The shell process serves data transfer requests by forking a worker process to fetch files from the file system. The worker sends the data to the shell process using a UNIX domain socket, and the shell process encrypts the data and sends it over the network to the client. All sshd forked processes belonging to the same session are destroyed when the client closes the connection.
All experiments ran on a Dell Precision 8300 workstation with a single 3.0 GHz Pentium IV processor and 1GB RAM. We ran all client applications on an identical machine connected to the TightLip host via a closed 100Mbs LAN. All graphs report averages together with a 95% confidence interval obtained from 10 runs of each experiment. It should be noted that we did not detect any divergence prior to the network write for any applications during our experiments.

7.7.1 Application Micro-benchmarks

In this set of experiments we examined TightLip’s impact on several data transfer applications. We chose these applications because they are typical of those likely to inadvertently leak data, as exemplified by the motivating Kazaa, web server, and distributed file system misconfigurations [54, 74, 99, 131]. Our methodology was simple; each experiment consisted of a single client making 100 consecutive requests for 100 different files, all of the same size. As soon as one request finished, the client immediately made another.

For each trial, we examined four TightLip configurations. To capture baseline performance, each server initially ran with no sensitive files. The server simply read from the file system, encoded the files’ contents, and returned the results over the network.

Next, we ran the servers with all files marked sensitive and applied three more policies. The continuous policy created a doppelganger for each process that read sensitive data and ran the doppelganger alongside the original until the original exited. Subsequent requests to the original process were also processed by the doppelganger.

The swap policy followed the continuous policy, but swapped in the doppelganger for the original after each network write. If the swapped-in process accessed sensitive data again, a new doppelganger was created and swapped in after the next write.

The optimized swap policy remembered if a process had been swapped in. This allowed TightLip to avoid creating doppelgangers when the swapped process attempted to further read from the same sensitive source; the system could return scrubbed data without creating a new doppelganger.

Figure 7.3, Figure 7.4, and Figure 7.5 show the relative transfer times for the above applications when clients fetched sensitive files of varying sizes.

Note that the cost of the additional context switches TightLip requires to synchronize the origi-
Figure 7.3: Apache relative transfer time.

Figure 7.4: NFS relative transfer time.

inal and doppelganger may be high relative to the baseline transfer time for smaller files. This phenomenon is most noticeable for the NFS server in Figure 7.4, where fetching files of size 1K and 4K was 30% and 25% more expensive, respectively, than fetching non-sensitive files. As file size increases, data transfer began to dominate the context switch overhead induced by TightLip; the NFS server running under all policies transferred 256KB within 10% of the baseline time.

Figure 7.3 shows that our Apache web server was the least affected by the TightLip. The overhead under all three policies was within 5% of the baseline, with continuous execution being slightly more expensive than the other two. This result can be explained by the fact that fetching static files from the web server was I/O bound and required little CPU time. Continuous execution was slightly more expensive, since the original and the doppelganger both parsed every client request.

Figure 7.5 shows that the overhead of using doppelgangers for sshd was within 10% of the baseline for most cases. This was initially surprising, since the original and doppelganger performed encryption on the output concurrently. However, the overhead of performing extra symmetric encryption was low and masked by the more dominant cost of I/O.
The swap policy performed better than the continuous execution policy for Apache and NFS. This result was expected since process swapping reduces the overhead of running a doppelganger. The benefit from process swapping was application-dependent though, as the time spent swapping the doppelganger for the original sometimes outweighed the overhead incurred by running the doppelganger—while swapping took place, the process was blocked and could not make any progress. Transferring 4K size files from sshd illustrated this point: sshd was almost done transferring all of its data after the first write to the network. Swapping the doppelganger for the original only delayed completion of the request.

To our surprise, the swapping policy applied to sshd actually reduced transfer times for 16K and 64K files. The reason for this behavior was that during swapping, the sshd shell process blocked and could not consume data from the UNIX domain socket. However, the worker process continued to feed data to the socket, which increased the amount of data the shell process found on its next read.

Since the shell process had a larger read buffer than the worker process, swapping caused the shell process to perform larger reads and, as a result, fewer network writes relative to not swapping. Performing fewer system calls improved the transfer time observed by the client.
swapping decreased as file size increased since the fixed-size buffer of the UNIX domain socket forced worker processes to block if the socket was full.

The optimized swap policy had the best overall performance among all three policies. Since all servers perform repeated reads from the same sensitive source, creating doppelgangers after every read was unnecessary. Even though this policy often improved performance, it did not apply in all cases. The policy assumed that sensitive writes depended on all sensitive sources that a process had opened. Thus, future reads from these sensitive sources always produced scrubbed data.

Doppelgangers affected memory usage as well as response time. Table 7.2 shows the average total number of extra memory pages allocated while running the entire benchmark. Each cell represents the additional number of pages created during the transfer of all 600 files (133MB).

We observed that the server applications behave differently under our policies. The best memory policy for Apache and nsd was continuous execution since for both servers’ process executes until the end of the benchmark. For these two servers any other policy increased the number of doppelgangers created and required more page allocations. Since an sshd process only executes for the duration of a single file transfer, continuous execution was not as good as swap-optimized execution. For all three servers, the swap policy produced the most page allocations, since it created more doppelgangers.

Overall, our micro-benchmark results suggest that TightLip has low impact on data transfer applications. The overhead depends on the policy used to deal with sensitive writes. In most cases the overhead was within 5%, and it never exceeded 30%. Even with doppelgangers running continuously, TightLip outperformed prior taint-checking approaches by many orders of magnitude. For example, Apache running under TaintCheck and serving 10KB files is nearly 15 times slower than an unmodified server. For 1KB files, it is 25 times slower [90]. Thus, even in the worst case, using doppelgangers provides a significant performance improvement for data transfer applications.

### 7.7.2 Web Server Performance

Our final set of experiments used the SpecWeb99 benchmark on an Apache web server running on a TightLip machine. We used two configurations for these experiments—no sensitive files and continuous execution with all files marked sensitive. Since the benchmark verified the integrity of every file, we configured TightLip to return the data supplied by the original instead of the
scrubbed data supplied by the doppelganger. This modification was only for test purposes, so that we could run the benchmark over our kernel. Even with this modification it was impossible to use SpecWeb99 on Apache with process swapping, since we could not completely eliminate the effect of data scrubbing; the swapped-in doppelgangers still had some scrubbed data in their buffers.

We configured SpecWeb99 to request static content of varying sizes. Figure 7.6 shows the server throughput as a function of the number of clients, and Figure 7.7 shows the response time. Our results show that the overhead of handling sensitive files was within 5%. The above graphs show that the saturation point for both configurations was in the range of 110–130 clients. These results further demonstrate that doppelgangers can provide privacy protection at negligible performance cost.

### 7.8 Related Work

Several recent system designs have observed the trouble that users and organizations have managing their sensitive data [28, 111, 121]. RIFLE [121] and InfoShield [111] both propose new hardware
support for information-flow analysis and enforcement; SANE [28] enforces capabilities in-network. All of these approaches are orthogonal to TightLip. An interesting direction for our future work will be to design interfaces for exporting sensitivity between these layers and TightLip.

A simple way to prevent leaks of sensitive data is to revoke the network write permissions of any process that reads a sensitive file. The problem is that this policy can needlessly punish processes that use the network legitimately after reading sensitive data. For example, virus scanners often read sensitive files and later contact a server for new anti-virus definitions while Google Desktop and other file indexing tools may aggregate local and remote search results.

A number of systems perform information-flow analysis to transitively label memory objects by restricting or modifying application source code. Static solutions compute information-flow at compile time and force programmers to use new programming languages or annotation schemes [87, 112]. Dynamic solutions rely on programming tools or new operating system abstractions [46, 67, 134]. Unlike TightLip, both approaches require modifying or completely rewriting applications.

It is also possible to track sensitivity without access to source code by moving information flow functionality into hardware [32, 40, 116, 121]. The main drawback of this work is the lack of such support in commodity machines. An alternative to hardware-level tracking is software emulation through binary rewriting [31, 39, 90]. The main drawback of this approach is poor performance. Because these systems must interpose on each memory access, applications can run orders of magnitude more slowly. In comparison, TightLip’s use of doppelgangers runs on today’s commodity hardware and introduces modest overhead.

A recent taint checker built into the Xen hypervisor [60] can avoid emulation overhead as long as there are no tainted resident memory pages. The hypervisor tracks taint at a hardware byte granularity and can dynamically switch a virtual machine to emulation mode from virtualized mode once it requires tainted memory to execute. This allows untainted systems to run at normal virtual machine speeds.

While promising, tracking taint at a hardware byte granularity has its own drawbacks. In particular, it forces guest kernels to run in emulation mode whenever they handle tainted kernel memory. The system designers have modified a Linux guest OS to prevent taint from inadvertently infecting the kernel stack, but this does not address taint spread through system calls. For example,
if email files were marked sensitive, the system would remain in emulation mode as long as a user’s email remained in the kernel’s buffer cache. This would impose a significant global performance penalty, harming tainted and untainted processes alike. Furthermore, the tainted data could remain in the buffer cache long after the tainted process that placed it there had exited.

TightLiP’s need to limit the sources of divergence after scrubbed data has been delivered to the doppelganger is similar to the state synchronization problems of primary/backup fault tolerance [4]. In the seminal primary/backup paper, Alsborg describes a distributed system in which multiple processes run in parallel and must be kept consistent. The primary process answers client requests, but any of the backup processes can be swapped in if the primary fails or to balance load across replicas. Later, Bressoud and Schneider applied this model to a hypervisor running multiple virtual machines [21]. The main difference between doppelgangers and primary/backup fault tolerance is that TightLiP deliberately induces a different state and then tries to eliminate any future sources of divergence. In primary/backup fault tolerance, the goal is to eliminate all sources of divergence.

Doppelgangers also share some characteristics with speculative execution [30, 92]. Both involve “best-effort” processes that can be thrown away if they stray. The key difference is that speculative processes run while the original is blocked, while doppelgangers run in parallel with the original.

### 7.9 Summary and Conclusions

Ensuring accountability for disseminating sensitive information is a very difficult problem, for which current technology does not present a credible solution. In this chapter we looked at the problem from a different angle, and devised a solution that could be applied today to reduce the risks of accidentally disclosing sensitive information. Our solution, TightLiP, is intended to help end users prevent unintended leaks of sensitive data, thus avoiding having the users become accountable for something they did not intend. While not a full solution, it is a step forward in our understanding of the problem. TightLiP also demonstrates an approach which can help reduce the incidence of data leaks, while we work on providing more thorough solutions.

TightLiP helps users define what data is sensitive and who is trusted to see it rather than forcing them to understand or predict how the interactions of their software packages can leak data. TightLiP introduces new operating system objects called doppelganger processes to track sensitivity.
through a system. Doppelgangers are spawned from and run in parallel with an original process that has handled sensitive data. Careful monitoring of doppelganger inputs and outputs allows TightLip to alert users of potential privacy breaches.

Evaluation of the TightLip prototype shows that the overhead of doppelganger processes is modest. Data transfer micro-benchmarks show an order of magnitude better performance than similar taint-flow analysis techniques. SpecWeb99 results show that Apache running on TightLip exhibits a negligible 5% slowdown in request rate and response time compared to an unmodified server environment.
Chapter 8

Application-specific Accountability

The previous two parts of this dissertation examine the problem of accountability in two different contexts. The first part presents a generic state-based framework to reason about the behavior of a network service (Chapter 3), and the second addresses the problem of software acting on behalf of a user and “framing” the user for unauthorized sharing of sensitive information (Chapter 7). This last concluding part poses and answers the following question: *is it possible, and if so how, to leverage the specific properties of an application to make the actions of its constituents accountable?*

To answer this question, we chose an example application of significant importance—a distributed virtual resource economy. Unlike the state-based approach developed in the earlier chapters, the approach presented in this chapter explicitly uses application-specific properties to devise solutions to accountability problems in the context of our example application. The techniques we develop do not require the full state-based accountability tools, but are applicable only to our target application. Our methodology, however, illustrates a potentially generic approach to addressing the accountability needs of specific systems.

8.1 Preliminaries

We now take a moment to describe the foundations and basic axioms of our chosen application.

8.1.1 Virtual Resource Economies

A virtual resource economy consists of a collection of resource *providers* and resource *consumers* engaged in mutual relationships: providers own and control sets of *resources* and offer access to partitions of each resource, *slivers*, to resource consumers. A resource is a good of interest to the rest of the economy. Among all possible resources, we are primarily interested in resources that can be accessed over a computer network. For example, CPU cycles on a remote compute server, storage capacity, network bandwidth, etc. These resources are of particular interest because they can be provisioned rapidly, do not require transportation and local setup, and can be accessed as
soon as they are needed. A virtual resource economy offers novel possibilities for connecting resource providers and consumers that can enable new business models and a new generation of applications and services.

In this thesis we focus on economies in which the interactions among providers and consumers are based on leasing, rather than owning. To lease a virtual resource, a provider and a consumer enter into a contractual relationship for a fixed period of time. The lease contract specifies the parameters of the leased resource sliver and the period of time, term, it is to be used. For example, a sliver could be represented as a number of units of a given resource type, such as CPU cycles, network bandwidth, storage capacity, etc. The lease contract effectively delegates control from the provider to the consumer of a resource sliver for a given term. Lease contracts may also specify additional agreements, e.g., quality of service guarantees. The term of the contract can optionally be extended to ensure that a consumer can access the resource even after the original lease expires.

Leasing is particularly appealing in the context of virtual economies, since it enables consumers on-demand access to a given resource without having to invest in costly infrastructure and maintenance. As a consumer's demand changes over time, more or less resources can be added under its control, providing a cost-effective way to handle load spikes. Leasing can also benefit resource providers by allowing them to offer added services in addition to raw hardware.

Virtual resource economies are of significant practical interest since they have the potential to simplify and optimize the usage of network-accessible electronic resources. The primary challenge of building systems of this kind is the need to ensure the integrity of individual interactions and operations, both to the economy's constituents and to external entities. The integrity of a system of this scale, and the willingness to participate in it, rests on the system's ability to prove the correctness of each transaction. Importantly, each violation must be detectable and traceable back to its perpetrator. In short, a virtual resource economy must exhibit a high degree of accountability to be sustainable.

8.1.2 Identity

A resource economy consists of a large number of autonomous consumers and providers, each potentially under the control of different real-world principals. Consumers and providers are unlikely to have any established trust relationships that would enable them to verify their identities. Be-
fore signing a new contract a trust relationship must be established between both parties to enable them to identify each other and to associate actions related to their new relationship with the corresponding entity. This is an essential requirement and forms the basis for accountability.

A naïve approach to identity management would require that each new participant in a resource economy establish a trust relationship with all other participants in the system. While this approach may work for many small-scale systems or systems with already existing strong identities, it is expensive and does not scale. To support accountability in a virtual economy we need an identity scheme that reduces the number of pairwise trust relationships.

We now present the elements of a distributed identity and trust management scheme that is suitable to be used in the context of a large distributed resource economy. This scheme extends earlier work in this space by Fu et al. in the context of the ShArp project [50].

We start by representing each principal in the system by a signed identity certificate, e.g., an X.509 certificate [61]. The hash of the certificate serves as a globally unique identifier and is associated with the holder of the corresponding private key. A certificate is signed by an identity provider. An identity provider is an entity entrusted to authenticate the identities of a collection of principals, e.g., all students within a university, all employees within a company, etc. An identity provider can have its identity certified by another provider following the same recursive scheme. Importantly, each principal can be its own identity provider. In this case, the actor’s identity certificate is self-signed.

If two actors share a common identity provider, e.g., both actors represent students in the same university, they can already verify each other’s identity and can enter into a contractual relationship. However, if two actors do not share a common identity provider, they will need to establish an out-of-band relationship, through which they exchange their own certificates and associate (for themselves only) the received certificate with the identity of the actor that it represents. This is a critical step and it may use any form of authentication or additional information needed by the actors to believe each other’s identity, e.g., credit cards, government issued identification, etc.

The exchange of identity certificates enables actors to authenticate each other and to interact as needed. After the exchange is complete, all communication between these two actors should be digitally signed to ensure integrity and non-repudiation. Since the recipient of a message possesses
the certificate of the sender, it can authenticate the message and associate the actions related to that message with the sender.

While this scheme provides the means for two actors to verify each other’s identity, it depends intrinsically on the existence of out-of-band channels to help with the task when no common identity providers exist. Since out-of-band channels are costly and not always possible to establish, we need to extend the scheme to reduce the instances in which such channels are required. To deal with this problem we modify the interaction protocol in the economy to enable actors to validate each other’s identities transitively. To achieve this we introduce a new actor type—a *broker*—and interpose it on the communication path between providers and consumers. We describe the role of brokers in the next subsection.

### 8.1.3 Brokers

![Figure 8.1: An example virtual resource leasing economy consisting only of a provider and a consumer. The provider and the consumer interact directly and require mutual trust relationships to authenticate each other.](image)

Before we explain the role of brokers and how they help simplify the problem of identity and trust, let us review the interaction between consumers and providers before the introduction of brokers. Without brokers, Figure 8.1, consumers and providers interact directly. The interaction starts with the consumer requesting a new lease and the provider granting the lease. The consumer can request that the lease be extended and the provider can grant the extension. Finally, the consumer can indicate that it no longer needs the lease.
Figure 8.2: An example virtual resource economy consisting of a provider, a consumer, and a broker. Dashed lines indicate pre-existing trust relationships, while the solid line indicates a transitive trust relationship. The consumer first interacts with the broker to obtain a ticket (a promise for resources). The consumer uses the ticket to redeem it with the provider for the actual lease. The ticket contains sufficient information to enable the provider and the consumer to interact without requiring a pre-existing trust relationship between them.

Figure 8.2 shows the resulting interactions after the introduction of a broker. The broker acts as an intermediary between the provider and the consumer. The leasing process is first bootstrapped by having the provider delegate control over the allocation of its resources to the broker for a period of time (a lease). To obtain resources, the consumer first contacts the broker to obtain a ticket—a promise for resources. The consumer then sends its ticket to the site to redeem the resources it has been promised and obtain the actual lease. Lease extensions proceed similarly, the client must first obtain an extension ticket, which it then exchanges for an extended lease. Note that a lease-based virtual resource economy must have at least one actor fulfilling the role of a broker, but there can be an unlimited number of brokers.

To see how brokers help reduce the number of pairwise trust relationships, we go back to Figure 8.2 and note that a provider and a consumer need to interact only if the provider has a valid ticket. The first ticket for a given resource is issued by a resource provider when it delegates to a broker the right to control the allocation of a set of resources. The key to enabling transitive trust is to have the provider endorse the identity certificate of the broker by signing it using its own private key. We refer to the endorsed certificate as a SHaRP certificate. A SHaRP certificate is embedded in the ticket the provider issues to the broker. Since the broker and the resource provider have a trust relationship, the provider can issue the SHaRP certificate, and the broker can verify that its issued SHaRP certificate is valid.

When the broker issues a ticket to the consumer, it uses the SHaRP certificate it received from
the provider to endorse the identity certificate of the consumer. Since the broker and the consumer have a trust relationship, the broker can issue the SHARP certificate and the consumer can verify the broker’s signature and extract the provider’s identity certificate. The consumer can now interact with the resource provider without having an explicit trust relationship.

When the consumer redeems its ticket, the ticket contains the SHARP certificate the broker issued to the service manager. Since this SHARP certificate originates from the site, the site can verify the whole certificate chain and obtain the consumer’s identity certificate. The site can now communicate with the consumer without having an explicit trust relationship.

This level of indirection simplifies the problem of identity and trust management by reducing the needs for all-pairs trust relationships. As long as a producer and consumer have a broker in common, the broker can endorse the identities of the provider and the consumer and enable the communication between the provider and the consumer. An economy with $M$ providers and $N$ consumers would require $M \times N$ trust relationships without brokers and only $M + N$ relationships with the addition of a single broker.

8.1.4 Broker Hierarchies

Brokers can help support complex hierarchical relationships that approximate better a real-world economy. For example, a broker may aggregate resources from multiple providers and offer new types of resources that neither of the providers can offer on its own. Brokers can create multiple competing distribution channels, which ultimately can result in better and cheaper service. See [57] for more information on the role and benefit of brokers.

Figure 8.3 shows an example broker hierarchy. A wholesale broker can acquire the rights over a collection of compute servers directly from a resource provider and delegate control over them to a series of retail brokers. Retail brokers can form complex networks of interactions by which they selectively and strategically delegate control over resources to each other. Next, consumers in need of computational resources contact one or more brokers to obtain rights over the required resources. The brokers delegate control over a subset of the resources to the resource consumers. To approximate a real-world economy even closer, resources may have a price and the economy may use some form of currency to help control the supply and demand for resources.
Figure 8.3: An example virtual resource economy consisting of a hierarchy of brokers. The broker hierarchy supports complex relationships between brokers to emulate closer real-world economies. Importantly, the hierarchy enables transitive trust relationships, thus reducing the complexity of the system and the cost to entry.

The right to control a set of resources moves down the supply chain from providers, through brokers, and stops at consumers. Each hop of the supply chain is a delegation of control, which helps extend trust transitively from resource providers down to consumers—each delegation extends the trust relationship and the validity of identities down to the next level. Each delegation chain starts from a resource provider and involves one or more brokers to terminate at a resource consumer. When a delegation is redeemed the provider and consumer can verify each other’s identity using the delegation chain.

8.1.5 ORCA

The research described in this and the next chapter has been conducted in the context of the ORCA resource management system [64]. ORCA is an implementation of a virtual resource economy based on resource leasing and uses the distributed identity scheme we described in the previous subsection.

Actors in ORCA communicate using the SOAP protocol [123]. Each incoming and outgoing message is digitally signed using the actor’s private key. Broker-issued SHARP certificates help certify the public key of a given actor and enable different actors to interact and verify each other’s messages. Since all communication is signed, and the recipient of a message contains sufficient information to identify the sender, the system’s communication and identity framework offers a solid basis for accountability.
ORCA uses slightly different terminology and we take a moment to define some of the basic terms. Resource providers in ORCA are known as *site authorities* or simply, *sites*. Each site owns and maintains a collection of network-accessible resources and offers them to be used by other actors. A *service manager* in ORCA represents the role of the resource consumer. A service manager is responsible for obtaining the resources for a specific service, e.g., a web farm, or a computationally intensive application. ORCA enables actors to act as brokers by serving as intermediaries between sites and service managers.

Sites, brokers, and service managers are the main actors within an ORCA resource economy. Each of these actor’s actions are driven by the actor’s own priorities and preferences. In essence, during its lifetime, each actor makes choices based on its desire to maximize its utility from participating in the system. We refer to the collection of all actions of an individual actor as its *policy*. ORCA supports pluggable resource policies, which make it possible to customize the actions of a given actor to meet its specific objectives.

### 8.2 Accountability in a Virtual Resource Economy

Interactions within a virtual economy require some form of trust and hence they must be made accountable. While all actors within a virtual economy are expected to conform to the rules of the system, each actor is generally self-interested and may have incentives to misbehave. For example, a site may terminate a lease before its expiration, so that it can assign resources to another client. More sophisticated and harder to detect forms of misbehavior are also possible. Misbehavior is undesirable and the economy should be able to detect, prove, and punish it.

Detecting misbehavior in a virtual resource economy is problematic. An economy consists of a larger number of actors with many different action functions. Each actor can be running a custom policy 8.1.5, which can be very different from the policy of other actors. To enable competition and flexibility the economy must allow actors to define their own action functions. However, as the number of actors grows, making sure that an actor’s policy does not violate the rules of the system is a challenge.

We could use the state-based approach from Chapter 3 to integrate support for accountability in a virtual resource economy. To do this we would need to represent each action function as a
sequence of reads and writes from/to internal state. Each actor can then use the CATS state store to organize its internal state. To verify the behavior of an actor we would need to inspect the actions that it has performed over a period of time. For each action, we would need access to its specification and to the CATS state store so that we can verify the correctness of the action.

While the state-based approach can make a virtual resource economy accountable, we must remember that the state-based approach is intended to be a generic solution. As such, the state-based approach operates on a very abstract and low-level state representation, which we may be able to improve upon if we were to leverage knowledge about the specific application. The state-based approach may also need to address issues, which may not be relevant to a specific system or types of action, e.g., highly concurrent updates to shared state. For example, a major part of the state of a broker is the union of all tickets that it has issued against its inventory, but a ticket issued to a service manager can only be extended or relinquished by that service manager.

While the generic portion of the state-based approach can collect information sufficient to detect misbehavior, full detection requires detailed knowledge about the specification of an actor's action functions. This requirement is easier to meet when the number of unique action functions is relatively small, as in the case of a single storage service (Chapter 5), but it becomes more challenging as the number of services, and the size of their action sets, increases. Verifying the correctness of the whole economy would require access to the specification of all action functions. This requirement is feasible, but it greatly reduces the ability to verify on demand the correctness of an action and increases the window of vulnerability, during which the misbehavior of an actor can propagate unrestrained within the system (Chapter 2).

These arguments speak in favor of an approach that identifies critical operations within a virtual resource economy and decouples them from the rest of the actor-specific activities within the economy. By identifying the underlying principles and invariants behind the common operations, we could make these operations accountable without having to use the full state-based approach. In essence, this application-specific approach is likely to reduce the complexity of the state machines that needs to be verified and the amount of state that needs to be inspected to determine the correctness of an action. The application-specific approach achieves this improvement by focusing only on the portion of state relevant to the given operations and by leveraging application invariants,
which short-circuit the action verification process.

In our analysis of lease-based virtual resource economies we identified the following critical operations:

- **Delegation of resource rights.** Transferring control over a collection of resources from one actor to another is the key operation within a lease-based virtual resource economy. Each transfer is a delegation from one authorized actor to another, which authorizes the recipient to control a given set of resources for a specified period of time. Such delegations enable complex interactions within a virtual economy, bringing it closer to a real-world economy. Since the integrity of this process is of critical importance for the stability of the economy, special care must be taken to ensure that the process is accountable.

- **Currency operations.** Transactions within a virtual economy may involve the exchange of currency for rights over resources. Currency distribution and recharging can be used as policy tools to stimulate specific behavior, e.g., to encourage actors to plan their resource usage over time, thus reducing sharp spikes in resource demand. A virtual economy must ensure the correctness of all currency operations to prevent misbehavior and abuse.

- **Lease management.** A lease is a contract between several parties. The contract has well-defined terms, e.g., amount of resources, and period over which they should be leased. All parties bound by a lease contract should comply with the contract, regardless of the specific action functions that may drive their behavior. Violations of a lease contract should be accountable—violations must be provable and traceable back to their origin.

Resource delegation and currency management are at the heart of a virtual resource economy. Each actor in the economy must perform these operations, regardless of the complexity of its actions. Importantly, these operations are often integrated and executed in the context of other more complex ones. If we were to focus on an actor’s actions as a whole, ensuring the correctness of resource delegation and currency transfers may require custom approaches for each actor. The application-specific approach would allow us to achieve some generality in enforcing accountability in the virtual resource economy. Similar arguments apply to the subject of lease management. If we could identify
critical lease invariants and essential properties, we could validate a range of action functions without needing to have detailed specification for each function.

In this chapter we focus on the first two problems and describe solutions with strong accountability properties. While we have made some progress in the area of accountable lease management, this subject is outside of the scope of this chapter and is part of our future work (Section 9.2).

8.3 Accountable Resource Delegation

Delegation of electronic resources presents an important challenge, not usually encountered when dealing with physical resources: an actor may delegate more resources than it controls at a given point in time. We refer to this problem as oversubscription. While oversubscription may have legitimate uses within a resource economy [50], we are interested in designing a mechanism to detect it and identify who is responsible for it. Such mechanisms can then be used selectively depending on the requirements of the system.

In addition to oversubscribing resources, an actor may misrepresent the nature of the delegated resources. For example, a site may delegate 2 physical hosts each with a single 1GHz CPU, while the broker may represent these hosts to a service manager as having dual 2GHz CPUs. In general, each delegation may modify the characteristics of the delegated resources and it is important that there exist mechanisms to trace the changes to resource characteristics across multiple delegations.

Finally, an actor may produce a “counterfeit” delegation and push it down the delegation chain. Actors receiving a delegation must be able to determine if a delegation, or a previous delegation that it derives from is authentic. Any “counterfeit” delegation must be traceable to its originator.

8.3.1 The SHARP Resource Delegation Model

The problem of accountable resource delegation has been previously studied by Fu et al. [50]. The SHARP model proposed by this work provides the means to delegate resources hierarchically. In this model an actor is given a fixed number of units of a given type, which it can control for a fixed period of time. The actor can subdivide its resources by extracting one or more units of the same type and delegating control over them to another actor for a possibly different period of time. This process can be invoked recursively, resulting in a sequence of delegations: each sequence starts with
a delegation made by a site authority and finishes with a delegation made to a service manager who consumes the resources described in the last delegation.

\[
\text{<ticket>}
\text{<delegation issuer="site0" holder="site0" type="1" units="100" start="100" end="1000" type="1" />}
\text{<delegation issuer="broker0" holder="broker1" type="1" units="10" start="100" end="1000" type="1" />}
\text{<delegation issuer="broker1" holder="server" type="1" units="10" start="100" end="1000" type="1" />}
\text{<delegation issuer="server" holder="system" type="1" units="10" start="100" end="1000" type="1" />}
\text{</ticket>}
\]

**Figure 8.4:** A SHARP ticket consists of one or more delegation records. Each delegation transfers control over a given number of units for a specified period of time. Each delegation is signed by its issuer. A ticket contains the full path of delegations from a site authority down to a service manager. Note that each ticket starts with a self-signed delegation used to indicate the total amount of available resources at the site.

A delegation chain in SHARP is represented by a ticket. Figure 8.4 shows a sample ticket with four delegations going through four actors: a site, two brokers, and one service manager. To obtain a lease each service manager redeems its ticket with the site authority indicated in the ticket. Sites keep a record of all tickets they receive and organize all tickets derived from the same initial delegation into a tree-like data structure. Each node in the tree represents a specific delegation and two nodes are connected if one delegation (child) has been derived directly from the other (parent).

The basic invariant of the structure is that the sum of all units over a time interval in all children of a given node should not exceed the number of units in the parent over the same interval. If the invariant is violated, then the issuer of the parent delegation has overcommitted its resources. Since all delegations are signed, one can easily prove the violation to a third party by presenting all child delegations and comparing them against the parent delegation. This property makes the SHARP protocol strongly accountable (Chapter 3).

The SHARP model separates allocation (who gets what and for how long), from assignment (what physical resources are used to satisfy a given request). In the SHARP model, sites have detailed knowledge of the underlying physical resources, while brokers operate on a logical view of the physical resources. This is a key feature of the SHARP protocol, which enables sites and brokers to operate semi-independently, preserves their relative autonomy, and reduces the need for communication during the process of resource delegation.

### 8.3.2 Divisible Resources

The SHARP model is a starting point of our work. It provides the basic mechanism to ensure accountable delegation of *non-divisible* resource units. That is, using SHARP it is possible to subdivide
collections of resources consisting of multiple units, into collections of different number of resource units. The granularity of the approach, however, is a single unit—it is impossible to split a unit into smaller sub-units. The reason for this limitation stems from the fact that splitting a single unit produces fragmentation, which makes it difficult and even impossible for a site authority to “pack” allocated resources onto the underlying physical units.

![Figure 8.5: An allocation that causes a problem when using SHARP for divisible resource units. (broker view)](image)

![Figure 8.6: An allocation that causes a problem when using SHARP for divisible resource units. (site view)](image)

To illustrate the problem better, consider the delegations illustrated on Figure 8.5. In this case, a broker is responsible for the allocation of two resource units. The broker receives two requests: each for two sub-units. One of the requests demands 2 x 1/3 of an unit, while the other needs 2 x 2/3. Figure 8.6 shows a possible assignment by the site, which prevents the site from satisfying the second request. Note that, despite all actions by the broker being correct, the site is unable to satisfy the second request.

While the basic SHARP model applies in many cases, there is a very important collection of resources, for which the ability to subdivide a single unit, is essential for ensuring efficient use. One particularly important example is delegating fractions of the resources of a single compute server. As the amount of resources packaged within individual servers are increasing (faster CPUs, more CPU cores, more RAM, etc), the amount of idle resources on a single server is likely to increase. Virtual machine technology [11, 125] now makes it possible to consolidate multiple physical servers into several virtual machines running on a single host to achieve better resource utilization and simpler management. Existing virtual machine technology enables fine-grain partitioning of the underlying host’s resources across multiple virtual machines. For example, a virtual machine can specify a fraction of the host’s CPU, memory, network bandwidth, and disk space.

The primary challenge for dealing with divisible units is preserving location information. Location information is necessary for two reasons. First, to avoid the scenario described in Figure 8.6
and preserve the ability of sites and brokers to make independent choices, which can be reconciled if needed. Second, the individual resources that make up a single virtual machine: CPU cycles, RAM, disk space, network bandwidth, etc., must all refer to the same physical host. This restriction is largely because current technology does not (and most likely will not) support virtual machines whose resources span across several physical hosts; it is currently impossible to create a virtual machine using CPU cycles from one host, and RAM from another.

### 8.3.3 Preserving Location Information

To subdivide a single resource unit, and avoid the problem described earlier, we need to be able to tag the resulting sub-units with information about the resource unit from which they have been derived. In essence, we need to supply sufficient information to enable site authorities to reconstruct the placement choices made by brokers and avoid being unable to satisfy a valid request. One possible solution is to associate an identifier with every single unit. This approach does solve the problem, but it may also unnecessarily restrict a site’s placement policy, since each sub-unit will be bound to exactly one unit, even if several units are partitioned in the same way.

On the other extreme, we can restrict how units can be partitioned in sub-units. For example, a compute server can be divided into exactly 4 virtual machines, each being allocated identical resource shares. Following this scheme, a sub-unit has fixed dimensions and can be placed on any unit from the same resource pool. This approach significantly simplifies the problem of allocating sub-units, but does this at the expense of restricting the choices available to end users. For example, when there is only one type of virtual machine to choose, some users may end up paying for more resources than they need, thus reducing efficiency and resource utilization.

Ideally, we would like to offer a range of choices about how brokers associate location identifiers with units: from a single identifier for all units, to an identifier per unit. A broker would then choose the approach that best fits its overall policy. Sites can also provide incentives to brokers to reduce the number of identifiers and allow for greater flexibility of placement choices, e.g., offer better prices for future resource purchases. Importantly, efficiency is also a concern—it must be possible to issue a ticket for a block of resources efficiently, without generating a prohibitive number of identifiers. A site’s management costs increase as the number of different resource subdivisions
Figure 8.7: Using binpools and split and extract operations for accountable resource delegation.

We next describe a generalization of the original SHARP protocol that supports allocation of divisible resources.

8.3.4 Extended SHARP Resource Delegation Protocol

We start by augmenting each resource unit with a resource vector. A resource vector contains an entry for every resource dimension along which a given unit can be subdivided. All units from the same resource type start with identical resource vectors. The resource vector for a given unit changes over time as the unit is subdivided into sub-units. A binpool then groups units from the same resource type with identical resource vectors. Each binpool has a unique identifier, and contains units that are interchangeable. Binpools also have a validity interval associated with them: a binpool ceases to exist after the end of its validity interval.

Initially, all resource units from a given type come into existence into a single root binpool (binpool 1 on Figure 8.7). To delegate resources from this binpool, an actor can perform two types of operations over the binpool: extract and split. An extract operation creates a new binpool by subtracting a resource vector from the resource vector of all units within a binpool for a given period of time. The resulting binpool has the same number of units as the binpool on which the extract operation was performed (binpools with ids 3 and 4). A split operation modifies the number of units within a binpool instead of their resource vector. A split creates a new binpool with a given number of units and with resource vectors identical to the resource vectors of the binpool from which the
split was performed (binpools with ids 2 and 5). Binpools produced as a result of split and extract operations have a parent-child relationship with the binpool from which they originated.

Multiple split and extract operations can be performed on a given binpool as long as they are either all splits or all extracts. Since splits and extracts do not modify the original binpools, we require that splits and extracts on the same binpool are not mixed. If a split must be applied on a binpool from which resources have been extracted, a new binpool must be first extracted and the resulting binpool can then be split. Similarly, if a binpool from which resources have been split must be the target of an extract operation, a new binpool must be split and then the resulting binpool can become the target of the extract operations. The need for these restrictions will become clear when we discuss the verification algorithm for resource delegation.

**Resource Delegation**

![Resource Delegation Figure](image)

**Figure 8.8:** A resource delegation: 3 units with a resource vector (5,4), and term (30, 60).

Each actor has at least one root binpool that it can subdivide into other binpools using split and extract operations. Each binpool that results from a split or extract can either be the target of future split or extract operations or can be delegated to another actor. Delegating a binpool makes that binpool a root binpool for the receiving actor, i.e., the delegated binpool becomes the source for resources to be transferred to the recipient. This process repeats recursively until one or more binpools are delegated to a service manager. Note that all units within a single delegation must have identical resource vectors.

**Delegation Records**

A resource delegation is expressed in a delegation record. A delegation record references one or
more source binpools to be delegated to a recipient. Each delegated binpool can only be a part of one delegation: using the same binpool in more than one delegation constitutes oversubscription and is a violation of the protocol. A delegation record identifies its issuer, its holder (recipient) and may attach an optional properties list to define additional important characteristics of the delegated resources. The issuer digitally signs the delegation record to ensure its integrity and certify its origin (Figure 8.8).

![Delegation Diagram](image)

**Figure 8.9:** A complete resource delegation: 3 units with a resource vector (5,4), and term (30, 60). Note that the delegation also includes binpools 1 and 2 from which the delegated binpool 3 has been derived.

In addition to each source binpool that directly contains the resources used by a delegation, a delegation record must also include all local predecessors of each source binpool, i.e., all binpools from which the source binpools have been derived. This chain terminates at one or more root binpools (Figure 8.9). Without this information, site authorities cannot reconstruct all split and extract operations an actor performed internally and have incomplete information that prevents them from detecting oversubscription. Including the complete information ensures that even if some delegation records never reach the site authority, all information contained in the ones that do reach the site is sufficient to verify all redeemed delegations.

**Resource Tickets**

A complete resource delegation is expressed in the form of a ticket. A ticket consists of at least one delegation record and a SHARP certificate. The SHARP certificate is created by the issuer for the holder of the delegation and combines the issuer’s SHARP certificate and the holder’s X.509 certificate. To do this, the issuer signs the X.509 certificate and uses the signature, the X.509 certificate, and its own SHARP certificate to produce the holder’s SHARP certificate. Note that an issuer may have more than one SHARP certificate (Figure 8.10).
**Figure 8.10:** To create a **SHARP** certificate for a given actor, the issuer of the **SHARP** certificate signs the actor’s X.509 certificate and produces a new **SHARP** certificate, which consists of its own **SHARP** certificate, the actor’s X.509 certificate and the issuer’s signature on the actor’s X.509 certificate.

Over time an actor may acquire more than one **SHARP** certificate. A **SHARP** certificate is based on the path in the broker hierarchy through which the actor obtained its resources. A **SHARP** certificate enables the holder of a delegation or the site from which the delegation originated to verify the integrity of each delegation step. This is why, when creating a **SHARP** certificate to be used in a resource ticket, the issuer must use a **SHARP** certificate that originates from the site authority that owns the delegated resources. This requirement ensures that both the site authority and the service manager who may eventually receive resources derived from this ticket will be able to communicate with each other.

In addition to the final delegation record, a ticket must also contain the full delegation chain of the resources that the issuer chose to satisfy the request. Namely, a resource ticket contains a delegation record for each delegation that has taken place as the resources originated from the site and trickled down through the broker hierarchy. This information allows the site to reconstruct the delegation decisions performed by other actors prior to the actions of the issuer of the final delegation.

A given issuer may use resources originating from more than one prior delegation. For example, a broker may obtain resources from two different channels and combine the resources into one delegation to a service manager. The issuer must ensure that all possible delegation paths from a root binpool created by a site, to each binpool used to satisfy a request are included in the final ticket. Since resources used in a single delegation may have followed multiple paths in the broker hierarchy, a resource ticket may need to include additional **SHARP** certificates to make sure that all delegation records that make up the resource ticket can be verified.

A ticket represents identical resources originating from a *single* site. A ticket cannot represent multiple resource types or resources originating from multiple sites. If a request demands multiple resource types from a single site, brokers must issue multiple tickets—one for each resource type. Similarly, if a request demands resources from multiple sites, brokers must issue multiple tickets—
one for resources from each site.

**Lease and Ticket Extensions**

As mentioned earlier, an actor may wish to extend the term of a delegation so that it can continue to have control over the resources represented by the delegation for a longer period of time. In general, a lease extension can modify not only the term of the lease, but also the number of units and/or the resource vector of all units.

One way to process a lease extension is to treat the extension as a new request. In this case, the broker allocates new source bin pools with the desired term, units, and resource vector. The new source binpools will have new identifiers and sites will have to determine how to process the lease extension request. In particular, a site must determine if it can preserve resources it has already assigned or it must assign new ones.

For many resource types, e.g., storage, virtual machines, etc., it is important that lease extensions do not result in service interruption and/or loss of state—an application running inside a leased virtual machine should continue making progress while a lease extension is in progress. Therefore, brokers may not always be able to treat ticket extensions as simply new ticket requests—a broker may need to attempt to preserve the location of already allocated resources.

Using our binpool model, if the original and the extended source binpool derive from the same parent binpool, then the lease for the original binpool can be extended “in-place”, since the parent binpool represents identical resources with the same location identifier—any unit from the parent binpool is interchangeable for another unit from the same binpool. However, if the original and the extend source binpools have different parent binpools, then the extension requires a “migration” operation, since the new source binpool derives from a binpool with a different identifier.

If a resource supports a migration operation, i.e., the original instance can be moved from one physical location to another, brokers may have the freedom to “migrate” the resource during an extension operation. If the resource does not support migration then any attempt by a broker to indicate a migration is a violation. Sites can determine if a migration was intended by examining the parent binpool for each source binpool used to satisfy an extension request. To avoid committing a
Figure 8.11: A resource ticket consisting of two delegations (SHARP certificate removed).
violation in such cases, brokers will have to reject extension requests that require migrations.

8.3.5 Assignment Tree and Assignment Forest

If we link each binpool to the binpool from which it was derived, the result is a tree whose structure preserves the history of subdivisions of resources from a given root binpool. We refer to this tree as the assignment tree for resources from a given root binpool. The child nodes of each node are all binpools derived from that node using either split or extract operations. The resulting tree has the following properties:

- A node is not overcommitted over an interval of time, if the sum of the resources contained in its children over that period does not exceed the resources that it contains.

- A node is not overcommitted if it is not overcommitted over any interval of time.

Using these properties, it is easy to verify if a sequence of allocations has resulted in overcommitment. To do this, we need to assemble all delegations for resources of a given type, deriving from a given site. That is, we need to combine the allocation decisions of each broker that has been involved in this process, construct the global tree of binpool subdivisions (Figure 8.12), and verify that both properties listed earlier hold. Note that we are only concerned with tickets redeemed by service managers for resources at the given site. Tickets representing delegation from one broker to another are not part of this process, since they eventually get represented in resource delegations to service managers.

The information contained in each ticket supplies one or more paths from the root of the assignment tree to a leaf. This information is sufficient to reconstruct the different levels of the tree and to detect oversubscription. Since each ticket contains full root to leaf paths, this approach also works, even if some tickets are never redeemed. In such cases it may not be possible to detect overcommitment, if no overcommitment actually takes place. Importantly, if overcommitment is detected, since every delegation is digitally signed, and tickets contain the needed SHARP certificates to verify all signatures, it is possible to produce undeniable evidence to demonstrate the fact that a given broker has misbehaved. That is, the delegation protocol is strongly accountable (Chapter 2).

Note that the portion of an assignment tree at a given broker may be a forest, rather than a single tree. A broker may have obtained several root binpools from multiple brokers, or the resources
obtained from a given broker may refer to multiple source binpools, e.g., broker$_3$ on Figure 8.12. In addition, at a given time a single broker may have more than one root binpools for resources of a given type: one being the current, and the other, covering a period in the future.

Figure 8.12: An assignment tree consists of all resource delegations deriving from resources from a given type and site. The tree can be used to detect and prove overcommitment by a broker.

8.3.6 Implementation

We implemented the extended SHARP protocol as a core library in the ORCA system. Our implementation allows a broker to avoid oversubscribing its inventory accidentally, while a site can detect any intentional oversubscription. The implementation supports both non-divisible and divisible resources and delegation chains of unlimited length. Our implementation consists of approximately 2000 lines of Java code.

The library can operate in two modes: untrusted (all delegations are signed) and trusted (no delegation is signed). Trusted mode is intended to be used for environments with a common trust base, e.g., all actors are owned and controlled by a single entity. In such cases, the integrity of each delegation record is not a concern, but one must still supply the full delegation paths to avoid making unsatisfiable allocations. Untrusted mode, on the other hand, does not assume any trust and requires that each delegation record be digitally signed by its issuer. Only untrusted mode ensures that the delegation process is accountable.

Brokers can use the library to keep track of their inventory binpools and how those binpools are subdivided and delegated to clients. To delegate resources to a client, a broker performs a series of split and/or extract operations and issues a ticket containing one or more binpools. The library
keeps track of the available resources for a given time interval and, by default, throws an exception if an allocation is about to result in oversubscription. In its default mode, the library will never issue an oversubscribed ticket.

As service managers redeem their tickets, a site organizes all tickets with the help of the delegation library and reconstructs the assignment tree for resources originating from that site. Each new ticket specifies at least one root to leaf path and the library verifies the tree invariants for each element of every root to leaf path contained in a ticket. If granting a ticket is about to result in overcommitting the site’s resources, the site can choose to reject the client’s request and alert the site’s operator. An alert specifies the identity of the misbehaving broker and the site’s operator can construct an undeniable proof of the broker’s actions—all tickets that the broker satisfied from the oversubscribed binpool.

When each delegation is digitally signed, sites can also protect themselves against “forged” tickets, i.e., tickets for resources at the site, issued by brokers that do not control any resources at the site. Since a site makes the initial digitally signed delegations to the top-level brokers, all a site must do to protect against forgery is to ensure that each subsequent delegation is also signed.

### 8.4 Virtual Currency

Similarly to a real-world economy, a fully functional virtual economy must use some form of currency to drive economic activity. Currency is generally preferable to other forms of payment, e.g., barter, as it can abstract out the relationships between buyers and sellers without requiring them to identify coinciding needs. In a virtual resource economy currency can be used as a policy tool to induce certain type of behavior. For example, currency could be used to encourage the economy’s participants to spread their demand over time and avoid sharp spikes in demand and shortages of supply [63]—ideally resources should never be idle if demand for them exists.

A virtual economy can be based on real-world money. The economy can be tied in to the existing electronic payment systems and leverage the existing infrastructure. This is an appealing approach but it has several important shortfalls that warrant a discussion.

The first problem of using real money is that once money is spent, the spender permanently relinquishes control over it. This is a desirable property, but it poses an important question for
virtual resource economies: how should actors participating in the economy be funded? If actors run out of money, resources may remain idle, even when demand for them exists. We refer to this problem as *starvation*. In real-world economies, starvation can be avoided by ensuring that every spender also has an income stream to fund future purchases. Income streams in a virtual lease-based resource economy derive from leasing resources to other actors, and not every resource consumer has the means or intends to lease resources to others in an attempt to fund its purchases: providers and consumers perform different, often separate, roles. A resource leasing virtual economy could be designed as a peer-to-peer system in which everyone performs the same roles and is thus required to contribute some resources, but this would essentially reduce the economy to a mere load-balancing tool, significantly constraining its full potential.

We could allow the members of a virtual economy to fund their purchases using real world money, derived from non-leasing activities. This option introduces a number of issues. Since the supply of real-world money is beyond the control of the virtual economy, allowing a potentially unlimited amount of real-world money to be injected in the virtual economy makes it practically impossible to use currency as a resource allocation mechanism; any policy choices that would depend on the distribution of currency in the virtual economy would be highly unreliable. Another problem with real money is that actors may save or *hoard* income over time to increase their spending power in the future, and may use that accumulation to corner the market, manipulate the price of resources in the system, or starve other users. Actors external to the economy, can also take control over resources and manipulate their prices. In addition, money economies are prone to cycles of inflation /deflation caused by fluctuations in the circulating money supply, which exposes the virtual resource economy to the cycles of growth and recession of the real-world economy.

An alternative form of currency, *virtual currency* may be more appropriate to virtual lease-based resource economies. Virtual currency is a type of currency that does not offer real-world purchasing power and is not influenced by real-world currency phenomena. Its use is restricted only to the confines of the virtual economy, although initial currency distributions may be based on real currency deposits, and it also may be possible to exchange virtual currency for real-world money.

While virtual currency differs from real money in several key aspects, it shares an important requirement: an actor should not be allowed to spend more than its budget (*overspend*). Real-
world currency schemes enforce this property at the instance money is being spent, by requiring synchronous verification of the spender’s account balance. This is a required step in the real-world economy, as the losses caused by overspending can be significant and can have far-reaching consequences. In the context of a lease-based virtual economy, however, we could relax the requirements of detecting overspending, since the risks of overspending are not critical—it is generally in the interest of the spender to maintain a good standing as it receives a useful service from being a member of the virtual economy. Importantly, “fake” virtual money can only be spent inside the virtual economy, significantly reducing the potential real-world benefits of overspending.

The asynchronous nature of virtual currency operations and the well-defined notion of overspending enable us to treat the process of managing virtual currency as an accountability problem. In the rest of this section we present the design of virtual currency that offers strong accountability—any overspending is eventually detected and can be traced by to its originator. Any accusations of overspending are provable to a third party.

### 8.4.1 Self-recharging Virtual Currency

Before we go further, we consider some additional properties on the virtual currency scheme that may offer useful benefits to the virtual economy. An important form of virtual currency suitable for a virtual resource economy was proposed by Irwin et al. [63]. The currency, called *credits*, distinguishes itself by providing solutions to both currency problems mentioned in the previous sub-section: starvation and hoarding.

An important property of credits is that they are *self-recharging*, i.e., the purchasing power of each spent currency unit is restored back to the spender after some delay, *recharge interval*. In effect, the credits self-recharge, automatically replenishing the spender’s budget. Importantly, the receiver of a spent credit unit has possession over the unit only for the duration of the recharge interval. If the same credits are spent again, the recharge time will not be affected. In essence, each subsequent recipient of the credits has control over them for a shorter period of time. The recharge interval is a critical parameter in an economy based on self-recharging currency.

The self-recharging property mitigates the problem of starvation, since it ensures that an actor’s budget is not going to be zero for prolonged periods of time. The self-recharging nature of credits
also bounds the amount of money that an actor can hoard. If an actor never receives money from other actors, its budget will never increase above its initial allocation. If an actor receives money from other actors, it is encouraged to spend it before the recharge interval to avoid loosing it.

8.4.2 Accountable Virtual Currency Design

Self-interested actors may lie, cheat, or steal to maximize their utility, which creates challenges for dependable currency management. For example, actors may attempt to spend currency they do not possess, or spend credits before they recharge. It is important that the design of the virtual currency protects against these threats by ensuring that currency operations are accountable. One solution is to coordinate all currency transactions through a trusted banking service. Such a solution requires a high level of centralization, which limits scalability and increases costs.

Credits offer a simple and enforceable decentralized alternative to cash currency, assuming reasonably synchronized clocks, which are prerequisite to any lease scheduling scheme. Credits have an inherent verifiable time element: recharges must be sufficiently delayed, and transfers have an expiration time. Our approach leverages this advantage and simplifies currency management by enabling local currency actions without synchronous policing. Instead, currency actions are accountable and are subject to certified audit after the fact. To this end we leverage the accountability properties of resource delegation (Section 8.3) to make credit transfers accountable: if an actor misrepresents its credits holdings and exceeds its budget, an auditor may detect any misbehavior and construct undeniable cryptographic proof that is verifiable by any third party.

We model self-recharging currency as a resource lease. We define a credit note to be a resource ticket for \( c \) units of resources of a special resource type: credits. A credit note is a resource lease consisting of a sequence of resource delegations. Resource delegations belong to two groups: distribution delegations describe the sequences of budget sub-allocations, while spending delegations describe the times, within a single recharge interval, that the currency was spent. The primary difference between both groups is the term for each delegation. Spending terms are smaller or equal to the recharge interval, while the distribution terms are greater than the recharge interval. All distribution and spending transactions are effectively time-bound delegations: each actor is given a currency budget for some period of time (e.g., membership in the economy), while spent currency
is only valid during the recharge interval. In essence, currency is just another resource type in the virtual resource economy.

Unlike other resources, which originate from site authorities, currency originates from one or more trusted central banks. A central bank is trusted to allocate currency budgets to actors in the system. Like other resources in the economy, currency can be distributed to each actor either directly or through intermediaries. Direct distribution requires that the bank and each actor can authenticate each other. Indirect distribution requires less assumptions and is more practical. For example, the bank can distribute money to a university, which can then subdivide it to its different departments. A department subdivides its budget among the various research projects taking place inside it. Finally, a project’s lead can allocate a project’s budget among all researchers working on the project. This hierarchical distribution parallels the process of resource delegation and can be modeled and implemented in exactly the same way.

While the banking service makes the initial budget allocations, actors can recharge their currency locally without having to involve the bank. Importantly, building the currency infrastructure on top of resource delegations, makes it possible to remove the bank from the need to arbitrate each currency transfer. These properties make the virtual currency decentralized and more scalable—actors verify locally individual transfers to ensure their correctness. Since this process depends on due diligence, the central bank, and other authorized entities may audit selectively currency transfers to ensure that actors do not overspend their budgets.

A consumer bids with its credits by delegating a new credit note for a subset of its credits, setting the term to the recharge time, and transferring the new credit note to a broker. The start time of the delegated credit note is a timestamp for the bid. The broker may inspect the credit note to determine that it is valid. When the consumer’s credit note expires, it can no longer generate valid bids until the bank issues it a new credit note, possibly adjusting the consumer’s credit allotment according to some global policy. The term of notes issued by the bank is chosen to balance stability and overhead with agility of the global budget policy.

When a buyer bids with a valid credit note, it delegates control of the bid credits to the broker. The broker may spend the credits at any time by delegating a new credit note to another supplier. The credit note’s term represents the expiration time for the note. As described in Section 8.4.1,
credit expiration prevents brokers from accumulating credits, and preserves the balance of credits in the system without allowing brokers to go into deficit or requiring them to hold working capital (after initial bootstrapping of a broker network).

### 8.4.3 Auditing

Following the scheme we described, bid credits pass up a chain of intermediary brokers and accumulate at the site providers, where the purchased resources reside and where the leased resource claims originate. Similarly, the resource tickets pass down the chain of brokers to arrive at service managers, who redeem them back to the sites for leases on the purchased resources as described in Section 8.3.4. Thus all tickets for resources at a given site, and all notes for credits spent to obtain resources, ultimately accumulate at the sites (Figure 8.13).

It can be seen that the credit notes include all information needed to audit all credits transfers and to detect and prove misbehavior. For example, if any end consumer overspends its credit holdings, or attempts to recharge its credits before their time, this misbehavior is evident in the set of credit notes that it issues. In particular, an auditor, or the central bank, can build the assignment tree (Section 8.3.5) of currency delegations for a given actor and validate that no tree node is overcommitted (overspent). In addition, it is necessary to check that all credit notes issued by an end consumer have term durations that are equal to the recharge time. We assume that clocks are synchronized within some tolerance that is small relative to the granularity of lease terms and

---

**Figure 8.13**: Auditing cycle. A trusted bank service issues initial credit budgets to consumers. Consumers self-recharge their credits and use them to obtain resource tickets through a network of brokers; credit transfers do not involve the bank. Spent credit notes eventually propagate to the originating resource sites, who redeem them to the bank as proof of value offered to the community. The bank or its delegates audit the credit notes to hold participants accountable for their transfers.
the recharge time: participants have an incentive to reject credits from peers whose clocks are fast, since they expire sooner.

Note that auditors need not be trusted, assuming precautions are taken to protect against collusion involving auditors (e.g., random selection). Proofs of misbehavior are undeniable and are verifiable by a third party: and cannot be fabricated. To improve performance, the audits may involve probabilistic sampling and they may run in parallel on multiple auditors. Of course, the audits are not privacy-preserving: all transfers must be exposed to the auditors, although it is possible in principle to encrypt predecessor notes with the banker's public key in order to conceal sensitive pricing information from other participants along the supply chain.

It remains only to consider the incentives for provider sites to expose credit notes to an auditor. The expired credit notes held by suppliers are proof of the value they have provided to the system. We propose that they may be returned voluntarily to the bank for a reward such as cash or a fresh supply of credits for consumers associated with the site.

### 8.4.4 Implementation

We implemented support for self-recharging virtual currency in ORCA. Our implementation consists of two primary classes. The *Credits Note* class represents a credit note and the *Wallet* class stores all credit notes owned by a given actor. The *Credits Note* class is a resource ticket and leverages the existing support in ORCA for resource tickets. Auditing of credit notes is supported by the implementation of the assignment tree structure (See Section 8.3.5). We also augmented ORCA resource request objects to enable actors to attach a credits note with the request.

The *Wallet* class provides support for the basic currency operations. An actor can use an instance of the class to store one or more credit notes. The class enables actors to spend credit notes as needed. To maximize the actor's budget, spending favors credit notes with earlier expiration. Each credit note issued by the *Wallet* class respects the recharge rule—issued notes expire after the recharge interval has elapsed, while new credits become available in the *Wallet* at the end of the recharge interval.

Credit notes originate from a *Bank* actor, which is a special type of an ORCA site authority. Clients of this actor can request their initial budget allocations using the ORCA standard ticket
request protocol. Unlike regular resources, credits are not redeemed by service managers, but by site authorities. As credits accumulate at a given site authority, it submits credits for audit by issuing redeem requests to the Bank actor. These redeem requests do not return any resource back to the site. They are simply used as a means to transport one or more credit notes to the Bank for auditing. This is an efficient approach that leverages the existing ORCA infrastructure.

8.5 Application-specific Accountability

This chapter illustrates how to integrate accountability into an example system using detailed knowledge about the system’s specific properties and invariants. While the problems we address are constrained to the example system and the resulting solutions may not directly apply in other contexts, our solutions demonstrate a common approach to dealing with problems of this type. In particular, the basic principles that guide us in each of the problem contexts are:

**Preserve identity.** Since accountability depends on the ability to identify the actor responsible for a given action, any action that may have an impact on the actor’s state or the state of others needs to be associated with an identity. The association must be undeniable. This is the primary reason why each resource delegation and currency transfer are signed.

**Preserve integrity.** Each data item used to reason about the correctness of an action must be certifiably valid. A piece of data is not suitable for reasoning about accountability if could have been modified once it was produced by the actor initially responsible for it. Digital signatures are a universal solution to dealing with this issue.

**Recreate data structures used to make decisions.** Many decisions are driven by internal data structures. Typically an actor uses a data structure to perform its own bookkeeping and guide its choices. If we could recreate the data structures used by the actor, we could recreate the choices it made and determine their correctness. To do this, each choice that is made visible to the rest of the world must carry some additional information to expose internal choices made by the actor that affect the externally visible choice. For example, a source binpool is a visible choice. To support recreating the internal choices that led to this externally visible choice, a resource delegation also contains the ancestor binpools from which it was derived.

An important issue to consider is that not all choices made by an actor may reach the auditor.
Any information contained in an externally visible choice should not depend on the auditor having seen all previous choices. Every choice should supply sufficient information to integrate into the recreated data structure, without making assumptions of what other choices the auditor has observed. Importantly, if the auditor does not have access to the full set of choices made by an actor, the auditor may fail to detect a violation. For example, a site may fail to detect oversubscription by a broker if some service managers never redeem their tickets.

The application-specific examples in this chapter show that it is possible to design flexible and easy to use and verify accountability approaches using detailed knowledge about the specific application. If we were to solve these accountability problems using a general purpose technique such as the state-based approach from Chapter 3, we would have to examine a much larger set of actions and application state. The application-specific approach allows us to isolate the specific problems form the rest of the actions performed by an actor. The separation reduces the complexity of the problem, while the knowledge of the application properties and invariants provides an important shortcut to integrating accountability into this specific system.

8.6 Summary

In this chapter we explored accountability in the context of a virtual lease-based resource economy. We described a distributed identity scheme that reduces the need for centralized trust and provides a form of strong identities that enable accountability of actions in the economy. We identified several critical activities within a virtual resource economy and described how to make two of them accountable: resource delegation and virtual currency distribution and spending.
Chapter 9

Final Thoughts

This thesis presents an approach to building trustworthy network systems based on integrating system-level support for strong accountability. Accountability provides means to identify unfaithful behavior and associate it with the actor responsible for it. Strong accountability ensures that any accusations can be proven to a third party without making any assumptions about the integrity of the accuser. Accountability ultimately discourages misbehavior and promotes compliance and cooperation.

9.1 Contributions

The hypothesis of this dissertation is that integrating accountability into the design of network services is a practical foundation for building trustworthy systems which are resilient to tampering and abuse.

- Chapter 2 defines the notions of accountability and strong accountability and identifies the core properties of accountable systems. We describe the challenges of building accountable network systems and outline two avenues of research. The first one investigates building accountable systems in an application-independent way. This approach is based on an abstract state-based application model, which is applicable to a range of state-driven services. The other avenue of research explores a parallel track—it focuses on the specifics of an application and uses detailed knowledge of application-specific invariants to integrate support for accountability in a specific system. We also compare and analyze our solution relative to prior and related approaches to building trustworthy systems.

- Chapter 3 describes the core of the state-based approach to building accountable systems. We start off by developing an abstract application model. The model represents an application as a set of action functions, each operating on a set of state variables. An action function may retrieve a number of state variables and can update a set of state variables. In essence, each
action is represented as a sequence of reads and writes to a collection of state variables. All state variables used by a service comprise its internal state. The state-based approach focuses on ensuring that a service maintains its internal state in a way that enables accountability. In particular, the approach provides a methodology to ensure that reads and writes to internal state are accountable. This methodology ensures that any evidence required to enforce the accountability of the service and its clients is preserved and is authentic, i.e., the origin and contents of all state variables are known and cannot be denied, fabricated, or tampered with without detection. State management is independent of application logic. To verify the logic of a specific application, the state-based approach uses application-specific verifiers. A verifier interacts with the state management substrate to retrieve state variables needed to validate the correctness of an action. The strong accountability properties of the state management substrate carry over to each verifier—any detected misbehavior is undeniable.

- Chapter 4 describes additional design choices and an implementation of the state-based methodology. The CATS toolkit presented in this chapter implements full support for the elements of the state-based approach. CATS is a reusable software component that can help services maintain their state in accordance with the state-based methodology. We evaluate the major components of the toolkit and the different cost dimensions. Our evaluation demonstrates that the toolkit introduces a noticeable, but not prohibitive overhead.

- Chapter 5 describes applying the methodology and the CATS toolkit to build a concrete accountable service. The CATS storage service is a simple storage service with support for strong accountability. The service maintains a directory of objects and allows its clients to retrieve, create, and update objects. The service acts as a focal point for its clients to exchange and share information. The design of the service ensures that both the service and its clients are accountable. The service can be challenged to demonstrate that each write is included in its internal state and is visible to all clients. The service can also be asked to prove that read responses return information contained in its state. The service history can also be audited over an interval of time, to ensure that the service does not hide object updates. A client that performs a write cannot deny its responsibility and can be held accountable if the contents of the write happen to be invalid. Our evaluation of the service shows that accountability
comes at a cost but that the cost is acceptable for environments that require strong levels of assurance. The degree of write sharing and object age have the biggest impact on cost and performance—older objects and higher degrees of write sharing are more costly.

- Chapter 6 presents an extension to the core methodology that provides support for services with more complex action functions. We show how to use the new extension to construct an accountable authorization service. The service provides the means to maintain a dynamic authorization policy accountably. We also show how to apply the approach to construct a simple accountable lock service. Finally, we consider the case of a class of services, whose action functions can be verified in a general and automated way.

- Chapter 7 addresses the issue of extending accountability to end users. In particular, this chapter is concerned with the problem of misbehaving software acting on behalf of a user. We consider the case of software disclosing sensitive information without the knowledge and approval of its user. Such actions can expose the user to sanctions as the user may be accountable for any leaked information. Our approach in this context does not ensure strong accountability, but, instead, helps inform the end user when sensitive data are about to leave the system. To do this, we develop a novel operating system abstraction, that helps track the flow of information within the user’s operating system. When the system detects a potential leak, it informs the user, who can determine if the leak should be allowed or prevented.

- Chapter 8 explores the area of application-specific accountability in the context of an example application—a virtual resource economy. We identify key accountability requirements and describe how to make two essential operations accountable. We first show how to extend prior research to ensure accountable delegation of partitionable resources. Next, we address the problem of virtual currency management and make the case for using accountability to ensure the correctness of currency spending. Our choice of virtual currency enables us to model currency spending and allocation as a resource delegation problem. The resulting protocols and implementation leverage the accountable resource delegation protocol. Unlike the work done in the previous chapters, our work in Chapter 8 leverages heavily the basic properties and invariants of the example application. This approach results in very efficient, but potentially, limited in scope solutions.
In summary, we validate our hypothesis by exploring, implementing, applying and evaluating two different approaches to building accountable systems. The first one ignores the specifics of an application and focuses on the common state management operations shared by state-driven applications, while the second one uses detailed knowledge of application-specific properties. The first approach results in a generic, but potentially complex to apply methodology, while the second one produces more streamlined and efficient, but somewhat limited in scope solutions.

9.2 Future Directions

The subject of accountability is complex and fascinating. In this dissertation we attempted to identify the basic questions in the area of building accountable systems and to provide answers to some of them. We hope that our answers are a starting point for more research and exploration. In this section we list some interesting questions and areas of future research.

In this dissertation we described several accountable services: a storage service, an authorization service, and a lock services. These services provide core functionality needed by a large set of distributed systems. To understand the full implications of the state-based approach, and to realize more benefits from it, we must explore applying the approach to other services and applications. It is important to understand what other critical services can be made accountable using the state-based approach. Given a set of core accountable services, we must also study what kinds of applications can be built on top of them. As services are layered one on top of the other, it is also necessary to understand how their accountability properties are affected.

The state-based approach intends to provide support for strong accountability and as such makes minimal trust assumptions. In particular, the security of a system based on the approach depends only on the security of the public key infrastructure—no other component must be trusted. This is a powerful approach and can be applied in a range of environments. The minimal assumptions, however, restrict the tools we can use to design the approach. An interesting area of investigation is to study the relationship of trust assumptions and accountability: what levels and types of accountability can be achieved as we introduce more trusted components and building blocks? Understanding this relationship is critical since not all environments require the highest form of assurance provided by our state-based approach.
The cost of accountability is a concern if we want to integrate accountability into performance-critical systems and applications. One possible way to reduce overhead is to organize actions and information into different tiers depending on how critical their correctness is for the stability of the system. Actions and data that require high levels of assurance of semantic correctness can use the more expensive accountability techniques, while actions with less stringent correctness requirements can proceed using the conventional implementations. Choosing the degree of auditing and granularity of historic snapshots can also reduce the cost. Going further, however, is likely to require introducing additional trusted components.

Secure hardware could help reduce the cost of accountability. A small, provably correct operating system (OS) running on top of secure hardware can be used to maintain the binding between actions and actors. The OS could provide the basic state management operations and ensure the correctness of state maintenance at a lower cost compared to the state-based approach. In particular, the resulting state store will not be vulnerable to freshness attacks. Not having to deal with freshness attacks removes the needs for auditing—a challenge would be sufficient to demonstrate the inclusion property for reads and writes. This approach, however, relies heavily on the assumption that the OS can be implemented without vulnerabilities, and that its code can be inspected and certified. Moreover, anyone whose accountability properties depend on the OS would have to trust the certification body. These assumptions are hard to meet today, but as the need for accountability increases, they may become a viable option.

Closing the gap between end users and software acting on behalf of them is another area of interesting research. When software performs an action, there must be a mechanism to determine if this action was explicitly requested by the user. Failure to prove a user’s involvement or to identify software actions not intended by the user makes accountability less strong, as users and software could frame/blame each other. Ensuring a trusted path between users and software is likely to involve some additional sources of trust. For example, a trusted operating system could certify the inputs received by an application and identify their source, a trusted web browser can certify if input supplied to a form’s field derives from the user or not. Software approaches are likely to require some form of hardware support. Intel’s LaGrande [36] technology is going to provide some of the assurance necessary to establish trusted paths.
Integrating accountability into specific applications by using application-specific techniques is an endless area of research. As the need for accountability becomes stronger, more systems designers would consider integrating some support for accountability into the systems they produce. Application-specific accountability techniques are likely to be developed to serve the needs of specific domains. By analyzing and studying core application principles and invariants, system designers would be able to integrate support for accountability. This thesis identified some of the basic principles behind this process, and future work is likely to standardize and streamline it further.

In our work on virtual resource economies we examined two critical processes, resource delegation, and currency management. A virtual resource economy offers many other opportunities to enforce accountability. In particular, a lease-based resource economy must be able to ensure the validity of leases. A lease is a contract between several parties, and each party must be held accountable for deviations from the contract. Since each party’s actions are driven by custom incentives and policy choices, it is important to identify principles and invariants that are common to all contracts. These invariants can then be used to verify individual leases and detect some set of violations without having detailed knowledge and specification of the action functions of each involved actor. Lease invariants are essential for enforcing accountability in a general way without restricting the actions of the constituents of the economy.

A critical issue in the context of a virtual resource economy, and in many other systems, is the process of collecting evidence needed to verify the validity of an action. Evidence is essential to reason about correctness, but actors may have incentives not to send or to delay critical evidence. In essence, this is an instance of the fair exchange problem [96]—when an actor sends a message to another actor, the sender would like to exchange its message for a signed receipt from the recipient. This problem, unfortunately, cannot be solved without the use of a trusted third party [95]—we need to add some additional layer of trust to ensure that all relevant evidence is collected. Trusted platforms, secure hardware, and more thorough auditing may all be part of the solutions to this problem.
Bibliography


187


188
**Biography**

Aydan Rafet Yurefendi was born on August 9th, 1979 in Radnevo, Bulgaria. He received a Bachelor of Arts in Mathematics and a Bachelor of Arts in Computer Science from *American University in Bulgaria* in Blagoevgrad, Bulgaria on May 12th, 2002 and a Master in Computer Science from *Duke University* in Durham, North Carolina on December 30, 2006. Aydan currently holds the position of a Senior Software Engineer at BlueStripe Software, Research Triangle Park, North Carolina and resides in Durham, North Carolina. He has coauthored several technical papers in peer reviewed conferences and workshops. His research area is distributed systems.