SAFE: A Declarative Trust-Agile System with Linked Credentials

by

Vamsi Thummala

Department of Computer Science
Duke University

Date: ______________________

Approved:

____
Jeff Chase, Supervisor

____
Bruce Maggs

____
Landon Cox

____
Mike Reiter

____
Ilya Baldin

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
2016

Abstract

SAFE: A Declarative Trust-Agile System with Linked Credentials

by

Vamsi Thummala

Department of Computer Science
Duke University

Date: ____________________
Approved:

____
Jeff Chase, Supervisor

____
Bruce Maggs

____
Landon Cox

____
Mike Reiter

____
Ilya Baldin

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
2016

Copyright © 2016 by Vamsi Thummala
All rights reserved except the rights granted by the

Creative Commons Attribution–Noncommercial–ShareAlike 4.0

Abstract

Secure Access For Everyone (SAFE), is an integrated system for managing trust using a logic-based declarative language. Logical trust systems authorize each request by constructing a proof from a context—a set of authenticated logic statements representing credentials and policies issued by various principals in a networked system. A key barrier to practical use of logical trust systems is the problem of managing proof contexts: identifying, validating, and assembling the credentials and policies that are relevant to each trust decision.

SAFE addresses this challenge by (i) proposing a distributed authenticated data repository for storing the credentials and policies; (ii) introducing a programmable credential discovery and assembly layer that generates the appropriate tailored context for a given request. The authenticated data repository is built upon a scalable key-value store with its contents named by secure identifiers and certified by the issuing principal. The SAFE language provides scripting primitives to generate and organize logic sets representing credentials and policies, materialize the logic sets as certificates, and link them to reflect delegation patterns in the application. The authorizer fetches the logic sets on demand, then validates and caches them locally for further use. Upon each request, the authorizer constructs the tailored proof context and provides it to the SAFE inference for certified validation. Delegation-driven credential linking with certified data distribution provides flexible and dynamic policy control within an agile security and trust infrastructure.

We evaluated SAFE by using it to build example applications based on case studies drawn from practice: (i) a secure name service similar to DNS that resolves names across multi-domain federated systems; (ii) a secure proxy shim to support rich access control in a key-value store; (iii) an authorization module for a networked infrastructure-as-a-service system with a federated trust structure (NSF GENI architecture); and (iv) authorization rules for a secure cooperative data analytics service that enables computation on sensitive data in compliance with secrecy constraints. We present empirical evaluation based on these case studies.

To my wife, Sona Rajamani, who believed in me as an engineer and offered her unwavering support throughout my graduate career.

Contents

Abstract **iv**

List of Tables **xi**

List of Figures **xiii**

List of Code Snippets **xviii**

List of Abbreviations and Symbols **xx**

Acknowledgements **xxi**

1 Introduction **1**

1.1 Motivation 4

1.2 Thesis Statement 6

1.3 Overview 8

1.4 Contributions 13

1.5 Organization 16

2 Background **18**

2.1 ABLP-based logics 19

2.2 Languages for Trust Management 22

2.3 Datalog-based logics 25

2.4 More recent logical trust systems 29

2.5 Proof-Carrying Authorization (PCA) 31

2.6 Credential Discovery 32

2.7 Summary 35

3 Logical Trust on the Network 36

3.1 Design Overview 36

3.1.1 Slang 41

3.1.2 SafeSets 42

3.2 End-to-end Trust on the Network 43

3.3 Assumptions 45

4 SAFE Logic (slog) 47

4.1 Running Example 47

4.2 Motivation 48

4.3 Foundations 49

4.4 Complexity 53

4.5 Syntax 54

4.6 Attribute-based Delegation 55

4.7 Types and Constraint Domains 59

4.8 Negation 63

4.9 Safety in slog: Range Restriction 63

4.10 Evaluation Procedure 65

4.11 Limitations 66

4.12 Summary 66

5 Certified Links: Enabling Automated Credential Discovery and Policy Mobility 67

5.1 Explicit Linking of Slogsets 68

5.2 Linking with Constraints 71

5.3 Hybrid Access Control 72

5.4 Organizing Slogsets 73

5.4.1 Docking SlogSets . . . 73
5.4.2 Materializing Slogsets via Templates . . . 74

6 SAFE Language (slang) 78

6.1 Motivation . . . 79
6.2 Syntax . . . 80
6.3 Functional Logic Programming . . . 82
6.4 Context-layer Functions . . . 87
6.5 Policy Templates . . . 89
6.5.1 Endorsement . . . 91
6.5.2 Delegation of Authority . . . 91
6.5.3 Objects and Capabilities . . . 93
6.5.4 Groups . . . 95
6.5.5 Attach Policy . . . 96
6.6 Limitations . . . 98
6.7 Summary . . . 100

7 Policy Expressivity in SAFE 101

7.1 ABLP “for” operator . . . 101
7.2 SDSI’s linked namespaces in SAFE . . . 102
7.3 Expressing RT_0 in Slog . . . 103
7.4 Common Access Control Policies . . . 104

8 Architecture 108

8.1 Runtime . . . 109
8.2 Implementation . . . 111

9 SafeSets: A Secure Metadata Service 114

9.1 Design . . . 115

9.2 Implementation 116

9.3 Safe Expressions (SafeX) 117

10 SAFE Service Integration 122

10.1 Caching Credentials 124

10.2 Managing Certificates 126

10.3 Discussion 128

11 Applications & Evaluation 130

11.1 Microbenchmarks 131

11.2 SafeSets: A Secure Proxy Shim for a Key Value Store 135

11.3 SafeNS: A Secure Name Service 137

11.4 SafeCoda: A Secure Cooperative Data Analytics 141

11.5 SafeGENI: Authorization in Federated IaaS Cloud Service 144

11.5.1 Trust Structure 147

11.5.2 Objects 150

11.5.3 Capabilities and Delegation 153

11.5.4 Refining Capabilities 155

11.5.5 Peering 155

11.5.6 Evaluation 158

12 Summary 162

12.1 Experience with SAFE 162

12.2 Future Work 164

12.3 Conclusion 166

13 Glossary 168

Bibliography 178

Biography 189

List of Tables

4.1 Built-in operators in slog. 56

4.2 Basic object types and collections supported in slog. 60

6.1 Common idioms of slang and slog highlighting the differences from standard logic programming systems [WSTL12]. 83

6.2 Builtin variables in SAFE and slang. The variable `?Selfie` is initialized using `defenv` in a slang program to set the issuer's keypair. The variables `?Self` and `?SelfKey` are computed based on the `?Selfie` value. The variables `?Speaker`, `?Subject`, `?Object`, and `?BearerRef` are passed as default arguments from a SAFE application to the guard function associated with a given request. . . 85

6.3 Context-layer functions in slang. 89

6.4 Policy APIs for slang specified in Scala. These APIs are implemented as trust-logic functional rules in slang. 90

11.1 Microbenchmarks of basic operations in SAFE on a 1kB of payload per certificate. Certificates are signed using 2048-bit RSA keys in the native SafeX format. Hash function is configured as SHA-256. All cryptographic strings are Base64 encoded. The *null* inference is the minimum latency penalty for invoking slog through slang. 133

11.2 Microbenchmarks of SafeSets operations on a 1kB of payload per certificate. Fetch and post measurements involve network latencies over WAN for reading a certificate and writing a slogset from/to SafeSets. Certificates are signed using 2048-bit RSA keys in the native SafeX format. Hash function is configured as SHA-256. All cryptographic strings are Base64 encoded. The attested fetch and post refers to measurements where the SafeSets server signs each response to the client in addition to standard process. 133

11.3 Analysis of programming effort for building declarative trust applications in SAFE. 134

11.4 Simplified trust API for GENI, a federated IaaS cooperative cloud testbed. These APIs are implemented in SafeGENI as trust-logic rules. With SafeGENI each API exposes a RESTful service interface and trust attributes/credentials are collected from the `$BearerRef` environment variable passed along with the request. 149

List of Figures

1.1 Overview of a Trust Management System illustrating the five primary components: authentication, authorization, accountability, credential discovery, and context management. Authentication identifies the principal making the request; authorization verifies whether the principal has authority to access the resource; accountability ensures that actions taken by the guard are auditable and policy compliant; credential discovery gathers the requisite credentials to substantiate a request, and context management supplies the tailored proof context pertaining to a request. 3

1.2 Elements of a Trust Management System. Real-world identities such as users, organizations, and corporations exercise control over software entities, which act as principals speaking for the associated identity. Principals exchange authenticated messages over a network. 8

1.3 A workflow of authorization in a Trust Management System. An issuer controls an object by attaching a policy and then grants a capability on that object for another intermediary principal (delegator), who further delegates the capability to a subject. The subject makes a request to the authorizer by passing a capability token obtained from the intermediary. The authorizer assembles the necessary credentials and object policy from all the concerned principals, builds a proof context, runs a prover to infer the access privileges for the subject, and allow/deny access. 9

1.4 Elements of SAFE, a system for secure multi-domain (multi-principal) applications using logical trust. The application code of each participant handles trust events by invoking user-defined scripts running on a local SAFE interpreter. The scripts produce and/or consume logic-based credentials passed as linked certificates through a shared key-value store. In this example, A issues an explicit delegation (e.g., a group membership or a capability for an object) to B. A passes a link for the delegation to B, who adds the link to a permission set. Then, B issues a request as a client to server C, passing a link to its permission set. C fetches B's credentials through the link, validates and caches them, and passes them to its local logic prover to check access for the request. . 11

3.1 Overview of a design elements of SAFE. The SAFE trust logic (slog) is rooted in datalog. The credential discovery and context management are implemented outside the logic layer—slog—-using a scripting language (slang). The app service provides the slang program and initiates it via an RPC. Slang represents each proof context using an abstraction called slogset. A slogset is a set of authenticated logic statements where each statement is qualified with the identity of a principal that issued the statement. Slang provides slogset templates via libraries that capture common security policies and credentials. Slogsets may be linked together and can be exported as signed certificates into a distributed shared repository (SafeSets). The certificates are encoded using Safe expressions (SafeX), which use logic for transport. A slang program is instantiated by a principal A with two slog templates sharing the same local name `‘‘/abc’’`. The slogsets are then materialized as a SafeX certificates and published to SafeSets. 39

4.1 Descriptive complexity of trust logics. 52

6.1 Overview of functionality of slang. Slang is based on functional logic programming that provides interfaces for interacting with service applications and SafeSets. These APIs include issuing and retrieving certificates, creating slogsets from templates, passing application and environment variables to slog programs, building a tailored proof context per request, and invoking slog with an appropriate proof context. 82

8.1 Server access control using SAFE. The SAFE instance runs as a separate process with a loaded slang program that contains context building procedure, the principal's signing key (`$Self`), and the authorizer's local policies specified in slog. The server application installs slang code in the SAFE process, which registers all the `defguard` APIs for access control checks. Credentials are passed as references to signed logic sets (slogsets) in a shared distributed store (SafeSets). The SAFE process fetches slogsets on demand, validates the signature and speaker, and caches them for use by the slog interpreter. 109

8.2 SafeSets implementation using an actor-based programming model for handling concurrent requests. The SafeSets client via a designated master actor distributes work via a pull based approach, i.e., the workers must request the work rather than the traditional approach where the master pushes the work. The pull-based approach avoids back pressure on the workers. The master is free to assign the work to multiple workers in parallel or re-assign the work in case of a timeout. The worker actors communicate to the SafeSets server nodes via an untrusted channel (over the Internet). The requests on the server are authorized via a configured proxy shim running SAFE inference for write validation. The server nodes form a ring cluster for balancing load. 113

11.1 Cost of inference with varying proof length and degree of backtracking b_d. The latency measurements show that the inference cost scales linearly if b_d is kept low. The plot also shows the overhead of calling slog from slang program is minimal ($< 5\%$). 134

11.2 Performance comparison of issuing a *write* to SafeSets store (write is a post with slogset signing excluded). 136

11.3 Workflow illustrating credential discovery, context building, context caching, and proof validation process for a secure name service, SafeNS, implemented using SAFE. The credentials are issued a priori (step 0) and materialized as slogsets in a shared distributed store (SafeSets). The principal `cs` writes to a slogset owned by `duke` due to a `speaksFor*` delegation capability issued by the owning principal (step 0). The SAFE process starts with a bearer slogset reference (`ICANN-ID`) provided by the client and builds the credential graph by traversing via linked references, and then tailors the context as per the resolver's programming logic written in slang (steps 1-17). The slang runtime invokes the slog interpreter by importing the relevant context. The slog interpreter validates the proof based on local trust anchors and policies, and certifies the response (steps 18-20). 138

11.4 Performance comparison of SafeNS with varying delegation lengths. The delegation length four is the average delegation length of a DNS service. The delegation length of eight is the observed maximum delegation length of naming systems from a sample DNS trace. 139

11.5 Impact of caching on secure naming resolution using SafeNS. The figure on the left shows the sample workload distribution we used to simulate the DNS traces with varying delegations following the lognormal distribution. The figure on the right shows latency measurements plotted against cumulative distribution function. 139

11.6 Overview of GENI components and its trust structure. All participating aggregates and coordinators (IdR, PA, SA) are endorsed by a common trust anchor called the federation root (G). Entities join the federation when they accept the root (G) as the trust anchor and are endorsed by the root. Entities install standard local rules to accept other entities endorsed by the root. The resource providers represent federation-approved cloud sites via aggregates. The trust structure emerges by inference from declarative statements of the participating entities. 145

11.7 SafeGENI credential workflow for a typical resource request ("slice") in GENI. The users and project investigators register to corresponding coordinator entities IdR and PA and issue credentials to SafeSets. Each virtual resource instance ("sliver") is obtained from a single resource provider (via aggregate manager interface) and is bound to a project and slice that have been approved by authorities trusted by the aggregate according to its local policy. The authorities approve users, projects, and slices by issuing credentials. The aggregator guards the standard controls on a slice (`instantiate, start, stop, info`) through SAFE service interface. The requester passes a bearer reference and the aggregator fetches the corresponding credentials and runs SAFE inference to authorize the request. The downstream services that fetch from SafeSets credential store may cache the credentials locally. 148

11.8 Performance comparison of SafeGENI with varying delegation lengths. The delegation length 2 is the average delegation length for the experiment setup with logarithmic delegation chains. The delegation length of 6 is significant due to power law distribution that demonstrates at most six-degrees-of-separation between human connections. 159

11.9 End-to-end authorization costs of SafeGENI with varying delegation lengths. 160

11.10 Delegation patterns of SafeGENI follows a geometric distribution with half of the users are reachable from the PI directly, i.e., within a delegation of length one. 160

11.11 Impact of caching on authorization in SageGENI. The figure shows the latency and throughput measurements plotted against the cumulative distribution function. For this experiment, we set N as 1024 and M as 4096, with delegations distributed according to the geometric distribution as illustrated in Figure 11.10. 161

List of Code Snippets

4.1 Alice access policy based on inference of attributes based on statements by other principals. 58

6.1 length() function for lists expressed in slang. 88
6.2 length() function for slogsets expressed in slang. 88
6.3 Endorse an entity by asserting an attribute. 91
6.4 Endorse an entity by asserting an attribute name-value pair. 91
6.5 Attribute-based delegation. 92
6.6 Delegate unrestricted authority to a subject via `speaksFor`. 92
6.7 Create an object by generating a self-certifying name, assign the subject as the owner, and also attach the standard object policy" 93
6.8 Standard access policy on an object. 94
6.9 Grant privilege ?Priv to subject ?Subject on object ?Object. 94
6.10 Grant privilege on all objects matching a prefix for the given subject or a group. 95
6.11 Grant delegate privilege ?Priv to subject ?Subject on object ?Object. 95
6.12 Check whether a subject is a member of a group. 96
6.13 Find all members in a group. 96
6.14 Create group with standard delegatable capabilities. 97
6.15 Grant membership for ?TargetGroup in ?ParentGroup. 98
6.16 Attach policy to an object. 99

7.1 SDSI's extended names and linking expressed in SAFE. 103

9.1 The slang code illustrates an example policy from SafeGENI (see §11.5), where the issuer `Geni` endorses a subject as a project authority. Upon post, the slogset is materialized as a certificate, which is shown in Code Snippet 9.2. 118
9.2 An example of certificate objects stored in SafeSets corresponding to the slang code from Code Snippet 9.1. Line 9-11 correspond to the slog statements lines 6-8 from Code Snippet 9.1. The other predicates are meta predicates that are filled with defaults. The certificate is encoded in native SafeX format, which makes it readable and navigable (by HMTL rendering the `link` predicates) to understand distributed trust structure. . . 119

10.1 Example client-server-SAFE interaction for the policy listed in Code Snippet 4.1. A client requests read access for a file `https://alice.org/secret/sensitive.pdf`. The application server of `https://alice.org/` authenticates the subject (extracts the subject and speaker from the client certificate, line 1-6) and routes the request to an internal SAFE service proxy `http://safeclouds.org/`. The SAFE proxy fetches the credentials using bearer reference and then cross check whether the subject has access to object (line 8-13). 124

11.1 SafeSets proxy shim using SAFE. 136
11.2 Code run by SafeRegistrar on the behalf of each principal for issusing tags. 143
11.3 Code run by SafeRegistrar on the behalf of each principal for delegating tags. 143
11.4 Code run by GENI federation root (G) to issue an endorsement for one of its coordinators IdR. 150
11.5 Code run by IdR to add the endorsement link issued by Geni root from Code Snippet 11.4. 150
11.6 Code run by IdR to endorse a user as a project investigator (PI). 151
11.7 Code run by SA to create a slice requested by the `?Subject` by passing the `?BearerRef` and `?Object` (project) references. 152
11.8 Example rules demonstrating the capability-based access control implemented in trust-logic. These rules allow a slice owner to delegate control to another user. 154
11.9 Example rules demonstrating the capability-based refinement implemented in trust-logic. These rules allow a delegator to constrain the access privileges permitted under a delegation. 156
11.10 Example of blacklisting a user from peer federation (`guestUser`) from creating a slice in local federation. 157

List of Abbreviations and Symbols

Abbreviations

SAFE	**S**ecure **A**cess **F**or **E**veryone
GENI	**G**lobal **E**nvironment for **N**etwork **I**nnovations
slog	**S**AFE **Log**ic
slang	**S**AFE **Lang**uage
SafeX	**SAFE** e**X**pression
SafeNS	**SAFE N**ame **S**ervice
SafeGENI	**SAFE G**lobal **E**nvironment for **N**etwork **I**nnovations
SafeCoda	**SAFE Co**operative **D**ata **A**nalytics
IaaS	**I**nfrastructure **as a S**ervice
DAG	**D**irected **A**cyclic **G**raph
TCB	**T**rusted **C**omputing **B**ase
DNS	**D**omain **N**ame **S**ervice
IName	Self-certifying Name
IID	Self-certifying Identifier

Acknowledgements

This dissertation could not have been accomplished without the help, support, friendship, and love of some amazing people. Poondla Gopal Reddy, my uncle and my role model, encouraged me from an early age to pursue a career in science and engineering. Kamalakar Karlapalem, my undergraduate advisor, is a wonderful teacher and a mentor. He opened me to the world of research and was crucial in steering me towards the graduate school.

I am deeply thankful to my advisor, Jeff Chase. His vision for how to build the future network systems inspired me to switch my research area about half a way through my graduate school. I learned a lot from him, ranging from how to hold a high bar in one's research to the craft of designing systems. Even though some of my ideas were not well formed initially, he supported me patiently, and allowed me to pursue my interests. He helped me to grow as a researcher. In particular, he taught me how to convey things succinctly and also improve my overall writing skills. His classes are inspiring and always contain nuggets of wisdom. I enjoyed serving as a teaching assistant for his courses. I also cherish the long walks we had discussing about the broader topics.

I am also grateful to my committee. Bruce Maggs, Landox Cox, Mike Reiter, and Ilya Baldin for being flexible with their schedules and providing precious feedback to improve this dissertation. Thanks to Bruce Maggs for getting me interested in the topic of energy-efficient systems. Thanks to Ilya Baldin for the allowing me to lease

the resources from ExoGENI racks.

When I arrived at Duke, Shivnath Babu, immediately welcomed me to undertake research in Database Systems. I am thankful for his time and collaboration.

I have been lucky to teach a class with Owen Astrachan. He is a great teacher, and I learned a lot from him about teaching and managing an undergraduate class.

My collaborators at Microsoft Research, Manoj Syamala and Vivek Narasayya gave me the opportunity to learn about research in industry while working on a SQLVM project.

Beyond research, many friends at Duke made the graduate school journey memorable. In particular, the graduate program coordinator, Marilyn Buttler, is always there to help for any need. She listened to me patiently and helped me with valuable advice. Bing Xie is a wonderful office mate, who read many of my drafts and gave suggestions. In addition, I enjoyed talking about research and other topics with colleagues at Duke including Nedyalkov Borisov, Amre Shikamov, JP Cafaro, Songyun Duan, Harold Lim, Sharath Kumar Raghavendra, Chittaranjan Tripathy, Harish Chandran, Nikhil Gopalkrishnan. I would also like to thank the staff at Duke including Diane Riggs, Melanie Eberhart, and Victor Orlikowski for their support.

Everything I do is built on the unconditional love and support of my family. Thank you mom for allowing me to pursue my graduate studies. I miss you dad. My sister taught me how to treasure good things. Thanks to my brother-in-law for filling in my shoes after I came to Duke.

I am lucky enough to have pretty terrific in-laws. Thanks for your support and for visiting here to take care of the family when needed.

Sona, my wonderful wife and best friend, is a superhero. She sacrificed her interest pursuing higher studies so that I can finish my graduate school. She made my life infinitely better with so many little things every day. Thanks for making me laugh, taking care of me through the worst times, and celebrating all the best times.

Finally, my little daughter, Aditi, brought me tremendous joy and has already taught me how to be better at managing my time. I am so excited to see her grow and be part of her adventures!

~~~~

This dissertation is partially supported by the US National Science Foundation through the GENI Initiative and under NSF grants OCI-1032873, CNS-0910653, and CNS-1330659, and by the State of North Carolina through RENCI. The author also likes to acknowledge the gracious support from Duke Summer Fellowship program, and financial assistance from Sona Rajamani during the final stages of this dissertation.
~~~~

1

Introduction

Trust is paramount to building secure networked systems. For example, consider an electronic voting scenario in which the votes are recorded online using a secure service over the Internet. The participants in this system are the users who choose to vote, an identity registrar that provides a voter ID for the user, and the service provider that administers the online voting service. For the voting to succeed, each of the participants must infer a trust decision: the user must trust that the service provider is not tampering with or exposing the votes; the identity registrar must trust that the user is a valid citizen who is eligible to vote; and the service must trust that the user possesses a valid voter ID, and exercises her voting right "only once" in compliance with the voting policy. For such a scenario, a trust management system is useful in automating and inferring a trust decision for each participant, given some set of beliefs (facts) and policies (rules).

Formally, a trust management system deals with specifying and interpreting security policies, credentials, and relationships among entities in a system to reach a trust decision. In common cases, the trust decision pertains to some form of an authorization. Authorization determines whether a given request (e.g., read, write) for

one or more *objects* (e.g., file, process) is permitted for the requesting *principal*. Policies are sets of rules that describe required conditions to permit access. Credentials are sets of statements about the principals issued by concerned parties delegating the chain of trust. To enforce a given policy, each object in the trust management system is guarded by a reference monitor that uses credentials in conjunction with the request attributes to infer the access control decision.

Authentication is the process of attributing each request or statement to the principal that issues it. Authentication is a prerequisite for authorization: authorization requires a means to identify the principal that issued the request. For example, if a request is signed digitally by a principal and correspondingly verified via the public key of the principal, then we can attribute the request to the principal that possesses the private key.

In addition to authentication and authorization, trust management systems often provide some form of accountability through auditing. Auditing is a process of recording, reviewing, and evaluating past actions taken by the trust management system. For example, each authorization decision taken by the trust management system can be recorded in an audit log along with the policies, user requests, or inputs used to make those decisions.

Authentication, Authorization, and Auditing are widely considered to be the "gold standard" for computer security and trust [Lam04].

This thesis focuses on decentralized federated systems where entities may span different administrative domains and may not be known to the authorizer a priori. The NSF GENI architecture [BCL+14, GEN15] is one example of a federated system (see § 11.5). In such systems, trust relations among entities and security policies must be *agile*: they may change frequently and dynamically as the trust structure evolves. Managing trust relations among entities and ensuring that the *right* credentials are issued and processed is paramount to the security of the whole system.

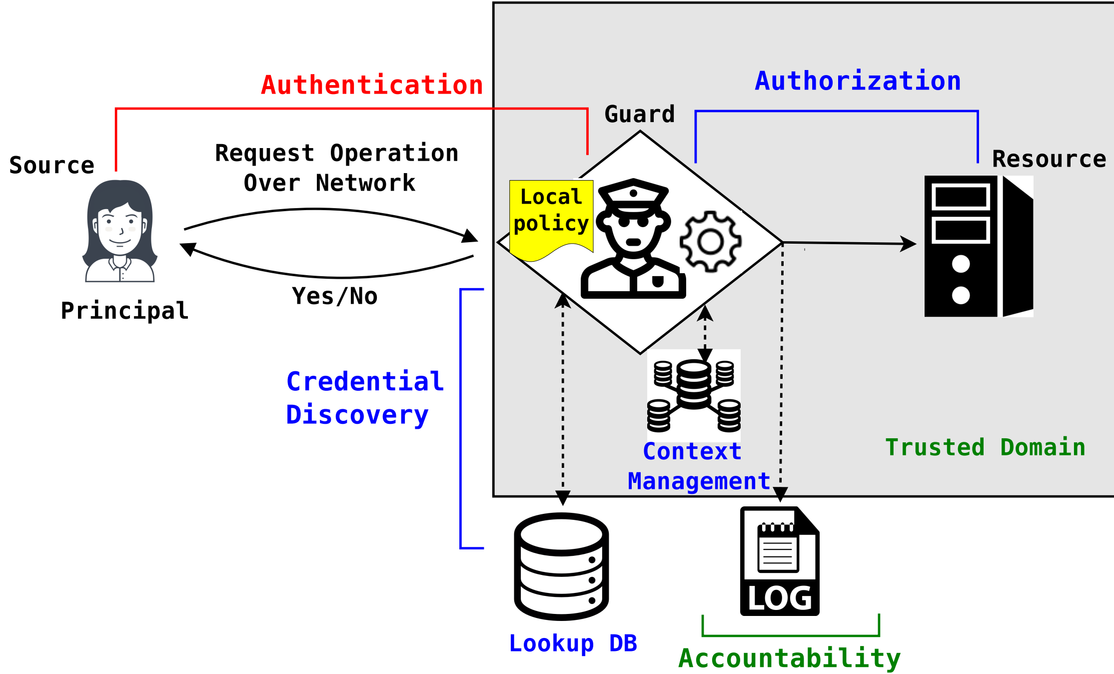


FIGURE 1.1: Overview of a Trust Management System illustrating the five primary components: authentication, authorization, accountability, credential discovery, and context management. Authentication identifies the principal making the request; authorization verifies whether the principal has authority to access the resource; accountability ensures that actions taken by the guard are auditable and policy compliant; credential discovery gathers the requisite credentials to substantiate a request, and context management supplies the tailored proof context pertaining to a request.

Decentralized trust management is a well-studied problem with a long history and rigorous foundations. Many of the trust management proposals use logic-based languages to manage trust: we call logic-based trust languages *trust logic*. Chapter 2 provides a summary of trust languages including trust logics. Although trust logics offer power and flexibility, their application to practice has so far been limited. In particular, a key premise of this thesis is that practical deployments face two key obstacles: *credential discovery* and *context management*. Credential discovery is a process of gathering the requisite credentials to substantiate the request for the trust decision. Credential discovery involves finding chains of credentials that delegate necessary authority from one or more trust anchors to the requester. Context man-

agement involves accumulating and tailoring each proof context pertaining to a given request so that a trust decision is reached quickly. Credential discovery and context management are challenging and open research problems in trust management systems. Figure 1.1 illustrates their role in these systems.

This dissertation introduces an integrated system called Secure Access For Everyone (SAFE) based on a logic-based declarative approach. SAFE provides a declarative trust logic for specifying credentials and security policies, and systems facilities to publish, retrieve, and organize credentials. SAFE builds on the existing foundations of logic-based declarative trust management. SAFE defines a logic-based trust management language (SAFE logic or "slog") that extends previous trust languages based on datalog, a well-understood language. One contribution of this thesis is to argue that extended datalog is the "right" foundation for logical trust because it is the maximally expressive tractable logic language, and is easily adapted for trust management.

However, the most significant contribution of SAFE is to address practical barriers to deployment by approaching end-to-end trust management as a *systems problem* encompassing credential storage, indexing, retrieval, and transport. SAFE addresses longstanding issues for the practical deployment of trust management systems: automated credential discovery; scalable credential revocation and rotation; tailoring of proof contexts for efficient logical inference; and integration with application service frameworks written in various languages.

1.1 Motivation

Services relying on computing and the Internet are ubiquitous, and increasingly the software and infrastructure resources we interact with "lives in the cloud" [AFG^{+}10]. The adoption of the cloud is also accelerated due to the increasing reliance on smart phones and smart devices—collectively known as the Internet of Things (IOT)—that

enable users and service providers to collect and exchange data in real time. The cloud may host these services on different geographical locations across the Internet, and the underlying user data and the service infrastructure may be managed by different entities. A major issue with today's trust model in the cloud is that the trust is driven by the identity and reputation of the service provider rather than the service it provides. For instance, X.509 (SSL) certificates authenticate a service to its identity or name/URL, but exclude other important attributes of the service and its deployment, or of other parties that support or endorse the service.

This thesis argues for a flexible and holistic trust, and introduces the notion of *trust agility*[1] for future clouds. Trust agility is "the ability for each participant in the system to decide independently whom to trust and what credentials to accept from another party based on its own local policies". From the perspective of the authorizer, trust agility involves inferring a local trust decision based on the real-world trust structure and the policies set by individual participants, including the issuer, the user, delegated authorities, and intermediary trust anchors. In particular, trust agility provides flexibility to the user to choose its own trust anchors and the attributes that guide its trust decisions. For example, in systems such as GENI, trust agility involves cooperation among self-organizing, federated entities to issue policies and credentials that are compliant with policies accepted by the authorizers, who may see different views of the dynamic trust structure. Importantly, trust agility subsumes and generalizes today's trust model (based on X.509 identity) by involving all of these participants in the trust decision.

[1]The term "trust agility" is first coined by Moxie Marlinspike in the context of TLS and authentication on the Internet [Mar11]. Marlinspike describes the trust agility as the ability for the user to choose its own authorities rather than relying on the rigid PKI hierarchy of Certifying Authorities (CAs).

1.2 Thesis Statement

"*Delegation-driven credential linking* and *trust agility* enable flexible, sound, and automated policy management for building secure networked systems."

Specifically, this thesis proposes an architecture for logical trust based on a new foundation: delegation-driven credential linking, which is independent of the underlying trust logic used for the inference. Credentials are linked in graph structures based on delegation patterns that reflect the real-world trust structure. Credential linking enables automatic discovery of credential chains and supports flexible trust structures among the participants. Our premise is that we can build trust-agile systems with the support of delegation-driven credential linking using a high-level declarative language that captures the user's trust intent precisely.

To evaluate this hypothesis, we implement a SAFE prototype system and evaluate its use to implement several applications.

SAFE integrates previous approaches and makes the applications trust-agile and crypto operations transparent. This approach is appropriate for an infrastructure to be used by application programmers since they are not experts in cryptographic algorithms or security protocols. In addition, the implemented system has a formal semantics and correctness guarantees that help to prevent programmer errors leading to security failures. This thesis aims to bridge the gap between the theory and practice of trust logics by offering the following features:

- **Safe declarative trust.** Policies, credentials and trust relationships are expressed in a declarative logic language (slog) that separates policy from mechanism. The language and proof semantics are independent of the mechanisms to store, retrieve, and authenticate credentials and of the semantics of the ap-

plications that use them. The language is expressive enough to support trust relationships and policies needed in practice.

- **Trust agility.** Each party can independently decide whom to trust and what credentials to accept based on its own local policies. Alternatively, each party may rely on a locally specified set of trust anchors for the appropriate credentials and policies. The parties can augment, retract, or refresh issued credentials and accepted trust anchors at their discretion—resulting in a dynamic trust structure.

- **Policy-driven credential linking and discovery.** Finding appropriate credentials to substantiate a given request is driven by a programmable policy for linking and organizing credentials. Actions to generate, link, and fetch credentials are coded in a scripting language (SAFE language or "slang") that is separate from the trust logic itself. These scripts guide the system on how to assemble the credentials (e.g., by name or through linked references) and tailor the assembled proof context for each decision.

- **Integrated service.** The system defines integrated services for credential manipulation, credential discovery, credential pruning, and building, organizing and caching proof contexts. In addition, the language provides built-in tools to integrate with the application service layer through REST APIs.

None of the existing trust management systems achieve all of these goals. Moreover, one of the goals for this thesis is to drive the progress from the current state of the art—the PKI and Certifying Authorities (CA) model—to a flexible trust model based on certified attributes of principals and objects. The progression from PKI, which is purely identity-based, to a certified attribute-based model is essentially a generalization. Instead of asserting only a single name attribute from a fixed set

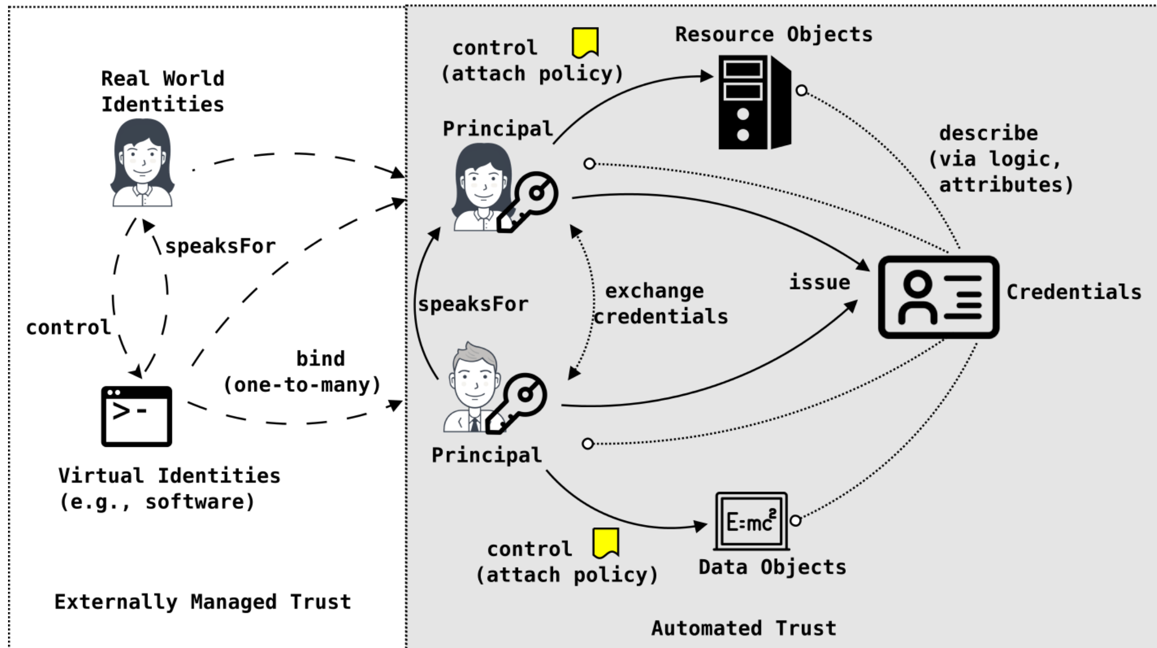


FIGURE 1.2: Elements of a Trust Management System. Real-world identities such as users, organizations, and corporations exercise control over software entities, which act as principals speaking for the associated identity. Principals exchange authenticated messages over a network.

of designated Certificate Authorities, any principal may assert any attribute, and clients choose their own authorities as per their local trust policies. This thesis describes the means for implementing such a trust-agile system using a logic-based authorization framework, and for reasoning about trust in such a system.

1.3 Overview

The elements of a trust management system include principals, objects, policies, and credentials. Figure 1.4 illustrate these concepts. Real-world identities such as users, organizations, and corporations exercise control over software entities bound to authenticated virtual identities (e.g., cryptographic keys), which act as the principals in the system. In practice, the binding of real-world identities to virtual identities is rooted in endorsements by other principals, e.g., identity services such as certificate

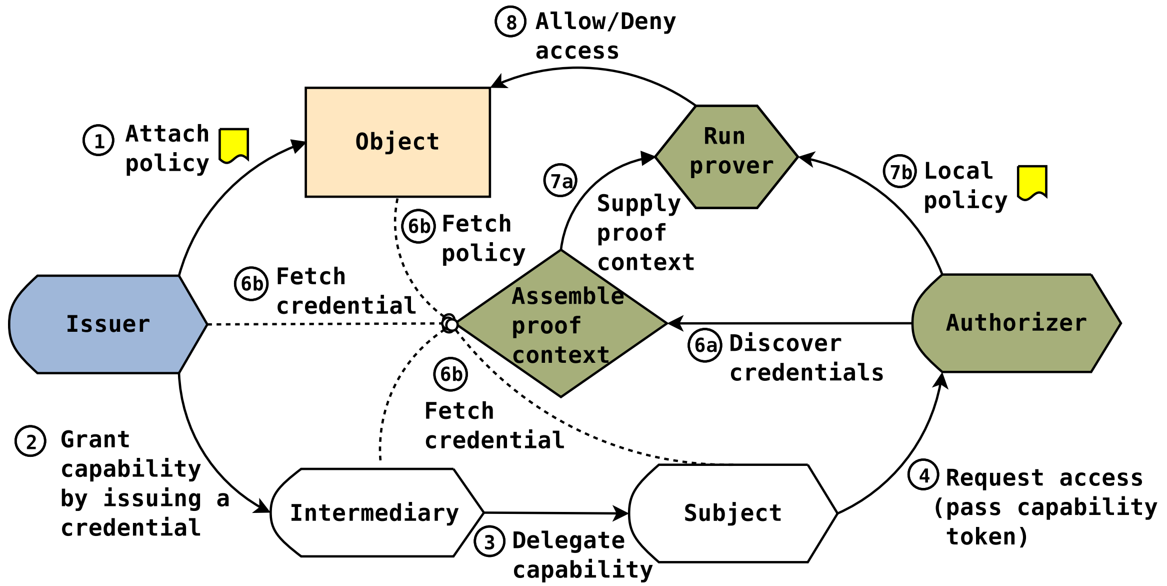


FIGURE 1.3: A workflow of authorization in a Trust Management System. An issuer controls an object by attaching a policy and then grants a capability on that object for another intermediary principal (delegator), who further delegates the capability to a subject. The subject makes a request to the authorizer by passing a capability token obtained from the intermediary. The authorizer assembles the necessary credentials and object policy from all the concerned principals, builds a proof context, runs a prover to infer the access privileges for the subject, and allow/deny access.

authorities or other endorsing entities (e.g., via PGP's web-of-trust model [Zim95]).

Principals may issue credentials, which are statements of belief about other principals and/or objects, authenticated under the issuer's keypair. For example, a credential may assert roles or attributes of a subject or object. Principals make decisions about trust or authorization by reasoning from credentials according to a local policy.

Figure 1.3 shows a typical workflow of authorization in a trust management system. An issuer controls an object by attaching a policy and then grants a capability on that object for another intermediary principal (delegator), who further delegates the capability to a subject. Authorization involves finding the necessary credentials that satisfy the local policy for a given access control request. When a subject re-

quests access to the object, the authorizer assembles and validates the credentials of the subject and other relevant credentials, builds a tailored proof context, and runs an inference.

A key obstacle for the authorizer is *credential discovery*: a trust decision may require reasoning from statements drawn from various sources, requiring a method to discover and retrieve them. Credential discovery is different from the certificate path discovery in X.509 certificates [EAH+01], since credentials in trust management systems generally have more complex meanings than simply binding names to public keys. More generally, the flexibility of logical trust systems comes at the cost of more certificates carrying more trust information from more diverse sources. The relevant trust chains may take the form of a directed-acyclic graph (DAG), rather than a linear path as in X.509. § 2.6 describes the common approaches of credential discovery.

SAFE takes a unified approach to credential management based on a foundational abstraction of linked logic sets. Credentials are stored as sets of authenticated logical statements (safe logic sets or `slogsets`), which are themselves first-class content objects with unique names. Slogsets can be linked together explicitly via a meta-predicate in logic to form credential chains representing the delegation patterns of the application. A *proof context* is a set of slogsets that are merged together to provide the requisite input for an inference decision. SAFE provides fine-grained context management to tailor the context per request level.

The purpose of credential discovery is to assemble a proof context for each trust decision. The proof context is the input to the compliance checker: it is a union of trust facts, policy rules, and credentials that the authorizer selects to consider for that decision. The context must be small: the cost of finding a proof scales with the number of statements in the context matching a goal. To keep the context small, it is necessary to identify the statements that are relevant and exclude the others.

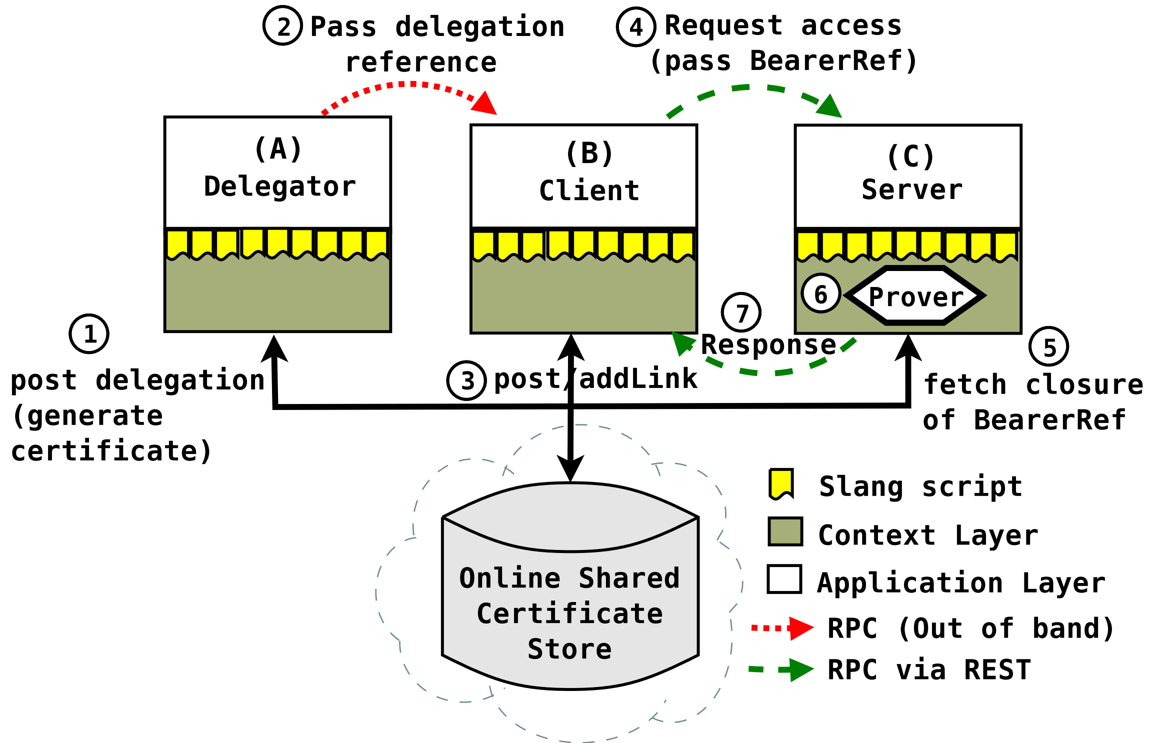


FIGURE 1.4: Elements of SAFE, a system for secure multi-domain (multi-principal) applications using logical trust. The application code of each participant handles trust events by invoking user-defined scripts running on a local SAFE interpreter. The scripts produce and/or consume logic-based credentials passed as linked certificates through a shared key-value store. In this example, A issues an explicit delegation (e.g., a group membership or a capability for an object) to B. A passes a link for the delegation to B, who adds the link to a permission set. Then, B issues a request as a client to server C, passing a link to its permission set. C fetches B's credentials through the link, validates and caches them, and passes them to its local logic prover to check access for the request.

The SAFE scripting language (slang) offers language constructs to build and modify slogsets, publish them by name, and add them as sub-contexts to a proof context. Slang also supports credential linking, a powerful technique to organize credentials and policies to enable automated credential discovery. A `link` is a meta-predicate in a slogset that serves much like a hyperlink in HTML documents. A slogset may link to a target set by its name: logically, the link incorporates the target as a subset, forming a union.

Delegation-driven credential linking is a construction procedure that integrates set linking with common primitives for delegation and endorsement, naturally materializing a graph in which each set links to the other sets needed to substantiate it. An authorization policy (a *guard*) specifies a context as a linked union of top-level subcontexts (e.g., logic sets associated with the subject, object, and policy). The transitive closure of the subcontext contains all sets relevant to a given authorization decision. Set linking also facilitates flexible policy because it is easy to attach policy rule sets to nodes in the graph.

The implementation of SAFE is based on a shared, distributed, and accountable credential store—*SafeSets*. Each credential in SafeSets is stored as a certificate signed by its issuer to enable any third party to verify the authenticity and integrity of the source. Certificates are stored in SafeSets, fetched on demand by their tokens, and cached locally. Certificates are named by secure *tokens* derived cryptographically from the issuing principal and a symbolic name assigned by the issuer. These tokens serve as a basis for certificate linking.

With SafeSets, the cost of linking is low because common subsets are cached at the authorizer. A further advantage with linking and shared storage is that the credentials can be prefetched and cached as materialized context sets for future queries. SafeSets and its caching layer also address a common problem of certificate-based systems. In current practice it is common for the client to transmit certificates with each request, leading the authorizer to validate them anew. The resulting redundant network traffic and cryptographic operations inhibit scalability. In SAFE, the client passes linked credentials by reference using a token: the authorizer uses the tokens to index its cache. The authorizer fetches and validates credentials as needed. The shared store is also a basis for addressing perennial problems with PKI certificate management, e.g., revocation, renewal, and key rotation.

Figure 1.4 shows the representative trust-agile system of Figure 1.3 implemented

using SAFE. The delegator in Figure 1.4 is either the issuer or the intermediary of Figure 1.3, and the client is either the subject or the authorizer. The delegator passes a certificate reference (link) to the client. The client preserves the reference by adding it to a “capability set” for later retrieval. When the client requests access to an object, the client passes a reference to the existing capability set to the authorizer; the authorizer fetches the slogset contents, validates and caches them, builds a proof context, and invokes the local logic prover to check access for the request. The authorizer can also augment the proof context with its own local policy or contents fetched from other trust anchors at its discretion. Linked data enables hybrid policies with flexible querying and credential retrieval, and provides the necessary foundation for trust agility.

1.4 Contributions

Our research focuses on practical challenges for using logical trust and credential discovery in secure networked systems. Our premise is that trust logic can be as simple as identity-based access control schemes (e.g., ACLs) in the common case, while enabling rich and flexible declarative trust policies with a rigorous logical semantics and verifiable properties. More generally, we believe that logical trust with certified linked credentials and policies can be a fundamental enabler for a network security architecture that is richer, safer, and more flexible than the architecture in place (e.g., X.509 and CA hierarchy).

This thesis builds a trust-agile system called Secure Access For Everyone (SAFE). SAFE was born out of necessity based on the need to build a flexible authorization system for NSF’s GENI project. This thesis makes the following contributions:

1. A declarative and tractable trust logic—slog—rooted in datalog with builtin support for the handling of objects and practical constraint domains (e.g.,

hierarchical and range domains). Slog is provably the most expressive tractable trust logic: any other trust logic is either not tractable or is reducible to slog, i.e., it is a syntactic variant. Slog provides a verifiable foundation for expressing statements of beliefs and trust.

2. A secure and flexible grouping of credentials and policies under a new logic set abstraction called *slogsets*. SAFE applications express trust-related code using a scripting language (slang) with primitives to generate and manage slogsets. With SafeSets, slogsets are issued and stored as certificates signed by the issuer and referenced by a globally secure slogset identifier (token).

3. An encoding format that encodes logic statements as "SAFE eXpressions" (SafeX). SafeX credentials may contain arbitrary logic content stored as variable-length plain text with attached cryptographic material (Base64-encoded), including optional signatures. SafeX enables credentials to be readable and navigable: the credential chains can be browsed by HTML rendering of embedded `link` metapredicates. The encoding is free of payload constraints, in contrast to the X.509 certificate standard. SafeX is independent of the mechanisms to store and authenticate credentials: for instance, SafeX logic content may be stored in Web servers and authenticated with TLS, replacing the SafeSets key-value store.

4. A high productivity scripting language and programming tool—slang—for managing credentials including credential generation using logic templates, credential linking and discovery, context construction, and revocation. Slang provides high-level semantics with builtin APIs/templates for handling practical trust policies. Slang scripts may implement standard trust models for off-the-shelf use by applications. Examples include assigning or assuming a role, joining a group, issuing and delegating capabilities, attaching an access control policy to

an object.

5. A flexible and dynamic method for credential discovery via delegation-driven credential linking for fine-grained context management. The construction procedure integrates slogset linking with primitives for delegation and endorsement, naturally generating DAGs in which each certificate links to others needed to substantiate it. In effect, SAFE distributes a large logic program over many sets and allows each principal to control which sets it imports for local execution, e.g., in a proof context. Credential linking with dynamic/mobile policy is a foundation for trust agility.

6. A shared, distributed, and accountable credential store—SafeSets—that leverages the concept of slogset linking to organize credentials. Slogsets may be passed by reference as linked certificate DAGs, and fetched from the store on demand. A client library included in each authorizer fetches the DAGs in parallel, validates each slogset, and caches them for later use. Pass-by-reference with caching avoids the common practice of transmitting redundant certificates with each request. The shared certificate store also helps to address the challenges with PKI certificate management: scalable revocation, renewal, and key rotation. The SafeSets abstraction is designed to be implemented as a strongly accountable storage service in the sense of CATS [YC07].

7. Mechanisms to integrate SAFE with application frameworks by running the trust code in a separate process (a SAFE instance) that encapsulates keying material and is accessed locally via REST APIs. Slang offers builtin constructs (`defguard`) to run any slang program as a “RESTful service”. Such integration enables legacy applications to use SAFE with minimal implementation changes. Moreover, a service can implement logic-based authorization as an add-on by running a SAFE instance as a proxy (shim) that checks incoming requests

before passing them to the application server.

8. An evaluation to demonstrate that SAFE is practical and supports a wide range of applications: (i) an authorization proxy shim that implements the security policy for SafeSets over a standard key/value store; (ii) a secure multi-component naming service (SafeNS) that subsumes secure naming, e.g., based on SPKI/SDSI; (iii) a proof-of-concept for a secure cooperative data analytics service (SafeCoda) that adheres to individual secrecy constraints while disclosing the data; and (iv) an authorization system for federated infrastructure-as-a-service system for the NSF GENI project (SafeGENI). The trust models for these applications are implemented with SAFE as short slang programs.

1.5 Organization

This thesis is organized into four parts: the introduction, the design, the implementation, and the applications.

The next chapter (Chapter 2) presents the related work to this dissertation. In particular, we summarize the seminal papers on trust logics, compare and contrast the major trust management systems, and discuss the prior approaches for credential discovery.

Chapters 3 to 7 introduce SAFE design and its foundations. Chapter 3 reviews the elements of logical trust systems: how to name principals, objects, and slogsets; the trust requirements to satisfy in a networked system; the security assumptions and design choices that guide the implementation of SAFE. Chapter 4 introduces slog, a tractable trust logic rooted in datalog. Chapter 5 presents a novel way of linking credentials to reflect the delegation patterns and also presents a set of common conventions for organizing the slogsets for credential discovery and efficient retrieval. Chapter 6 introduces the scripting layer called slang that provides tools

and templates for credential management. Chapter 7 shows how to express some standard security policies using slang.

Chapters 8 to 10 provides implementation of SAFE. Chapter 8 presents an overview of SAFE architecture. Chapter 9 introduces the shared certificate storage layer called SafeSets. Chapter 10 discusses how SAFE can be used for service integration with application frameworks; how credentials are assembled and cached in the authorizer; the implementation of certificate revocation, re-issuing, and key rotation in SAFE; and discusses a summary of issues when managing credential delegation and certificate management.

Chapter 11 presents the evaluation and applications of SAFE. § 11.1 presents microbenchmarks of standard SAFE operations. § 11.2 shows how the authorization for SafeSets is itself implemented in SAFE. § 11.3 presents an example of name-based authorization such as SPKI/SDSI and its implementation in SAFE. § 11.5 presents the authorization system for GENI and its implementation in SAFE. § 11.4 extends the application of SAFE to cooperative data analytics.

Chapter 12 discusses the future work and possible extensions of SAFE to other application domains. Chapter 13 defines—with examples—the concepts, the terminology used in this thesis with reference to SAFE.

2

Background

SAFE builds upon the foundations of trust logics—an extensively studied area with a wealth of prior results. Thus the related work for SAFE encompasses authorization logic [BFG10, SdR+11, DeT02, Jim01, BSF02, AL07, LGF03], trust management [BIK03, BFL96], proof-carrying authorization [AF99, LLFS+07], and tag-based access control systems [HGL+12, MLM+14].

Using logic for access control. Consider a ternary relation `capability(Principal, Object, Privilege)` that holds whenever a given principal has a given privilege on some object. An instance of this predicate naming a particular principal and a particular object represents a trust fact as an atomic logic statement. An access control list is a set of such facts grouped by objects. A capability list is a set of such facts grouped by principals. The logic formulation decouples the representation of trust material from the underlying indexed groupings while storing and retrieving facts. Another benefit of the logic formulation is that users may express security policies as logic rules and infer access control decisions given a set of facts and rules. For example, a privilege may be expressed as a conjunction of other privileges, which is naturally captured by a rule.

From logics to trust logics. For a logic to serve as a trust logic, it must be sufficiently powerful to reason about *attribution* of statements to specific principals, and to represent *delegation* of trust among principals (e.g. as policy rules). Attribution requires an authenticated source for each statement. For example, statements may be signed and issued as certificates.

To represent attribution, trust logics use some variation of a `says` operator that qualifies each atomic statement with a principal who says or believes it. For delegation, they either use some form of a `speaksFor`[1] operator to represent that one principal trusts another, or incorporate the `says` operator into logical inference rules as described later in this chapter.

Credential discovery. In addition, for a logical trust system to be useful, it must retrieve, accumulate, and assemble requisite credentials into a *proof context*—a set of facts and rules used as an input to a prover for a particular trust decision. Credential discovery involves finding chains of credentials that delegate necessary authority to the requester. In *proof-carrying authorization (PCA)* the onus is on the requester to provide the proof context. Other systems use a distributed query model to identify and retrieve relevant credentials on demand.

In this chapter, we summarize the logical foundations of trust from seminal papers. We compare trust logics by grouping them into three categories: ABLP-based logics with high expressive power (§ 2.1), special-purpose languages for trust management (§ 2.2), and tractable trust logics based on datalog (§ 2.3). We then summarize other systems related to managing trust (§ 2.4), and discuss approaches to credential discovery in logical trust systems (§ 2.6).

2.1 ABLP-based logics

BAN logic. BAN logic [BAN90] is a pioneering work on applying logic to the

[1]`speaksFor` is also known as `actsFor` in some logics [ML97]

problem of authentication. It provides a proof system for reasoning about authentication protocols including Kerberos, secure RPC, Needham-Schroeder Public-Key, and X.509. Although BAN logic focuses on authentication, it introduces several key concepts of authorization logic representing beliefs, statements, and authority of participants in a distributed setting. In particular, BAN logic introduces constructs called `believes` and `controls`, which form the basis for later trust logics.

If P is a principal and S is a statement then "P `believes` S" means that the principal S acts as if S is true, and may assert S in messages.

If O is an object, then "P `controls` O" means that the P has jurisdiction over O. The principal P is an authority on O and should be trusted on statements about O. For example, a server is often trusted to generate encryption keys properly. This trust may be expressed by the assumption that the principals believe that the server controls the keys and therefore has jurisdiction over statements about the quality of keys.

ABLP logic. ABLP [ABLP93, LABW92] introduces access control calculus to reason formally about the security policies and principals in the system. The Taos operating system used a form of ABLP logic as a basis for authorization [WABL93]. This is the first use of a trust logic for authorization in practice. ABLP inspired many successors.

ABLP builds on BAN logic with a focus on richer authorization semantics such as groups, roles, and delegation. ABLP expresses attribution using a `says` operator, which is equivalent to the `believes` construct from BAN logic. A statement with the prefix "A `says`" indicates that the statement is asserted by and/or believed by A.

Abadi provides a revised overview of ABLP axioms and variants relevant to authorization logic in general [Aba08]. In particular, every principal *controls* its own beliefs through axioms known as *Hand-off* and *Bind*. Bind enables mobility of policy

rules: a principal may import authenticated policy rules and evaluate them locally to infer beliefs of their issuers. Hand-off captures transfer of authority from one principal to another. It is based on closure of the `says` operator under implication. In a simplified form, hand-off is expressed as: "if A `says` (B `says` $S \implies A$ `says` S) $\wedge$ B `says` S, then A `says` S", where A and B are principals, S is a statement that is hand-off from B to A. § 4.3 discusses how equivalents of the hand-off and bind axioms are manifested in slog.

ABLP logic introduced a `speaksFor` operator to represent delegation of authority. ABLP treats `speaksFor` as the foundational construct for all delegation of trust. `speaksFor` in ABLP states that if B speaks for A, then all the statements said by B are also said by A. Formally, B `speaksFor` A is stated as:

$(\forall s : (B \texttt{ says } s) \Rightarrow (A \texttt{ says } s))$.

`speaksFor` delegates full authority over beliefs from one principal to another. The successor logics of ABLP developed various uses of `speaksFor` to constrain the authority to a particular attribute or a property as a practical basis for delegation [SWS11, HK00b]. § 4.6 discusses an alternative logical basis for delegation that does not rely on `speaksFor`.

In ABLP logic, principals can be users, roles, machines, I/O channels, or encryption keys. In addition to atomic principals, ABLP introduces several constructs to represent compound principals directly in logic:

- $A \vee B$: this principal represents the group containing A and B.
- $A \wedge B$: this principal issues statements signed jointly by A and B.
- $A \mid B$: this principal (pronounced as "A quoting B") issues statements said by A to originate from B.
- A `for` B: this principal represents A speaking on behalf of B, which is a

stronger notion than $A|B$.

- A `as` R: this principal represents A assuming the role R.

These constructs make ABLP a powerful and expressive trust logic that can represent a wide range of trust systems logically. However, in the general case, these constructs also make the logic intractable and/or undecidable. Abadi *et al.* [ABLP93] proposed some restrictions for a tractable ABLP subset but the result is inconclusive. Howell *et al.* further argues that certain constructs of ABLP (e.g., quoting) are ambiguous [HK00b].

2.2 Languages for Trust Management

In conjunction with ABLP logics, several attempts were made to design trust management systems to express and verify security policies with special-purpose languages. These systems include PolicyMaker [BFL96], KeyNote [BIK03], SDSI [RL], SPKI [EFL^{+}99], and RT [LMW02].

PolicyMaker and KeyNote. PolicyMaker introduced the problem of decentralized trust management—formulating security policies and credentials in a distributed setting [BFL96]. In PolicyMaker, the credentials and policies are asserted in a declarative language similar to SQL. Credentials are interpreted locally. If there exists a delegation statement, it can be used to transfer authority only if the local policy has also indicated explicitly that such delegation statements may be trusted. PolicyMaker has a graph-theoretic semantics, but the language in general is undecidable because programs contained in its assertions can be written in a Turing-complete language.

KeyNote [BIK03] is a successor to PolicyMaker with a more restrictive trust language. The system relies on PKI for authentication. The language for specifying credentials is declarative with credential labels representing the public keys of the

issuers. KeyNote attempts to represent policy compliance as a value that captures the degree of compliance of a request with a policy. REFEREE [CFL+97] is a system that applies the ideas of PolicyMaker and KeyNote to secure web applications.

KeyNote has several drawbacks. For instance, KeyNote does not provide a language for defining and manipulating groups. More importantly, KeyNote is not rigorously defined and it is intractable. A later effort attempted to capture the power of KeyNote as a tractable and rigorously defined Delegation Logic [LGF03], which also inspired further work on Role-Based Trust (see below).

SDSI. The Simple Distributed Security Infrastructure (SDSI) [RL] addresses the problem of secure naming and authorization in distributed environments. SDSI introduces the notion of *local names* bound to (issued by) a public key. A local names may be referenced globally by qualifying it with the public key of the issuing principal. An issuer associates access privileges with a particular local name, and any principals bound to that name by SDSI name certificates are granted those access privileges. Thus, SDSI implements named groups or roles as a basis for authorization.

SPKI. The Simple Public Key Infrastructure (SPKI) adds to SDSI the ability for a principal to bind a capability or access privilege to another principal's public key by issuing an authorization certificate. Since the functionality of SPKI and SDSI are similar, they were merged into a single framework called SPKI/SDSI 2.0 [EFL+99]. SPKI/SDSI treats principals as public keys; without loss of generality, the identity of the principal is taken as a hash of the public key.

The certificates of SPKI/SDSI 2.0 come in three forms: name-to-key, authorization-to-name, and authorization-to-key bindings. A principal issues a name-to-key certificate to bind a local name to a public key [Aba98, HVdM01]. Authorization certificates are interpreted to mean that a given key has the right to exercise a named privilege. Privileges are represented as strings and are combined using string rewrite rules proposed in [CEE+02]. Each certificate has a boolean flag that de-

scribes whether the recipient may further delegate the privilege. The authorization process involves verifying the validity of certificates and computing the intersection of privileges to determine whether a named principal may access a particular resource.

SPKI/SDSI was applied in practice to implement an access-control mechanism for protecting web pages [CEE+02]. An interesting feature of SPKI/SDSI is the handling of certificate revocation lists (CRLs). SPKI/SDSI mandates that each certificate must identify the keys that issue CRLs and the locations where they are published, and CRLs for a certificate must carry non-intersecting validity dates. This makes it possible to accept certificates only in the presence of a valid CRL, and it ensures that only a single CRL is valid at a given time.

RT* languages. Role-based Trust (RT) defines a set of multiple languages with varying expressiveness and complexity [LMW02]. Initial versions of RT relied on a constrained variant of Delegation Logic ($D1LP$) that is monotonic and limits delegation depth [LGF03]. The base language RT_0 defines statements to associate named roles with principals and infer roles based on conjunctive policy rules. RT showed that the delegation can be captured naturally via roles/attributes and closure of the *says* operator under inference.

RT_1 is an extension of RT_0 that provides parameterized roles. RT_1^C further extends RT_1 to describe structured resources such as hierarchical names [LM03]. RT_2 adds support for objects and object attributes via *o-sets*. The language RT^D provides a mechanism to describe the delegation of privileges and role activations, and RT^T adds support for policies involving threshold and separation of duties. The thresholds and delegation features can be used with standard languages creating combined languages such as $RT_0^T, RT_1^{CD}, RT_2^{TD}$.

Snowflake. Snowflake is a language and system inspired by ABLP and SPKI/S-DSI. A key outcome of the work is a formal semantics of SPKI/SDSI using ABLP logic [HK00b], which also addresses the potential for a tractable subset of ABLP.

Howell argues that the `quoting` operator in ABLP is ambiguous and not necessary for reasoning about trust. Howell introduces a restricted `speaksFor` known as `speaksForOn`[2] as the foundational construct in the Snowflake system [How00]. Further, the role and control operators in ABLP are supplanted by `speaksForOn` definition.

The Snowflake project also promoted a principle of *end-to-end authorization* common to many logical trust systems (including SAFE), in which authorization decisions are made locally based on authenticated statements of multiple parties, without a need for trusted intermediaries [HK00a].

2.3 Datalog-based logics

In a quest to simplify the complexity of ABLP-based logics and other trust languages, simple and expressive languages based on first-order logic were developed. In particular, datalog played an important role in the development of trust management systems. Many systems including QCM [GJ00b], SD3 [Jim01], and Binder [DeT02], SecPAL [BFG10], SeNDLog [AL07] are based on datalog and its extensions.

Datalog is a simple, expressive, and tractable subset of first order logic restricted to Horn clauses with a transitive closure operator [AHV95, CGT89]. Datalog may be viewed as a declarative subset of the Prolog [SS94, WSTL12] language for logic programming. The *safety* condition of datalog guarantees program termination in polynomial time under various restrictions, including restrictions on the use of negation so that the logic is monotonic.[3]

Although datalog cannot express structured data directly (due to restrictions on function symbols), datalog with constraints [LM03] can express ordered domains that are important for practical trust management systems, i.e., integer and tree-

[2]Howell calls `speaksForOn` the `speaks-for-regarding` operator.

[3]Safety condition in datalog is also known as range restriction.

structured name spaces such as IP addresses and pathnames.

Further, it is shown that datalog with a `says` operator reduces to datalog [LM03], forming a tractable and practical trust logic. Constrained delegation is captured directly in datalog-based trust logic by means of policy rules that delegate control over a specific logical predicate (representing a role or attribute) to another principal using *says*. Such a delegation is inherently constrained to a specific predicate. Datalog cannot capture unconstrained delegation (`speaksFor`) directly due to the absence of a `forall` construct (see § 2.1), but it has limited value in practice, and is intractable.

§ 2.2 summarized several trust languages that were not derived from logic. However, in later work, the RT authors analyzed the descriptive complexity of their system relative to SPKI/SDSI and datalog. In the process, they provided a first-order logic semantics for SPKI/SDSI (see also [Aba98, HVdM01, HK00b]). They did this by translating SPKI/SDSI to RT_1, and further reducing RT_0 and RT_1 to datalog with constraints [LM03, LM06]. A notable result from the work is to show that the logical semantics for SPKI/SDSI is easier to analyze than the string rewriting semantics. Moreover, their result shows that the standard SPKI/SDSI proof procedures in RFC 2693 [EFL+99] are semantically incomplete. Among other benefits of a logical point of view, the constraint feature of RT_1^C is equivalent in expressive power to SPKI/SDSI tags, but is algorithmically tractable.

Thus the languages described in § 2.2 are shown to be either intractable or to reduce to restricted, tractable first-order logic (datalog with constraints). Moreover, the RT work shows that SPKI/SDSI, RT_0, and RT_1 are weaker than datalog. SPKI/SDSI lacks conjuctions, and RT_0 and RT_1 lack support for objects and object attributes, all of which are important in trust management systems. While RT_2 claims to provide all of these features, the published work defines no rigorous semantics for RT_2 and, in particular, no translation of RT_2 to datalog. Although the Attribute-Based Access Control (*libabac*) software from ISI claims to implement RT_2

by translation to Prolog [RFWS11], the semantics of RT_2 are not defined precisely, except by this implementation.

We now turn our attention to trust logics that were inspired by datalog from their conception, and use a syntax based on datalog-with-says.

SD3. Secure Dynamically Distributed Datalog (SD3) [Jim01, JS01] and its predecessor Query Certificate Manager (QCM) [GJ00a, GJ00b] are both based on relational algebra. SD3 extends QCM to add recursive policies that support delegation.

In SD3, the credential discovery procedure is encoded in logic itself (see § 2.6). Jim describes a prototype for SD3 and shows how to implement a a DNSSEC-like secure federated naming system in [Jim01]. Much of the structure exists to facilitate navigation of the name space and retrieval of chains of credentials proving the binding of a global name. In SD3, these proof chains may optionally pass through a local evaluation procedure known as *certified evaluation*, in which the output of SD3 query evaluator is checked by a relatively simple proof checker. The assumption here is that proof checking is simpler than proof construction; the proof checker can be small and simple, which reduces the size of the trusted computing base. SD3 uses a database query optimizer for both credential discovery and proof checking. In SD3, these steps are executed sequentially rather than interleaved as in RT [LWM03].

Binder. Binder [DeT02], similar to SD3 [Jim01], relies on datalog as its foundation. An authorizer evaluates a set of datalog rules and facts to check compliance with local policy. These rules and facts exist in a *context*, which is a set of statements accepted by a principal (e.g., the authorizer). Rules and facts spoken by a principal can be exported from its context by signing them. Thus signed datalog statements form the certificates in the Binder system. Importing from another context assigns a speaker to the imported statements with the *says* operator in a way similar to other ABLP-inspired logics.

In effect, Binder distributes a large datalog program over many contexts and

allows each principal to decide explicitly which statements to accept from other contexts. Although Binder allows flexibility on the flow of statements among contexts, Binder lacks the notion of local names and does not address the issue of credential discovery. In Binder, each principal has a single context.

The notion of a context and export/import of contexts in SAFE are similar to Binder. § 3.1 discusses the design choices of SAFE that are inspired from Binder. A key contribution of SAFE is to introduce the concept of multiple, named, fine-grained contexts as a means to manage credential discovery and credential flow among contexts, and to limit the number of statements passed to the prover.

SeNDLog. Secure Network Datalog (SeNDLog) [AL07], builds on Binder and unifies two languages: Binder [DeT02]—a secure trust language; and Network Datalog (NDLog)—a database query language for declarative networks [Loo06, LCG+09]. As in Binder, SeNDLog uses `says` to import predicates (statements) from another context. For instance, an import predicate is of the form "`N says p`" in a rule body, where `N` is the principal that is asserting the predicate `p`. In addition, SeNDLog defines an export predicate with location identifier of the form "`N says p@X`" in the rule head, which says the principal `N` exports the predicate `p` to the context of principal `X`. The support for export predicates makes SeNDLog suitable for specifying secure network protocols.

In contrast, SAFE does not have export predicate and does not treat datalog programs as distributed evaluation. SAFE logic is declarative rather than programmatic: a separate scripting layer orchestrates export/import of statements from a shared store and establishes relationships among statements by linking them.

SecPAL. SecPAL [BFG10] and its predecessor Cassandra [BS04] attempt to provide natural language semantics that translate to datalog with constraints. SecPAL is notably powerful among datalog-based trust logics in its support for stratified negation and negation among goal predicates. SecPAL and Cassandra are designed

for practical use cases involving the authorization of United Kingdom's electronic health records. Neither project addresses the problem of credential discovery.

2.4 More recent logical trust systems

This section gives an overview of more recent systems for trust management.

Alpaca. Alpaca provides an extensible authorization framework for authentication systems. Alpaca builds on PCA, expressing a credential as an explicit proof of a logical claim [LLFS$^+$07]. Alpaca generalizes PCA to express not only delegation policies but also the cryptographic primitives, credential formats, and namespace structures needed to use foreign credentials directly. The goals of Alpaca are quite different from SAFE in that Alpaca aims to provide a path to unifying and generalizing PKIs to support rich authentication systems.

NAL. Nexus Authorization Logic (NAL [SWS11]) was introduced as a basis for logic-based authorization in the Nexus trustworthy operating system and related networked systems. NAL builds on Abadi's treatment of CDD[4] [Aba08], augmented with rules for existential quantification, sub-principals, and groups. NAL has a powerful set of system state parameters and predicates as inputs to authorization. In NAL, the approach for handling groups and roles is similar to Delegation Logic [LGF03]. NAL has not been shown to be tractable.

Soutei. Soutei [PK06] is similar to KeyNote, but using Binder as a security language. Soutei is implemented in Kanren [kan09], a logic-based language implemented in Scheme. Soutei is used as an authorization system for publish-subscribe data web serving system.

Macaroons. Macaroons [BPE$^+$14] is an alternative integrity mechanism for credential-based authorization. Macaroons are credentials that permit a holder to

[4]CDD is based on Dependency Core Calculus, which simplifies the semantics of ABLP. A good overview of CDD in comparison with access control logics is provided by Abadi in [Aba08].

append various restrictions before delegating them to another party. The restrictions in macaroons are *caveats* that attenuate and contextually confine when, where, by whom, and for what purpose a delegated credential may be used. The integrity of the delegation chain is assured by incorporating nested, chained MACs (e.g., HMACs) into the delegation chain, so that the authorizing party (the root credential's origin) can validate that all restrictions are included in a delegated credential. HMACs are more efficient and easier to deploy than credentials signed using asymmetric cryptography. However, macaroons credentials are issued for a specific target site: i.e., the target site must be known a priori to the issuer, and subsequent delegations of a credential are limited to refinement of the single credential chain.

Tag-based authorization. Tag-based authorization (TBA) aims to provide a hybrid approach combining the ease of extensional systems while still maintaining the flexibility of logical systems [WJL09, HGL+12]. Extensional access control systems are systems in which users specify atomic values for roles, privileges, and delegation. Example of extensional systems include the access control lists (ACLs), role-based access control (RBAC), and Bell-La Padula mandatory access control model. Penumbra is a TBA system for authorizing accesses to shared data in a file system. [MLM+14].

Tag-based access control systems associate tags to both subjects and objects. The authorization policy is defined in terms of tags. The policy for an object (O) specifies the required tag access rights (R) needed to access it. A subject (S) may obtain those access rights by delegation. TBA differs from standard logic-based access control systems in that tag has a fixed semantics and delegations are defined outside the logic. Advocates of TBA contend that they are more understandable to users than other schemes for rich access control.

Purpose-based access control. Purpose-based access control is an application of tag-based access control for privacy preserving data access where the queries are

specified with an intended purpose [BL08]. It is useful in scenarios where the organization structure and the expected queries are stable (e.g., relational data analytics), and the intended purposes are known a priori.

Purpose-based access control with explicit negation allows policies to be expressed succinctly.

User authentication in Self-Certifying File System (SFS). Kaminsky et al. [KSMK03] propose an approach to authorization in a distributed file system without involving certificates. The access control model is based on nested federated groups (similar to SPKI/SDSI). Their approach requires one server per domain. The local servers fetch user and group definitions from remote servers in advance and cache them. During a file access, a server can establish identities and group memberships for users based solely on local information.

2.5 Proof-Carrying Authorization (PCA)

Appel and Felten proposed the PCA approach [AF99] that attempts to provide a generic solution to trust management systems by combining a logic approach with a requirement that each client request carries a complete proof of access, possibly involving logic credentials from multiple sources. PCA was subsequently developed by Bauer in his PhD thesis work [BSF02, Bau03]. PCA borrows concepts from proof carrying code [Nec97] where untrusted code must be accompanied by a safety proof that is checked by the consumer of that code.

The PCA work used a higher-order AF logic as a language for defining an purpose-specific access-control logic for each application. Although the higher-order logic is intractable and undecidable, the premise of this approach is that it can be used to express tractable application-specific logics for common scenarios. For any particular application, an application-specific logic can be defined where the rules of inference in that logic are lemmas in the general higher order logic. The requester can construct

a proof using a particular application-specific logic without involving the authorizer. For instance, SPKI/SDSI is encoded as application-specific logic in PCA.

Similar to SPKI/SDSI, the PCA approach has been used to for protecting web pages [BSF02, Bau03]. A variant of the proof system was used in the Grey Project [BGR05, BCR+08], which uses a smart phone platform to authorize access to offices and shared labs according to logic-based policies.

2.6 Credential Discovery

Credential discovery is a process of gathering the requisite credentials to verify that a request complies with an authorization policy. Credential discovery involves finding chains of credentials that delegate necessary authority from one or more trust anchors to the requester. Much of the work in trust management does not consider the credential discovery problem: it merely assumes that an authorizer obtains all of the potentially relevant content by some means before checking compliance [BFL96, BIK03, WABL93, BFG10]. Some works that considered credential discovery are QCM [GJ00b], SD3 [Jim01], SPKI/SDSI [CEE+02] and RT [LWM03].

Several approaches to credential discovery are discussed in the literature.

Subject providing credentials. The subject may provide the requisite credentials by herself to the authorizer; this is the idea behind proof-carrying-authorization [AF99] (PCA). PCA solves the credential assembly problem by requiring requesters supply the complete proof to obtain access. In such a system, the authorizer is substantially reduced and simplified; it needs only to verify the credentials and proof, which is much less costly than identifying the credentials and finding the proof. However, the problem is not solved; the burden is merely shifted to the requester.

On-demand distributed querying. Upon receiving a request from a subject, the authorizer launches a distributed query [BGR07, LWM01] to fetch the credentials (or partial proofs) from the relevant sources (e.g., the issuer, intermediary, and object

from Figure 1.3). A related approach to credential discovery based on distributed proving is proposed by Bauer *et al.* in [BGR05, BCR+08].

The credential retrieval in QCM and SD3 is based on distributed query evaluation in the style of a database system. They assume that the issuers store and maintain all the credentials from the initial published time; every query in SD3 is answered by doing a forward search from the issuer to the requester. An SD3 query evaluator automatically fetches remote certificates needed to fulfill access requests. In addition, it allows certificates to be requested using wildcards, and caches remote certificates after they have been fetched.

The credential discovery in SPKI/SDSI is addressed by Clarke *et al.* [CEE+02] via a bottom-up term rewriting approach. Their proposed algorithm merges the inference process with credential retrieval by viewing each credential as a rewriting rule, and retrieval as a term-rewriting procedure. The algorithm handles cycles using a bottom-up approach to perform a closure operation over the set of all credentials before it finds the next delegation in the chain.

The authors of *RT* introduce the distributed credential discovery problem in in the context of RT_0, and propose a solution that interleaves the credential retrieval and evaluation steps through a goal-directed approach. Credential discovery in RT_0 shares two key characteristics with discovery in SDSI: linked names give credential chains a non-linear structure and role definitions can be recursive. In RT_0, credentials are represented by edges. Principals, roles, and other role expressions are represented by nodes. Chain discovery is performed by starting at the node representing the requester, or the node representing the role (permission) to be proven, or both, and then traversing paths in the graph trying to build an appropriate chain. The evaluation process (the logical inference) can begin with incomplete set of credentials. If the evaluation cannot find a possible match, then the evaluation is suspended and a distributed query is launched for further credential discovery. The process iterates

until the search is exhausted or a possible solution is reached.

In contrast to distributed proving techniques, the focus of this thesis is on end-to-end authorization techniques that evaluate all requests locally based on credentials obtained before the proving starts. A secondary goal is to tailor the context to include only the relevant material. This is important because the cost of inference increases with the number of statements present in the proof context.

We believe that interleaving credential discovery with proof validation is needlessly complicated to implement and quite inefficient for common policies. In SAFE, we take the goal-directed approach inspired by RT, but we perform the credential discovery sequentially followed by certified evaluation rather than interleaving both the steps. Credential discovery is aided by *explicit* linking of credentials that are built based on the delegation patterns of the applications (see § 5.1), and a common storage layer to fetch and cache credentials by their links. The advantages of credential caching are explored in multiple projects including REFEREE [CFL+97], SD3 [Jim01], SFS [KSMK03], and Grey [BGR05].

Shared distributed storage. The SAFE implementation uses a decentralized system for certificate (slogset) storage based on a highly available key-value store or DHT (SafeSets; see Chapter 9). The early X.500 model proposed a distributed certificate repository, as discussed by the SPKI/SDSI authors [EFL+99], who judge it to be impractical given its reliance on a single global symbolic name space. In contrast to the X.500 approach, the SafeSets model indexes certificates by cryptographic identifiers, and does not rely on global principal names other than public keys, as advocated by SPKI/SDSI. Various other systems have proposed certificate storage for use in credentials-based authorization (ConChord [ACMR02], CERT-DIST [SPJF09]) using various indexing and naming schemes, but they do not support set linking. SafeSets supports secure unforgeable certificate (slogset) identifiers for a key/value store by qualifying them with a public key hash (similar to SPKI/SDSI or

self-certifying identifiers in SFS). A similar technique has been used in many DHT applications [MKKW99, LKMS04, SMK+01].

2.7 Summary

This chapter summarizes the foundations of trust logics and the systems related to SAFE. Although there are many formulations of trust logics, at the core, all trust logics employ a `says` operator and provide constructs for restricted delegation. For example, the ABLP `speaksFor` operator provides full delegation of authority from one principal to another, and can be extended as a basis for restricted delegation. Datalog-based logics (as in SAFE) express restricted delegation by using the `says` operator in conjunction with logical inference rules, enabling a principal to form beliefs based on statements made by delegates that match the rule. In addition, SAFE addresses a central issue in trust management: credential discovery and context management.

3

Logical Trust on the Network

In this chapter, we present the design elements of SAFE: the foundations of SAFE trust logic; how the contexts are managed based on a new set-based abstraction; the decoupling of context management from the logical inference; how credentials are linked externally to reflect the delegation patterns of the application; the native support for flexible encoding in certificates; and how SAFE leverages a distributed shared repository for scalable certificate management. We also present how SAFE names its principals, objects, and slogsets; the trust requirements to satisfy in a networked system; and the security assumptions that guide the implementation of SAFE.

3.1 Design Overview

SAFE is an integrated system for end-to-end trust management. The design of SAFE is influenced heavily by the previous work on trust management. The naming of principals and objects follows new conventions based on self-certifying names. The trust logic—slog—is rooted in datalog following the earlier systems such as SD3 [Jim01], Binder [DeT02], and SecPAL [BFG10]. We also address several missing pieces: auto-

mated credential discovery, scalable certificate distribution and revocation, language support for programmable policies, and application service integration.

A goal of our design is to select carefully the right choices from existing research, synthesize them with new abstractions to manage proof contexts, and innovate on the whole so that the resulting system is practical to adopt in real systems. We designed slog based on datalog-with-**says** with some extensions for supporting practical constraint domains. Slog, as a trust logic, is sufficiently powerful to represent useful access control features needed in our applications. We believe the slog is "right" trust logic to start with for practical use since it is the maximally expressive tractable trust logic (see Chapter 4).

This thesis focuses on system-level abstractions that are independent of the trust logic variant in use. The design philosophy of SAFE is guided by the following principles:

- **Separation of concerns.** The trust-related scripts and logic are factored out of an application into a separate container that serves as the trusted computing base (TCB). The application service (app service) interacts with the TCB through a well-defined API. No other interactions with the TCB are permissible.

- **Scripted language to manipulate logic content.** The logic content produced via trust logic involves credentials and delegation of credentials for other principals. The credentials must be archived and shared, and then later retrieved and cached to infer a trust decision. The constructs for publishing, retrieving, and caching are abstracted via a scripted language that sits above the logic. The scripts encapsulate logic concerns behind a trust API for use by the main application code.

- **Fine-grained context management.** A proof context may contain state-

ments issued by many principals originating from different sources. These proof contexts may be shared across queries: for example, two queries may share a common trust anchor or common root authority. Sharing and indexing contexts by principal IDs and their tokens enables fine-grained context management that improves the overall efficiency of the trust system.

- **Delegation-driven credential linking.** The credentials are linked in graph structures to mirror the delegation patterns of the application. A credential may link to other credentials by their tokens: the link incorporates the target credential set as a subset, forming a union. Credential linking enables delegation-based context pruning that can reduce validation and proof costs.

- **Flexible transport format.** A credential containing a set of logic statements can be encoded in a format that is readable, navigable, and easily transportable via multiple forms (e.g., email, RPC).

- **Online shared storage.** The credentials are signed as certificates and stored online via a secure shared storage service. The shared certificate store also helps to address perennial problems with PKI certificate revocation, renewal, and key rotation.

Figure 3.1 depicts the SAFE system. It has four components: the app service; the slang context layer; the slog prover; and the SafeSets shared certificate storage. The app service interacts with the context layer; the context layer invokes the prover, and manages the credentials and delegations for the application.

The app service is a container that hosts the main application code, which is written in any general-purpose language. An application needs to manage its trusted computing base (TCB) carefully. In SAFE, the kernel of trust is specified using a scripting language (slang), and the scripts are called *trust scripts*. The scripts isolate

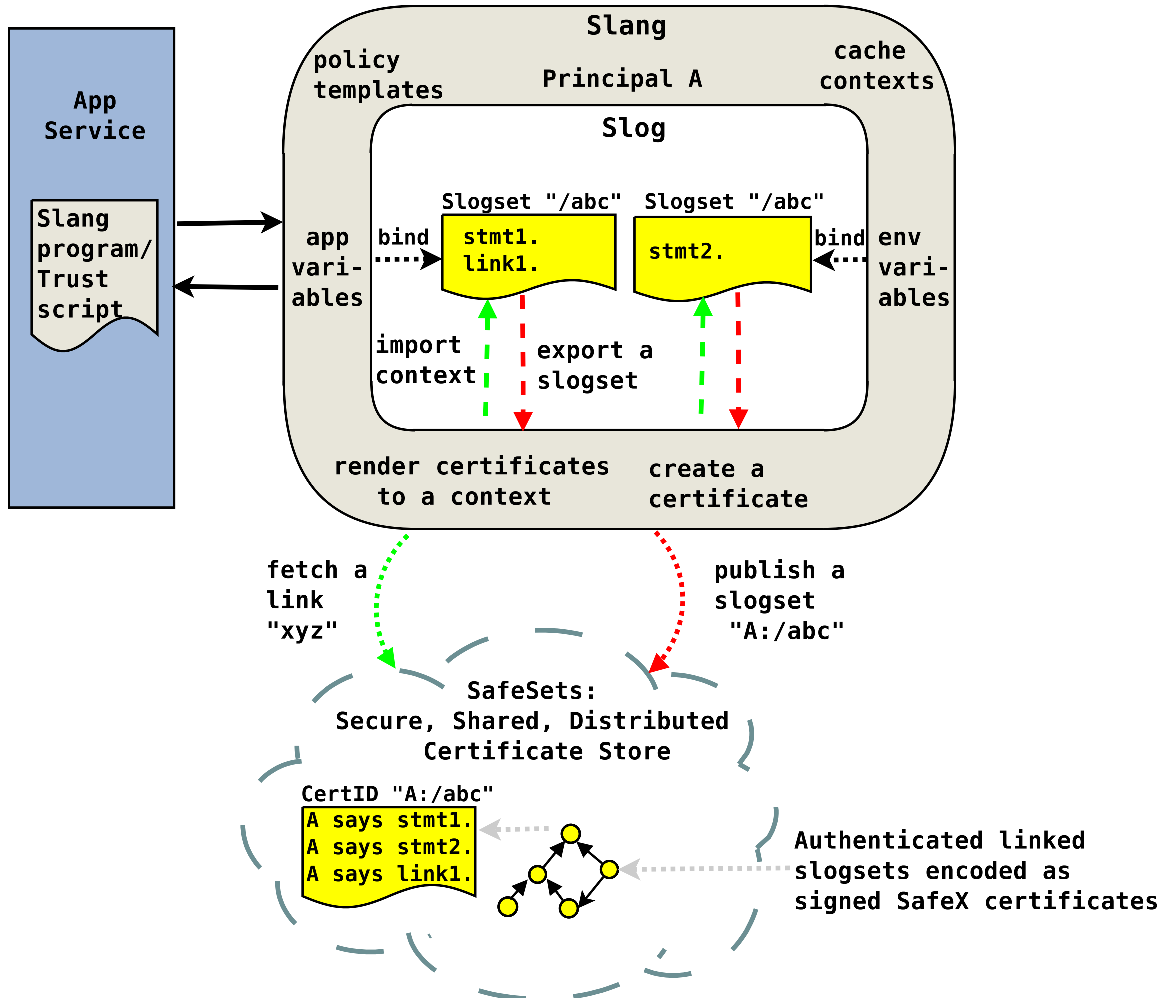


FIGURE 3.1: Overview of a design elements of SAFE. The SAFE trust logic (slog) is rooted in datalog. The credential discovery and context management are implemented outside the logic layer—slog—-using a scripting language (slang). The app service provides the slang program and initiates it via an RPC. Slang represents each proof context using an abstraction called slogset. A slogset is a set of authenticated logic statements where each statement is qualified with the identity of a principal that issued the statement. Slang provides slogset templates via libraries that capture common security policies and credentials. Slogsets may be linked together and can be exported as signed certificates into a distributed shared repository (SafeSets). The certificates are encoded using Safe expressions (SafeX), which use logic for transport. A slang program is instantiated by a principal A with two slog templates sharing the same local name `‘‘/abc’’`. The slogsets are then materialized as a SafeX certificates and published to SafeSets.

the TCB from rest of the application implementation by defining an API for trust events to use and rely on for all the trust decisions. In particular, the scripts define

how an application can use trust logic (slog), including the predicate symbols and secure name constants for its statements in slog. Each participant's TCB includes the slog prover, script interpreter, and trust scripts, which are all under its direct control: authorization is naturally end-to-end.

The application interacts with the context layer by initiating a trust script via a SAFE RESTful API. A trust script includes slang constructs for specifying the programming logic for credential retrieval, credential pruning, certificate management, and caching. In essence, a trust script interprets each request and generates the requisite content for a proof to succeed, and supplies it to the slog prover. Subsequently, using the given context, the slog prover validates the request.

A proof context may contain statements issued by many principals originating from different sources. A fundamental problem for building a proof context is automated credential discovery. SAFE provides a way to organize credentials using a novel approach based on linking slogsets. Using a link metapredicate, slogsets can be linked explicitly to reflect the delegations of the application. Linking two slogsets does not guarantee a delegation or endorsement, but may guide the credential discovery process in the right direction. Linking and retrieval is done outside the trust logic.

SAFE decouples the proof validation from proof construction or building a proof context. This decoupling is important since it ensures the design principles of linking are applicable independently of the underlying trust logic used.

The trust scripts generate slogsets from templates, using set linking and string substitution/interpolation from script parameters, environment variables, and other string variables. They also issue queries—including guard queries—to a local slog prover. A simple example of a guard query is: *"may Alice read file F?"*, where F is an object name or ID. For each query, a guard script assembles the context as a slogset. Because the scripts take parameters, they can tailor the context for each

request.

SAFE introduces a new abstraction called *slogset* that captures the set of policies and credentials that are expressed in a trust logic (slog). Each statement in a slogset is authenticated to the principal that issues the statement. Slogsets are materialized as signed certificates to enable a third party to verify independently the authentication of the source. Similarly, when a certificate is retrieved, the signature on the certificate is verified before importing the statements to a slogset. Slogsets can be linked together explicitly via a meta-predicate in logic to form credential chains capturing the delegation patterns of the application. A *proof context* is a set of slogsets that are merged together to provide the requisite input for an inference decision. Compared to Binder, a context in SAFE is more fine-grained since it is tailored per request rather than indexed by the principal. Fine-grained context management helps in optimizing the end-to-end latency as well as sharing sub-contexts across requests. Slogsets and links are described in Chapter 5.

In the next sections, we describe the two primary components of SAFE that are built upon the slogset abstraction: (i) slang, the credential management and caching layer; and (ii) SafeSets, the certificate storage layer.

3.1.1 Slang

Slang is a scripting layer for specifying the programming logic for credential retrieval, credential pruning, certificate management, and caching. In slang, slogsets are first-class objects with unique names. As discussed in Chapter 6, slang provides a high-level interface for manipulating slogsets: (i) merging two slogsets with a common name; (ii) unifying all slogsets linked together under a single context; (iii) publishing them as signed certificates; and (iv) fetching the certificates and importing them as slogsets upon verification.

The slang scripting layer is decoupled from the underlying trust logic so that both

can co-exist independently. Importantly, the decoupling of slang from the trust logic ensures that the proof validation procedure is independent of the process of credential discovery, retrieval, and caching. In addition, slog is completely agnostic to the authentication mechanism, which validates the speaker of each imported statement. Moreover, the decoupling allows evaluating the proof locally in accordance with the principle of end-to-end authorization [SRC84, HK00a]. Slog can be viewed as a certified evaluation step as in SD3 [Jim01]. In addition, slog templates can be instantiated from slang due to support of lexical-scoped variables in the language. Lastly, decoupling the layers allows SAFE to support hybrid approaches tailored to the request of the query: for instance, if the requester provides the proof, then SAFE may choose the PCA evaluation approach; if the requester provides a bearer reference to a credential chain, then SAFE may follow the default approach, which retrieves the certificates from SafeSets, builds the proof contexts, and invokes slog inference. Slang is described in Chapter 6.

3.1.2 SafeSets

SafeSets is designed as a distributed shared repository for storing certificates based on a key/value store that addresses various issues with certificate management. SafeSets offers a metadata service that stores the slogsets encoded as SAFE expressions (SafeX) signed under the issuer's keypair. Every slogset in SafeSets is identified by a globally unique self-certifying identifier (token). Slogset identifiers serve as pass-by-reference tokens where privacy is a concern. The identifiers are cryptographically unforgeable: it is computationally infeasible to obtain the local name of the set given the identifier unless the identifier was minted using the issuer's key.

SafeSets is a replaceable component within the SAFE design since the slang scripting language abstracts the details of slogset construction, export, and import from the applications. In principle, slogsets could be stored in secure web directories

maintained by the owning principals (e.g., the secure Web with TLS support can be used as a shared storage). SafeSets is described in Chapter 9.

3.2 End-to-end Trust on the Network

The soundness of logic depends on naming the entities in the trust system. The security of a slang program depends on the authentication of names and objects. SAFE supports a comprehensive set of self-certifying naming conventions for all entities in the system: principals, objects, and slogsets.

The trust logic, slog, treats the names as opaque constants. Inference is sound as long as these constants are unique and distinct, independent of the underlying naming scheme used.

To use the trust logic in a networked system the following requirements must be met: (i) network messages are authenticated as originating from a named principal; (ii) each statement in the logic is authenticated to its named speaker; (iii) each object name is securely bound to a given principal who controls the name; and (iv) to the extent that one party accepts or relies on another's statements, the parties must agree on the meaning of predicate symbols and names used in those statements.

The first requirement is met, without loss of generality, by taking principal IDs as public keys (or hashes) and transporting statements in certificates signed by the named speaker, following SPKI/SDSI. Principal IDs represents a person, institution, group, or an organization within a security domain. In slog, principals are entities to which beliefs are attributed—either directly through statements made by themselves or indirectly through inference.

In practice, a standard approach for identifying principals is through a trusted third party. On the Internet, digital identities such as email ID, usernames, or social network handles can serve as principals once the client authenticates to the service provider—typically by supplying a secret/password. Such verification assumes that

the communication channels are secure to prevent man-in-the-middle attacks on the network traffic.

The second requirement is met by the semantics of the `says` operator and its implementation in SAFE: upon fetch, each slogset is verified for its authenticity and imported into a proof context by associating each statement with its speaker—the issuer who signed the statement. Similarly, when publishing slogsets, by default all statements in the slogset are tagged with the issuer as their speaker. The issuer then signs the certificate to authenticate the statements. When a SAFE instance imports a certificate into a context, it validates the signature and checks for a match between the speaker of each statement and the authenticated issuer.

The third requirement is met by implementation of "controls" operator that binds each object name to the root principal that controls the object. The object IDs may auto-generated using GUIDs or provided using some alternative naming scheme of the controlling principal.

The naming of slogsets is controlled through local namespaces, i.e., each principal has its own object name space; request to create those names are served only through a server controlled by that principal or in possession of that principal's key. The naming of slogsets is defined at the context layer through slang constructs. The issuer selects an arbitrary string label for each slogset. A *token* or a slogset ID is a reference to a certificate formed by concatenating the principal ID and the slogset name (label). The implementation hashes the principal ID and slogset name to form the token. The slogset objects are identified globally by their tokens.

The fourth requirement is met by standards and conventions in the code. SAFE applications may define their own vocabulary of predicates (e.g., `coworker`). Note that common conventions are needed only for interoperability, but not for soundness. The soundness of SAFE inference requires only that statements are authentic and that the relevant name constants are unique and distinct. For example, if an

authorizer receives a statement spoken by a non-compliant principal (say `Mallory`), it simply ignores the statement unless there is a policy rule delegating power to `Mallory` to influence the authorizer's beliefs. It does not matter what the ignored statement means or if it is a lie. Similarly, the secure name formats are conventions in the SAFE runtime.

In general, the `says` operator abstracts from the details of authentication. Following Abadi [Aba09], when a principal `P` says `S`, `P` may transmit `S` in a variety of ways:

- on a local channel via a trusted operating system within a computer,
- on a physically secure channel between two machines,
- on a channel secured with shared-key cryptography [Sha49], or
- in a certificate with a public-key digital signature [DH06, Mer78].

3.3 Assumptions

We make certain assumptions about the threat model, SafeSets availability, principal keypairs, and credential linking.

- Although SafeSets relies on PKI for data authentication and integrity, SAFE is agnostic to the underlying crypto/technology used as long as the requirements in § 3.2 are met. In particular, the slog inference is agnostic to PKI. In domains with a central authority, it is conceivable that SAFE principals are unique names (e.g., email IDs) and the root authority provides a name to key binding. For example, in Amazon AWS [AWS15], the user accounts are mapped to cryptographic keys by the AWS root authority; in the Web, the CAs provide the URL to public key bindings.

- The SAFE instance running the inference should be a part of the owning principal's trusted computing base. All other components including SafeSets need not be trusted.
- SafeSets is configured to be a highly available storage system. With scalable key-value stores, this requirement is easily met.
- For ease of key rotation, every principal creates subprincipals to speak for them, and stores the master keypair securely off-line.

4

SAFE Logic (slog)

SAFE logic (slog) is a tractable logic based on datalog extended with support for the **says** modal operator, which qualifies each statement atom with a principal (the speaker) who says or believes it. Slog is a synthesis from our study of the literature in trust logic. Our goal in designing slog is not to reinvent trust logic, but to select and combine logic features from earlier work into a unified system for practical use and experimentation with trust logic applications and credential linking. We chose the datalog approach because datalog has a familiar syntax, well-studied formal properties, and well-developed techniques for efficient implementation. It is accessible to anyone with an elementary understanding of mathematical logic. Moreover, datalog extended with constraint domains and restricted negation is provably the most expressive of all tractable proof systems

4.1 Running Example

We use a simple access control scenario as a example to illustrate the syntax and semantics of slog. Alice is an investigative journalist who wishes to share certain

information—stored in a document `‘sensitive.pdf’` and hosted on a protected server `‘alice.org’`—only to concerned parties as per her specific trust requirements. Her policy is that these parties include: (i) Any editor of a magazine approved by Electronic Frontier Foundation (EFF); (ii) a friend Bob and his trusted coworkers. Using a system such as SAFE, Alice issues rules that represent her trust policy. Independently, Bob issues credentials asserting that Charlie is his coworker. Due to Alice's delegation of trust to Bob to identify his authorized coworkers, Charlie acquires a privilege to access Alice's sensitive document. When Charlie attempts to access it, the authorizer running SAFE on the Alice server performs an access check to determine the policy compliance of the originating request.

4.2 Motivation

A trust logic must have an unambiguous syntax and semantics. Naturally, first-order logic makes a useful foundation for trust logics. However, it is well understood that first-order logic is undecidable, i.e., there exists no proof that a logic formula is true under all possible interpretations [Gö31]. Hence, to be of practical interest, we further need to impose restrictions on first-order logic. We argue that for a trust logic to be practical for use, the proof system must have the following properties:

1. **Accountable** so that it is easy to understand which policies and credentials led to a particular access control decision, and to attribute those statements to specific principals.

2. **Decidable** so that all queries terminate with acceptance or rejection.

3. **Declarative** so that the order of issuing statements or the order of organizing atoms within a statement does not affect the access control decision.

4. **Scalable** so that an authorizer can perform access checks locally—assisted by a proof context—without having a global view of all the participants involved.

5. **Expressive** so that the policies of interest can be expressed succinctly. In particular, a practical trust logic must capture the domains for representing objects and resources. These domains include naming domains such as filename hierarchies, DNS, and IP, and delegation of names from these spaces.

6. **Tractable** so that the proof is computed within a reasonable time i.e., the proof time is polynomial in the size of policies, credentials, and requests.

4.3 Foundations

We introduce SAFE logic (slog) that extends datalog to meet these requirements. Datalog restricts first-order logic to monotonic logic (by restricting negation) and function-free Horn clauses [CGT89, AHV95]. Statements are built up from atomic formulas (atoms) and the logical operators conjunction and implication. An atom is a predicate symbol applied to a sequence of terms, which may be variables or constants representing principals, objects, or symbolic values.

Slog is based on datalog augmented with the classic "says" operator of BAN belief logic [BAN90] and ABLP logic [ABLP93]. Every atom has a first term representing a principal who "says" it (the *speaker*). In a networked system all statements are authenticated to their speakers (as described in § 3.2) before presenting them to the prover. In this respect slog is accountable.

By default, all atoms constructed and issued within a running SAFE interpreter instance are assigned a speaker, `$Self`, which is the identity associated with the instance. Consider this slog statement from our example scenario:

```
authorize(?Subject) :-
```

```
Bob: tag(?Subject, coworker),
     tag(?Subject, approvedEditor).
```

The statement is read as: "*$Self infers authorize(?Subject), for any given ?Subject, if Bob asserts that ?Subject is his coworker and $Self believes that ?Subject is an approvedEditor*". The `:-` is datalog syntax for logical implication: this statement is a *rule*. The text to the left of the `:-` is the *head* of the rule, and the text to the right is the *body*. The head is a single atom whose terms may include one or more variables (e.g., `?Subject`). Variables in slog are prefixed with `?` or `_` to avoid ambiguity with common names starting with an uppercase letter. The `coworker` is a constant representing a named attribute, e.g., as defined in [HFK+14]. More generally, the predicate in an atom or fact captures a property, attribute, role, relationship, right, power, or permission associated with the principals and/or objects named in its terms.

A rule allows the proof solver to infer that the head is true, for some substitution of its variables with constants, if the body is true under that substitution. The body is a sequence of atoms (called *goals*) separated by commas, which indicate conjunction: all the atoms in the body must be true for the rule to "fire". Slog includes some special syntax for builtin operators and constraint domains. Section § 4.5 gives the BNF grammar for slog.

In pure datalog, and therefore in slog, all variables in the head must also appear in the body. A ground *fact* is a statement with an empty body and no variables in the head. All facts are ground: the proof solver takes any fact (or any statement) in the context as true.

As noted above, each atom is bound to a speaker. In the example, the atoms in the body are prefixed with a "says" operator (`:`) naming the principal `Bob`. If an atom does not name a speaker then the default speaker is `$Self`, the principal

identity bound to the local instance who issues the statement. If principal Alice acts as the local authorizer, then `$Self` is bound to `Alice` through an invocation from higher scripting layer described in Chapter 6. In slog, terms prefixed with `$` (including `$Self`) are environment variables, which are bound before invoking the slog program.

In slog, the semantics of *says* operator is consistent with the Hand-off and Bind axioms discussed earlier. Hand-off represents transfer of authority on an atom between two principals. Consider the statement in SAFE:

```
A: p2() :- B: p2(). // atom p2() is hand-off from B to A
```

The atom `p2()` is a hand-off from principal `B` to principal `A`, i.e., `A` believes that `p2()` is true if `B` states that `p2()` is true.

Slog predicates capture named properties of entities (i.e., either principals or objects as defined in Chapter 3 and Chapter 13) and named relationships among them. An attribute in slog represents a name-value pair used to express some property of an entity. For example, `coworker` is an attribute that Charlie should possess to gain access to Alice's document.

To support scalable inference, slog adds support for delegations of authority based on attributes and *says* (see § 4.6). To support expressible trust policies (Chapter 7), slog adds constraint functional domains such as hierarchical domains and ranges, without compromising the tractability in § 4.7.

As in datalog, slog disallows user-defined function symbols and imposes certain restrictions on variables, as described above for pure datalog. Quantifiers are also restricted in slog and datalog. All variables are universally quantified. For example, the following first-order logic statement *cannot* be expressed in slog.

"For all ?X, there exists ?Y and ?Z such that ?X is between ?Y and ?Z".

```
forall(?X) exists(?Y, ?Z), between(?X, ?Y, ?Z).
```

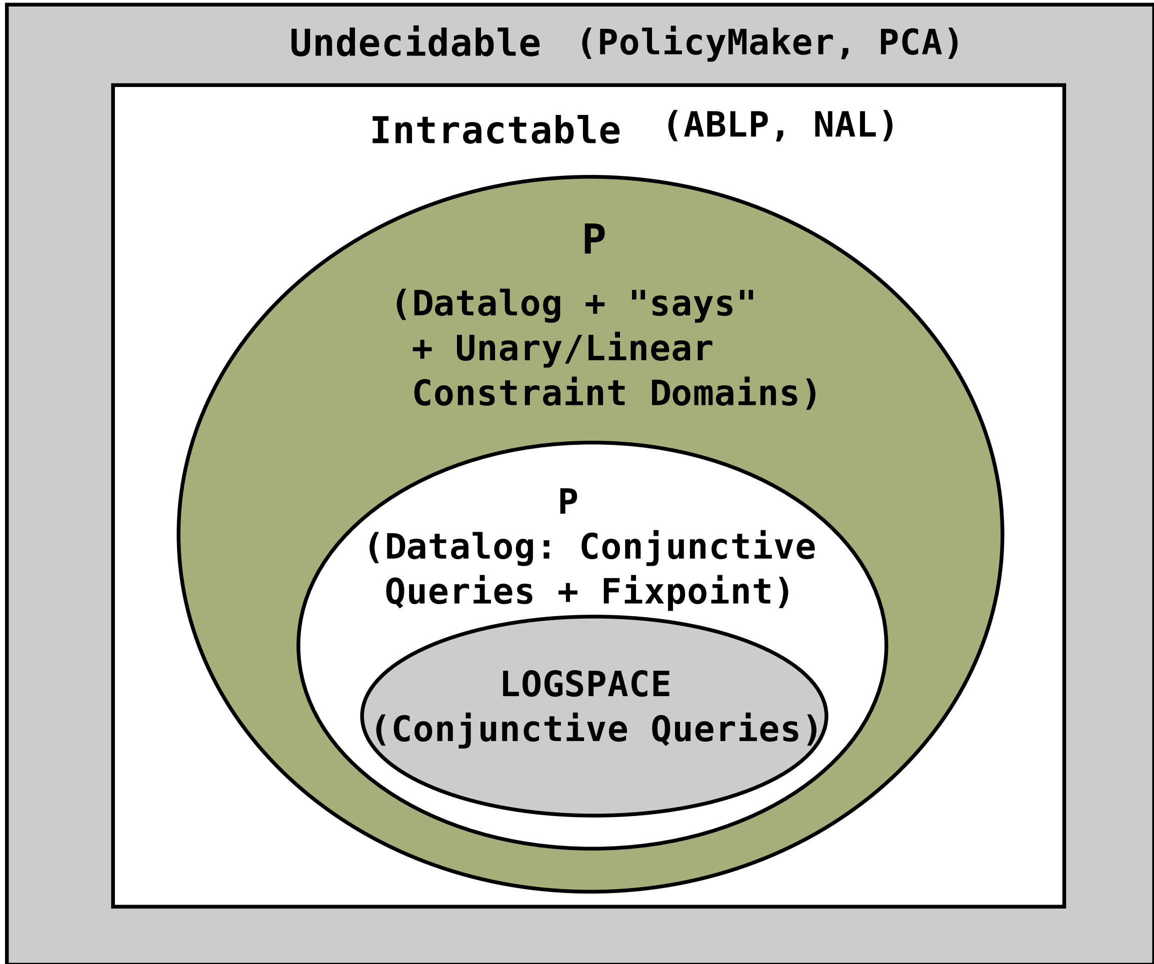


FIGURE 4.1: Descriptive complexity of trust logics.

These restrictions together with other restrictions on constraint domains ensure that that the logic is decidable and tractable. The slog parser verifies these restrictions syntactically. Thus, any slog programs that are parsed as valid are assured to implement decidable and tractable queries. § 4.7 discusses the syntactic restrictions enforced by a slog parser to achieve tractability.

4.4 Complexity

The quest for logic capturing PTIME has a long history. In 1973, Ronald Fagin proved that existential second-order logic captures non-polynomial time (NP) [Fag73]. The result is remarkable because it is a characterization of the class NP that does not invoke a model of computation such as a Turing machine. Importantly, the result gave rise to the field of descriptive complexity theory. Neil Immerman analyzed the descriptive complexity hierarchy for a variety of complexity classes include logics in the landmark paper [Imm95].

In 1979, Aho and Ullman [AU79] showed that relational calculus is unable to express the transitive closure of a given relation. They suggested extending the relational calculus by adding the least fixed-point operator, which becomes datalog.

In 1982, Immerman [Imm82] and Vardi [Var82] independently proved that least fixed point logic captures polynomial time over finite structures. The quest for tractable logic for all structures—ordered and un-ordered—is still an open research area. Martin Grohe provides a survey of the literature and discusses the recent advances in this area [Gro10].

The complexity of datalog can be analyzed from combined results of Aho-Ullman and Immerman-Vardi: "Unless P is NP, datalog(pure) precisely captures PTIME". There is no other language that can be more expressive than datalog and can run in PTIME.

Following Immerman's descriptive complexity diagrams, the detailed complexity analysis of trust logics is illustrated in Figure 4.1. Since slog represents the speaker internally as the first argument of an atom, all slog atoms are datalog atoms with arity $n + 1$. Hence, slog(pure), i.e., without constraints, is tractable. The result is not surprising since datalog(pure) is tractable. However, it is an open research question about what constraints can be added to slog without compromising tractabil-

ity [Rev98, Rev02, LM03]. We discuss the constraint domains in § 4.7. The other results of the Figure 4.1 follow from the descriptions in related work (see Chapter 2).

4.5 Syntax

We present the grammar for slog in Backus-Naur Form.

⟨*program*⟩	::=	⟨*statement*⟩ | ⟨*program*⟩
⟨*statement*⟩	::=	⟨*query*⟩ | ⟨*assertion*⟩ | ⟨*retraction*⟩
⟨*query*⟩	::=	⟨*literal*⟩ ‘`?`’ | ‘`,`’ ⟨*query*⟩ ‘`?`’
⟨*assertion*⟩	::=	⟨*clause*⟩ ‘`.`’ | ‘`assert`’ ⟨*clause*⟩ ‘`end`’
⟨*retraction*⟩	::=	⟨*clause*⟩ ‘`~`’ | ‘`retract`’ ⟨*clause*⟩ ‘`end`’
⟨*clause*⟩	::=	⟨*rule*⟩ | ⟨*groundFact*⟩
⟨*rule*⟩	::=	⟨*atom*⟩ ‘`:-`’ ⟨*body*⟩ ‘`.`’
⟨*body*⟩	::=	⟨*infixTerm*⟩ | ⟨*atom*⟩ | ‘`,`’ ⟨*body*⟩
⟨*literal*⟩	::=	⟨*atom*⟩ | ‘`!`’⟨*atom*⟩
⟨*atom*⟩	::=	⟨*speaker*⟩ ‘`:`’ ⟨*predicate*⟩ (⟨*args*⟩)
⟨*infixTerm*⟩	::=	⟨*variable*⟩ ‘`:=`’ ⟨*atom*⟩ | ⟨*atom*⟩ ‘`=`’ ⟨*atom*⟩ | ⟨*atom*⟩ ‘`=:=`’ ⟨*atom*⟩
⟨*groundFact*⟩	::=	⟨*speaker*⟩ ‘`:`’ ⟨*predicate*⟩ (⟨*constantArgs*⟩) ‘`.`’
⟨*predicate*⟩	::=	⟨*symbol*⟩
⟨*args*⟩	::=	⟨*term*⟩ | ‘`,`’ ⟨*args*⟩
⟨*constantArgs*⟩	::=	⟨*constant*⟩ | ‘`,`’ ⟨*constantArgs*⟩
⟨*term*⟩	::=	⟨*constant*⟩ | ⟨*variable*⟩
⟨*variable*⟩	::=	‘`?`’ | ‘_’ | ‘`?`’⟨*symbol*⟩
⟨*constant*⟩	::=	⟨*domainConstant*⟩ | ⟨*stringConstant*⟩ | ⟨*symbol*⟩

⟨*speaker*⟩ ::= ⟨*stringConstant*⟩ | ⟨*symbol*⟩

⟨*domainConstant*⟩ ::= ⟨*symbol*⟩`'`⟨*symbol*⟩`'`

⟨*stringConstant*⟩ ::= `"`⟨*symbol*⟩`"` | `'`⟨*anySymbol*⟩`'` | ‘`nil`’

⟨*symbol*⟩ ::= ‘`[a-zA-Z0-9]+`’

⟨*anySymbol*⟩ ::= ‘`.*`’

Slog also supports builtin functional operators as described in Table 4.1. These builtin operators are used for numeric comparisons, creating an object name, and extracting the root principal (controlling authority) from the self-certifying ID of a given object.

4.6 Attribute-based Delegation

Delegation is a fundamental operation in trust logics. Earlier logics such as ABLP [ABLP93] and NAL [SWS11] proposed `speaksFor` as a foundational primitive for delegation. However, in slog we base delegation on predicates and inference rules, following the datalog-based trust logics and RT [LMW02].

Slog also supports *attribute-based delegation* described in RT. Consider a rule where the speaker of a goal atom is a variable rather than a constant.

```
tag(?Subject, approvedEditor) :-
        tag(?EFF, effRoot),
   ?EFF: tag(?Subject, approvedEditor).
```

The `?EFF` is a principal speaking for the EFF organization. The authorizer (Alice) may import a fact endorsing a `effRoot` trust anchor.

```
tag('hash-of-eff-public-key', effRoot).
```

The rule states that the authorizer believes a predicate `tag(?Subject, approvedEditor)` is true, if it is spoken by any principal `?EFF`, possessing the specific attribute `effRoot`.

Table 4.1: Built-in operators in slog.

Syntax	Alias	Semantics	Description
=	unify	unification	Unifies two terms
:=	is	assignment	Evaluates the right expression and assigns to the left variable
=:=	compare	comparison	Evaluates the left and right expressions and compares their values
,	and	conjunction	Combines two or more atoms to form a clause
;	or	disjunction	Used in the body of a rule to specify multiple rules sharing the same rule head
:-	if	implication	Infers the left atom as true if conjunction of all atoms on the right is true
:	says	origin of source	Identifies the asserting principal for that statement
.	assert .. end	assertion	assert a statement
~	retract .. end	retraction	retract a statement
? (as suffix)	query .. end	query	query a sequence of goals and stop at the first answer
?? (as suffix)	queryAll .. end	query	query a sequence of goals and output all the answers
!	not	negation	negation operator restricted to goal terms in a query
<	lt	less than	comparison operator over numerics and hierarchical domains
<=	lteq	less than equal to	comparison operator over numerics and hierarchical domains
>	gt	greater than	comparison operator over numerics and hierarchical domains
>=	gteq	greater than equal to	comparison operator over numerics and hierarchical domains
+, -, *, /	NA	arithmetic ops	performs the arithmetic operations on numeric types. Note: Arithmetic operators are only supported through prefix notation like standard predicates.
rootID()	NA	builtin op	extract the root ID for a given object
iName()	NA	builtin op	generate a self-certifying name

This form of rule is known as *attribute-based delegation* because the delegate in the rule is identified only by its attributes. Delegation based on attributes presents several interesting properties in slog that helps to make the logic scalable, i.e., the inference relies on local policies with trust anchors rather than a global view of its participants.

Attribute-based delegation presents interesting properties that can be summarized as follows [LMW02].

1. **Decentralized attributes.** A principal may issue a statement that another principal has a certain attribute.

   A tag is useful to assert an attribute on an object or a principal (see Chapter 13 for examples).

   ```
   // Bob tags Charlie as his coworker
   Bob: tag(Charlie, coworker).

   // Bob endorses Charlie as his coworker.
   Bob: endorse(Charlie, coworker).

   // Equivalent expression with attribute as a predicate.
   Bob: coworker(Charlie).
   ```

2. **Delegation of attribute authority.** A principal may delegate authority over an attribute to another principal, i.e., the principal trusts another principal's judgment on the attribute.

   ```
   // Alice accepts ?Subject as her coworker
   // if Bob says ?Subject is his coworker
   Alice: tag(?Subject, coworker) :- Bob: tag(?Subject, coworker).
   ```

3. **Inference of attributes.** A principal issues a policy rule stating that one or more attributes can be the basis for inferences about another attribute.

```
// Alice grants read access to ?Subject on the document
// 'sensitive.pdf' if Bob says ?Subject is his coworker
// and EFF says ?Subject as an editor.
Alice: read(?Subject, 'sensitive.pdf') :-
  Bob: tag(?Subject, coworker),
  EFF: tag(?Subject, editor).
```

Code Snippet 4.1: Alice access policy based on inference of attributes based on statements by other principals.

4. **Attribute-based delegation of attribute authority.** The key to slog's scalability is the ability to delegate to unknown principals whose trustworthiness is determined based on their own certified attributes, which may be issued by one of the authorities which the authorizer accepts as a trust anchor. Consider the rule where a local authorizer, Alice, believes someone is an approved editor by delegating the authority on identifying editors to EFF, which may in turn delegate the authority to a member of EFF's board.

   ```
   tag(?Subject, approvedEditor) :-
     tag(?EFF, effRoot),
     ?EFF: member(?Delegate, effBoard),
     ?Delegate: tag(?Subject, approvedEditor).
   ```

   By delegating the authority recursively, the local authorizer avoids having to know all the board members of EFF to recognize the approved editors.

5. **Parameterized attributes for delegation.** What if the delegated attributes themselves have parameters? For example, the delegation is on the constraint that the subject possesses some asserted property by the issuer. For example, Alice accepts the judgment on the approved editors only from EFF board members, who served on the board for at least 5 years. In slog, we can express this as:

```
tag(?Subject, approvedEditor) :-
  tag(?EFF, effRoot),
  ?EFF: member(?Delegate, effBoard),
  ?EFF: service(?Delegate, ?NumYears),
  ?NumYears >= 5,
  ?Delegate: tag(?Subject, approvedEditor).
```

However, pure datalog does not allow function symbols and operations over terms. For example, `>=` is disallowed in pure datalog as well as in SPKI/SDSI linked names, which can only contain named identifiers, i.e., symbolic constants. In the next section, we describe how to add functional domains to slog but still preserve the properties of decidability and tractability.

4.7 Types and Constraint Domains

Pure datalog has limitations as a foundation for trust logics. For example, without function symbols and the supporting operators, one cannot specify attribute delegation over structured resources such as file hierarchies and DNS names; ranges such as IP addresses and dates; and discrete set domains such as object enumeration types used in blacklisting. Consider the rule where Alice would like to grant read privilege to all the documents hosted under '`alice.org`'. If '`sensitive.pdf`' is hosted under '`alice.org/secret/sensitive.pdf`', then rather than providing the exact path, Alice might like to simply say grant read access to '`alice.org/*`', which represents permissions to access an unbounded set of resources hosted under '`alice.org`'.

Slog supports and infers the basic types including numeric and symbolic string constants. Object names are by convention encoded as string constants qualified by the name of the object's controlling principal (its root principal). For example,

Table 4.2: Basic object types and collections supported in slog.

Semantics	Syntax	Description
Numeric	`42`	Integer
	`42d | 42D`	Double
	`42f | 42F`	Float
	`42.3`	Double
	`42b | 42B`	Byte
	`42s | 42S`	Short
	`42z | 42Z`	Big Integer
	`0xCAFE`	Hexadecimal
String	`cafe`	Symbolic string constant
	`‘cafe parlor’`	Symbolic string constant; string delimited by spaces can be enclosed in ‘ ’
	`h‘CAFE’`	Hex encoded string constant
	`u‘cafe’`	Base64 encoded string
	`regex‘‘C.*?’’`	Regular expression
	`r‘‘C.*?’’`	Alias for a regular expression
	`‘‘?CAFE is $Rating’’`	Interpolated string which substitutes the values for the variables `?CAFE` and `$Rating`; `?CAFE` is defined locally in the rule and `$Rating` is obtained globally from the environment
Seq	`seq‘‘[1, 2, 3, 4]’’`	Ordered sequence
	`range‘‘[1..4]’’`	Syntactic sugar for a sequence; expands to `seq‘‘[1, 2, 3, 4]’’`
	`range‘‘[a..z]’’`	Syntactic sugar for a sequence; expands to `seq‘‘[a, b, c ... z]’’`
	`range‘‘(1..4]’’`	Syntactic sugar for a sequence; expands to `seq‘‘[2, 3, 4]’’`
	`range‘‘(1..4)’’`	Syntactic sugar for a sequence; expands to `seq‘‘[2, 3]’’`
	`range‘‘[1:2:7]’’`	Syntactic sugar for a stepped sequence; expands to `seq‘‘[1, 3, 5, 7]’’`
	`path‘‘/home/user/safe’’`	Syntactic sugar for a sequence; expands to `seq‘‘[home, user, safe]’’`
	`url‘‘cs.duke.edu’’`	Syntactic sugar for a sequence; expands to `seq‘‘[edu, duke, cs]’’`
	`ipv4‘‘192.168.0.0/16’’`	Syntactic sugar for a sequence; expands to `range‘‘[3232235520..3232301055]’’`
	`ipv6‘‘2001:db8::/125’’`	Syntactic sugar for a sequence; expands to `range‘‘[h‘2001db800000000000000000000007’..h‘2001db800000000000000000000007’]’’`

a resource object with a local name '`name-generated-by-guid`' given by a root principal '`hash-of-root-principal`' is encoded as:

```
'hash-of-root-principal:name-generated-by-guid'
```

§ 6.5.3 provides a detailed example of usage of objects in SAFE. In addition, slog supports collections: sets containing un-ordered discrete elements, and sequences containing ordered functional elements. All elements of the collections must belong to one of the basic types, i.e., either numeric or string types. The object types and collections with syntactic sugar support by slog are described in Table 4.2.

The numeric types and sequences are appealing, but they come at a cost: numerics are functional, and the domain of their arguments is infinite in contrast to the finite domains of user-defined constants. To illustrate this point, consider the simple slog rule:

```
isLessThan(?X, ?Y) :- ?X < ?Y, fact(?X, ?Y).
```

If the goal `isLessThan(?X, ?Y)` is computed from left to right using SLD resolution as in slog, the goal `?X < ?Y` ranges over two infinite domains: the one for `?X` and the other for `?Y`, which makes this rule intractable. The slog compiler detects such a rule and issues an "unsafe" exception at assertion time. However, if we reorder the two atoms as

```
// < is used as infix operator here: ?X < ?Y = <(?X, ?Y).
isLessThan(?X, ?Y) :- fact(?X, ?Y), ?X < ?Y.
```

then the slog compiler accepts the rule since the existence of the atom `fact(?X, ?Y)` bounds the values for `?X` and `?Y`. Note that allowing numeric and functional domains may affect the declarative semantics, i.e., the intended meaning of the two former rules should be the same, although only one of them is accepted by the slog compiler. However, we note that the slog compiler assists the user in reordering

the rules by rejecting unsafe rules. We describe the complete set of rules for a slog program to satisfy to be "safe" (i.e., tractable) in § 4.9.

The example illustrates that the functional predicate `<` operating over numeric domains is troublesome if its range is not constrained before invoking the predicate. We introduce other functional predicates and their respective domains in slog and describe the constraints that must be satisfied to remain tractable following the work on datalog with constraints [LM03, Rev98].

- **Equality constraint domains.** A primitive constraint over basic types has the form `?X = ?Y` or `?X = constant`.

- **Order constraint domains.** A primitive constraint over basic types has the form `?X` θ `?Y` or `?X` θ `constant` or `constant` θ `?X`, where $\theta \in$ { `=, <` }.

- **Range domains.** Range domains are syntactically sugared order domains over numeric types. A primitive constraint has the form `?X = ?Y`, `?X = constant`, `?X <: range ''[`$constant_1$`..`$constant_2$`]''`, where '`<:`' is a containment operator aliased to `in`.

- **Hierarchical domains.** Hierarchical domains are syntactically sugared order domains over string types. A primitive constraint has the form `?X = ?Y` or `?X` θ `seq''[`$constant_1$`,` $constant_2$`, ...` $constant_n$`]''` where $\theta \in$ { `=, <, <=, <<, <<=` }. Operator '`<`' captures child of the `seq''[`$constant_1$`,` $constant_2$`, ...` $constant_n$`]''`, and '`<<`' captures the descendant of `seq''[`$constant_1$`,` $constant_2$`, ...` $constant_n$`]''`.

  Hierarchical domains are useful to represent structured resources such as file systems, DNS names, and principal hierarchies in an organization. Range domains are useful for IP range delegation for software-defined networks. Example policies illustrating the use of these domains are provided in Chapter 7.

4.8 Negation

Negation when combined with recursion in datalog can lead to multiple possible models [Rev98] or even undecidability [Ull94]. Negation is still an active area of research. There are well known restrictions of negation which provides stable semantics for the model and thus making logic decidable. One such restricted semantics is based on "stratified" semantics of datalog, where certain non-monotonic programs are allowed as long as they satisfy certain criteria. For example, the stratification check involves finding a cycle free edge from a negated atom to a positive atom through a global analysis of the asserted statements. The "well-found semantics" extends stratified semantics for a three-valued logic with `true`, `false`, and `undef` [AHV95]. Although the logic may be decidable through restricted negation, the computational complexity of the program depends on the number of negated atoms pertaining to a goal. Moreover, in practice, the trust logics may not have a global view of all the statements a priori, and hence global analysis of the program at run time can be prohibitively expensive. With these observations, we restrict negation in slog to goals in the authorization queries, but not assertions. SecPAL [BFG10] is another system that restricts negation to obey stratified semantics. For example, the common negation query such as blacklisting principals from an authorization decision is trivially supported using negation in query goals.

4.9 Safety in slog: Range Restriction

We present the "safety" conditions of slog, which ensure that slog programs remain tractable for all inputs. If any of these conditions are not met, the slog compiler throws an "unsafe" exception when importing or issuing the statements. In general, ensuring safety for slog programs is undecidable. The undecidability follows the reduction from safety conditions of datalog programs [Ull90]. However, some

simple sufficient conditions can be imposed, which ensures guaranteed terminations on all inputs. These conditions are sufficient but not necessary, i.e., there are some programs which may violate these conditons, but still terminate.

A slog rule is safe (range restricted) iff:

- Each distinguished variable in a rule head also appears in a relational subgoal or a constraint domain subgoal as defined in § 4.7.

```
// Unsafe since ?Subject is unbound
Alice: tag(?Who, coworker) :- Bob: tag(?Subject, coworker).
// Safe
Alice: tag(?Subject, coworker) :- Bob: tag(?Subject, coworker).
```

- A rule contains some equality goal `?X = ?Y`, where `?Y` occurs in a relational subgoal or `?Y` is bound.

```
// Safe
assign(?Y) :- ?Y = 10.

// Unsafe
assign(?X, ?Y) :- ?X = ?Y.

// Safe
assign(?X, ?Y) :- fact(?Y), ?X = ?Y.
```

- Each variable in an arithmetic subgoal other than equality appears in a relational subgoal or is bound.

```
// Unsafe; ?X and ?Y are unbound
isLessThan(?X, ?Y) :- ?X < ?Y, fact(?X, ?Y).
```

```
// Safe; ?X and ?Y are bound
isLessThan(?X, ?Y) :- fact(?X, ?Y), ?X < ?Y.
```

- All negated goals are stratified, i.e., there are no cycles. See Chapter 15 from [AHV95] on sufficient conditions for stratification.

4.10 Evaluation Procedure

Query evaluation methods for logic based languages are broadly categorized as either bottom-up approaches (also known as forward-chaining), or top-down approaches (also known as backward-chaining). Bottom-up approaches answer a query by applying the rules of a program to all the ground facts and derive new facts based on their satisfaction of rule bodies. The minimal model for the given program and ground facts are explicitly materialized as a new proof context (discussed in § 6.4); the answer to the query is then obtained through a simple unification operation over the materialized context. In contrast, top-down methods answer a query by pushing selection criteria (i.e. constants) from the query down into rules that may answer the query (i.e. rules deriving into predicates being queried), creating more (sub)queries from the atoms of these rules bodies; the subqueries are in turn answered in a similar top-down fashion.

Traditional datalog evaluation follows a bottom-up approach. The well known techniques are naive evaluation, semi-naive evaluation, and magic sets transformation [BMSU86]. In contrast to standard evaluations of datalog, we evaluate slog in the top-down fashion using SLD resolution [SS94] as in Prolog. Evaluating slog via the top-down approach is beneficial because trust logics can halt after finding one possible answer rather than producing all the feasible answers to a given query.

4.11 Limitations

Slog—by design—is restrictive and does not accept function symbols to meet the practical requirements of trust logics: decidability and tractability. Hence, not all policies can be expressed directly in slog. In particular, counting involving aggregation or `speaksFor` involving second-order logic quantification cannot be expressed directly in slog. In Chapter 6, we introduce slang through which we can support rich policies that are not directly expressible in slog.

4.12 Summary

Slog is an elementary trust logic based on datalog-with-**says**. Slog provides support for attribute-based delegation and object encoding that is sufficiently powerful to express practical trust policies. Slog supports hierarchical and linear constraint domains that can capture common structures representing file hierarchies and ranges.

5

Certified Links: Enabling Automated Credential Discovery and Policy Mobility

The flexibility of trust logics creates new obstacles to harness their power in practical distributed systems. For example, authorization in decentralized federated environments involves finding the necessary credentials that satisfy the local policy for a given access control request. A key obstacle is *credential discovery*: a trust decision may require reasoning from statements drawn from various sources, requiring a method to discover and retrieve them. In general, credential discovery is the process of finding chains of credentials that delegate authority from one or more trust anchors to the requester. Credential discovery is different from the certificate path discovery in X.509 certificates [EAH+01] since credentials in trust management systems generally have more complex meanings than simply binding names to public keys. For example, the credential chains often form a DAG, rather than a linear path as in X.509. The issues with credential discovery are summarized earlier in § 2.6.

This chapter presents a novel approach for linking slogsets (§ 5.1) that enables hybrid access control policies (§ 5.3). We also discuss how to constrain linking (§ 5.2)

and how to organize slogsets so that they can be retrieved efficiently (§ 5.4).

5.1 Explicit Linking of Slogsets

We propose a novel approach for automated discovery of credentials. Our approach uses *explicit linking* of credentials via link predicates in slogsets to build contexts containing the trust chains in advance. A `link` is a meta-predicate of the slogset and serves much like a hyperlink in HTML documents. A presence of a `link` does not guarantee a delegation or endorsement, but may guide the credential discovery process in the right direction. A further advantage of our approach is that it naturally supports caching of context sets for future decisions.

The linking procedure is distributed across the participants who issue and receive credentials—slogsets containing endorsements and delegations. Each participant collects and stores the received credentials by using meta-predicate `link` to cross reference them into credential sets that it maintains. The issuer of a credential may use `link` in the set constructor to indicate the support for the current set: the set contents may infer additional facts based on the contents of the linked set due to the presence of a delegation or an endorsement. If all issuers follow this convention, then by induction the transitive closure of any given credential contains the totality of upstream credentials that an authorizer needs to validate it—the credential's *support set*. In this way, set linking naturally forms delegation chains in the credential graph. The authorizer uses its local checker to validate that these chains lead back to one or more trust anchors according to its policies.

For example, consider the slogset issued by Bob from the example described in § 4.1. A slogset has a name and a set of logic statements enclosed in { } as shown below.

```
1 Bob: "endorse/Charlie"{
```

```
2     tag(Charlie, coworker).
3     mkLink("EFF:endorse/Charlie").
4   }
```

The detailed syntax for constructing slogsets and instantiating them via templates is discussed in § 6.2 and § 5.4. Bob issues two statements: a statement endorsing Charlie as his coworker (line 2), and a statement with `mkLink` meta-predicate adding further support for Charlie. The `mkLink` will create a `link` by passing the argument as $H_1(Issuer : H_2(Name))$, where H_1 and H_2 are hash functions, and issuer is `Self`, the principal signing the statement, and name is the argument of `mkLink`, which is `‘‘EFF:endorse/Charlie’’`. Alternatively, the issuer can precompute the slogset reference (token) and directly assert the `link` statement in a slogset.

The linked reference, `‘‘EFF:endorse/Charlie’’`, must be issued by EFF a priori. Suppose EFF has endorsed Charlie as an editor as shown below.

```
1   EFF: "endorse/Charlie"{
2     tag(Charlie, editor).
3   }
```

Now, consider the query that requires a `?Subject` to possess attributes coworker and editor endorsed by Bob and EFF respectively.

```
1   {
2     import!($Context).
3     authorize(?Subject) :-
4       Bob: tag(?Subject, coworker),
5       EFF: tag(?Subject, editor).
6     authorize(?Subject)?
7   }
```

Here, the proof context (line 2) is either supplied by Charlie with reference to a slogset issued by Bob (`"Bob:endorse/Charlie"`) or directly fetched by the authorizer given the request parameters of Charlie. An advantage of explicit set linking is that the additional support set issued by EFF is automatically fetched upon the initial reference to Bob's set—since the default fetch computes the closure of the credential graph by traversing all the links and ignoring cycles.

The bearer reference link provided by the subject (`"Bob:endorse/Charlie"`) makes the authorizer inspect the user endorsements that she received. Linked support sets make it easy for an authorizer to obtain all credentials necessary for an authorization decision by "pulling" a credential set token passed as an argument in a request, fetching the closure of the linked subsets recursively, caching them, and adding them to the proof context. Each participant is free to organize its credentials as it sees fit, possibly across multiple sets. What is important is that each issuer links sufficient support into each endorsement or delegation, and that each requester passes sufficient support to justify each request. The linked sets may contain a superset of what is required: the authorizer's slog engine searches the context for relevant content. We emphasize that in practice, a server finds frequently linked supporting credentials in its cache, and does not fetch or validate them again on each request.

In this way, set linking organizes credentials and policies into a DAG that facilitates discovery and assembly of proof contexts. We note that fetch is cycle-safe: it ignores any cycles, which do not affect the contents of the closure. The DAG is collaboratively editable: each node in the DAG is controlled by its owners, and changes to a set by its owners are visible in other sets that link to it. The sets are, in essence, materialized views for standard queries, in which the subset owners control what statements to include in the views.

5.2 Linking with Constraints

Although linking is powerful and gives flexibility to the issuers to cross reference each other's slogsets, linking naively to too many other sets may result in a large proof context at the authorizer. On the other hand, a missing link may result in an incomplete proof and invalid access for the requester. Hence, use of linked slogsets should be carefully balanced.

SAFE allows issuers to specify constraints on the links to provide flexibility to authorizers to fetch the links satisfying the local constraints as they see fit. Since `link` is a logical statement, linking can be specified as a rule rather than a ground fact. Consider the name resolution example `''cs.duke.edu''`, where multiple branches are possible from the root `''.''`. For example, consider the subdomains from the root: `''edu''`, `''com''`, `''org''`. The slogset issued by root can be specified as:

```
1  '.': "endorse"{
2    endorse('edu-ID', edu).
3    endorse('com-ID', com).
4    endorse('org-ID', org).
5    mkLink('edu-set-ID') :- subname(edu).
6    mkLink('com-set-ID') :- subname(com).
7    mkLink('org-set-ID') :- subname(org).
8  }
```

The constraint `subname(edu)` on the link is useful to curtail the credential discovery to the specified link given the query `''cs.duke.edu''`. The other links containing constraints `subname(com)`, `subname(org)` are ignored as they are not relevant to the query. The name resolution example is discussed further in § 11.3.

5.3 Hybrid Access Control

An advantage of set linking is that it naturally supports *policy mobility*. In particular, the linked logic sets may include policy rules. The receiver applies the rules automatically: if the receiver incorporates an imported rule into a query context, the inference engine uses the rule. It is safe to apply the rules: the inference engine uses any rule only to the extent that other policies of the authorizer delegate policy control to the issuer of the rule.

For instance, set linking enables an object's root principal to control the terms of the object's use by linking policy rules into an issued object credential. The root principal may vary these terms from one object to another or change them at any time. Consider the example where the owner Alice attaches the policy to the object '`sensitive.pdf`':

```
1 Alice: "controls/sensitive.pdf"{
2   grantAccess(?Subject, 'sensitive.pdf') :-
3     Bob: tag(?Subject, coworker),
4     EFF: tag(?Subject, editor).
5 }
```

An authorizer, who performs the access check on the object on the behalf of Alice, may simply import the object policy and run the inference. The authorizer's query is defined as:

```
1 {
2   import!("Alice:controls/sensitive.pdf").
3   grantAccess(?Subject, 'sensitive.pdf')?
4 }
```

In this way, policy mobility allows hybrid access control: the policies are defined either by the object's root or some trust anchor, but are supplied by the requester itself. The explicit set linking approach in SAFE makes credential discovery practical and scalable. Linked slogsets naturally support caching and pass-by-reference for credentials. SAFE provides this flexibility, which is used in GENI to implement policies specific to object types (see §11.5).

5.4 Organizing Slogsets

Each principal organizes the credentials it issues or receives into common slogsets identified by well-defined local names. Here we describe the conventions for organizing these slogsets and how they are used.

An authorizer runs a standard discovery procedure to assemble the credentials into a proof context. To obtain the credentials, the authorizer pulls one or more slogset identifiers either passed by the requester or standard identifiers that are derived from the subject, object, or other attributes of the request.

The challenge is for the concerned parties—subjects, issuers, and the delegators—to determine which statements to place in the various slogsets, so that the authorizer can accumulate all of the credentials that are required to substantiate a request. There is some flexibility for these choices to enable different policies in the authorizer. The slogsets are, in essence, materialized views for standard queries: each set has an identifiable owner, and the set owners control what statements to include in the views.

5.4.1 Docking SlogSets

For any given request, the authorizer needs the set of all credentials that are issued either directly or indirectly to the requester, together with their supporting credentials of the delegators, and the set of all policy rules describing the policy for

granting access. The process of accumulating and caching locally all of the relevant credentials to substantiate a request is called docking.

Support Sets. The set of all upstream credentials that the authorizer needs to validate for a given request. These credentials are either issued directly to the requester or assigned indirectly through delegation.

Anchor Sets. The set of all locally established policies that an authorizer needs to verify before granting access to a request. These policies are either "anchored" locally or imported from a trusted root anchor. For example, the policy defined at the object root (see § 5.3) is an example of an anchor set.

5.4.2 Materializing Slogsets via Templates

Slog provides flexibility for expressing any application specific vocabulary/predicate definitions. SAFE provides a common vocabulary for common policies via slogset templates following certain conventions. The issuers are free to choose their own names for the slogsets. What is important is that if all the issuers follow these conventions, then the proof context can be built automatically by the authorizer given the request attributes.

Each principal organizes the slogsets it issues or receives. In practice, each slogset is associated with a "type" that captures its purpose and use. The slogsets of various types are linked and retrieved according to a well-defined structure.

Issuer slogset types. These slogsets are maintained typically by the issuer.

- **Endorsement Set.** An endorsement set $\{I, E\}$ is identified by an issuer and an entity. Each issuer I places endorsement for an entity E into this set. An entity is either a subject or an object. The issuer follows the naming convention for the endorsement set as:

```
"endorse/$E"{
```

```
// slog statements
}
```

Examples:

(1) Issuer endorses Charlie as a coworker.

```
"endorse/Charlie"{
  tag(Charlie, coworker).
}
```

(2) Issuer endorses Starwars movie with rating good.

```
"endorse/Starwars"{
  tag(Starwars, rating, good).
}
```

- **Delegation Set.** A delegation set $\{I,\ S\}$ is identified by an issuer and a subject. The subject S receives the authority via delegation from the issuer I. The issuer follows the naming convention for the delegation set as:

```
"delegate/$S"{
// slog statements
}
```

Example:

Issuer Bob delegates the authority to Charlie to speak for him.

```
"delegate/Charlie"{
  speaksFor(Charlie, Bob).
}
```

- **Grant Set.** A grant set $\{I, S, O\}$ is identified by an issuer I, a subject S, and an object O for which the capability is issued. The issuer follows the naming conventions for the grant set as:

```
"grant/$S/$O"{
// slog statements
}
```

Examples:

(1) Grant a `read` capability to Charlie on 'sensitive.pdf'.

```
"grant/Charlie/sensitive.pdf"{
  grant(Charlie, 'sensitive.pdf', read).
}
```

(2) Grant an `ownership` to Charlie on 'sensitive.pdf'.

```
"grant/Charlie/sensitive.pdf"{
  grant(Charlie, 'sensitive.pdf', owner).
}
```

- **Control Set.** The control set $\{I, O\}$ is a slogset created by an issuer I that is a root for some object named by a GUID O. The object O may be created at the request of a subject principal, which we may refer to as the object's owner, and which is distinct from the root. The issuer follows the naming convention for the control set as:

```
"controls/$O"{
// slog statements
}
```

Example: Attach a policy to an object `sensitive.pdf`.

```
"controls/Bob:sensitive.pdf"{
  authorize(?Subject, 'sensitive.pdf') :-
    grant(?Subject, 'sensitive.pdf', owner).
  authorize(?Subject, 'sensitive.pdf') :-
    EFF: tag(?Subject, editor).
}
```

Subject slogset types. The receiver of the credentials—the subject—may optionally index the slogsets issued to her by adding links to the standard templates provided by SAFE. These templates and naming conventions are duals for the issuer templates discussed earlier. All the subject slogset types are a level of indirection via `link` predicates pointing to the actual issuer slogsets. Subject slogset types allow the subject to pass the credentials to the authorizer directly as in Proof-Carrying Authorization [AF99].

- **Issuer Set.** An issuer set $\{I, E\}$ is identified by the receiver E that received an endorsement from the issuer I. The issuer set is a dual of the endorsement set with the name convention as: ``issuer/$$I$''.

- **Authority Set.** The authority set is the dual of the delegate set identified by $\{S, I\}$ with name convention as: ``authority/$$I$''.

  Issuer and authority sets are useful when the receiving subject would like to renew the certificate or relinquish a credential.

- **Capability Set.** The capability set is identified by $\{S, O\}$ with the name convention: ``capability/$$O$''.

  In general, the common practice for a requester is to pass a bearer reference to the capability set to an authorizer when requesting access for an object O.

6

SAFE Language (slang)

Slang provides a high-level interface for manipulating slogsets: publishing, revoking, re-issuing, and fetching slogsets from a secure certified distributed repository (SafeSets). Since slog does not distinguish an object from a group or an attribute from a role, slang provides functional interfaces and templates that match the application programmer's intent for specifying these entities. In addition, slang provides tools for integrating with application service frameworks: transparent crypto operations to materialize slogsets into certificates and perform validation upon fetching; a caching layer for managing and tailoring proof contexts; and limited support for unrestricted delegations (`speaksFor`) and aggregation operators on slogsets.

Slang is a simple hybrid functional-logic programming language with an extended logic syntax supporting higher-order structures with nested function symbols. Slang is designed to be used as a scripting language for credential discovery, credential pruning (tailoring proof context based on authorizer's policies and the requester credentials), and certificate issuing and revocation. A slang program is a set of logic statements that structurally resembles a Prolog program rather than a datalog

program.

An important distinction between slog and slang is that slang statements may act as functions that return values, including slogsets. In general, the programs execute a functional evaluation rather than inference: evaluation follows a deterministic path with no backtracking—presuming that for each slang goal there is at most one rule with a matching head (the common case).

A slang program permits usage of higher-order constructs to process collections: lists, nested predicates, and slogsets as *first class* objects. For example, slogsets can be manipulated as values directly by assigning them to variables and passing them as arguments to other functions. Compared to slog programs, which are transported over the network (slogsets encoded as certificates), slang programs are *local to the authorizer*. SAFE considers an authorization decision as valid only if slog certifies the evaluation.

6.1 Motivation

The design of slang is motivated in part by our experience with building an authorization system for GENI.

- We used slog as an embedded language library for a general purpose language and observed the impedance mismatch between language layers. For example, the slog program is passed as a string from the host language, which results in deferring checking of the "safety" properties of slog until actual execution time.

- We observed common patterns (fetching, publishing, and renewing) for managing credentials. These operations are handled efficiently using a scripting layer with a functional style and built-in support for the slogset abstraction, including manipulating slogsets as values.

- Certain useful logical primitives such as `speaksFor` and aggregation cannot be captured at the slog layer but can be supported in a useful form at the higher layer without compromising tractability of the logical inference.

- Writing slog code directly is tedious and prone to mistakes since the principals and objects are hashed values rather than simple mnemonic names. Slang makes it particularly convenient to define policies naturally through the use of lexically scoped program variables, environment variables that capture system properties, and builtin library functions that operate on slogsets directly.

- Slang also supports programming through policy templates so that policies are written once and instantiated accordingly as per the environment context and scope.

- Lastly, slang is declarative and resembles slog closely while being expressive. Slang performs traditional scripting functions: file manipulation, escaping to the host environment for program execution, and variable substitution.

6.2 Syntax

We present the grammar for slang in Backus-Naur Form.

⟨*program*⟩ ::= ⟨*statement*⟩ | ⟨*program*⟩

⟨*statement*⟩ ::= ⟨*query*⟩ | ⟨*assertion*⟩ | ⟨*retraction*⟩

⟨*query*⟩ ::= ⟨*literal*⟩ ‘`?`’ | ‘`,`’ ⟨*query*⟩ ‘`?`’

⟨*assertion*⟩ ::= ⟨*clause*⟩ ‘`.`’ | ‘`assert`’ ⟨*clause*⟩ ‘`end`’

⟨*retraction*⟩ ::= ⟨*clause*⟩ ‘`~`’ | ‘`retract`’ ⟨*clause*⟩ ‘`end`’

⟨*clause*⟩ ::= ⟨*functionRule*⟩ | ⟨*rule*⟩ | ⟨*fact*⟩

⟨*functionRule*⟩ ::= ⟨*functionPrefix*⟩ ⟨*term*⟩ ‘`:-`’ ⟨*body*⟩ ‘`.`’

⟨*rule*⟩ ::= ⟨*term*⟩ ‘:-’ ⟨*body*⟩ ‘.’

⟨*body*⟩ ::= ⟨*infixTerm*⟩ | ⟨*atom*⟩ | ‘,’ ⟨*body*⟩

⟨*functionPrefix*⟩ ::= ‘`definit`’ | ‘`defenv`’ | ‘`defcon`’ | ‘`defguard`’ | ‘`defun`’

⟨*literal*⟩ ::= ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*predicate*⟩ (⟨*args*⟩)

⟨*infixTerm*⟩ ::= ⟨*variable*⟩ ‘:=’ ⟨*forExpr*⟩ | ⟨*variable*⟩ ‘:=’ ⟨*atom*⟩ | ⟨*atom*⟩ ‘=’ ⟨*atom*⟩ | ⟨*atom*⟩ ‘=:=’ ⟨*atom*⟩

⟨*term*⟩ ::= ⟨*constant*⟩ | ⟨*variable*⟩ | ⟨*functor*⟩ (⟨*args*⟩) | list | slogset

⟨*fact*⟩ ::= ⟨*atom*⟩

⟨*predicate*⟩ ::= ⟨*symbol*⟩

⟨*functor*⟩ ::= ⟨*symbol*⟩

⟨*args*⟩ ::= ⟨*term*⟩ | ‘,’ ⟨*args*⟩

⟨*list*⟩ ::= ‘[’ ⟨*args*⟩ ‘]’

⟨*slogset*⟩ ::= ‘{’ ⟨*slog-program*⟩ ‘}’

⟨*collection*⟩ ::= ⟨*list*⟩ | ⟨*slogset*⟩

⟨*forExpr*⟩ ::= ‘`for`’ ⟨*variable*⟩ ‘`in`’ ⟨*collection*⟩ ⟨*caseExpr*⟩ ‘`end`’

⟨*caseExpr*⟩ ::= ‘`case`’ ⟨*term*⟩ ‘`where`’ ⟨*variable*⟩ ⟨*compOps*⟩ ⟨*constantArgs*⟩ ‘:’ | ‘`case`’ ⟨*term*⟩ ‘:’

⟨*constantArgs*⟩ ::= ⟨*constant*⟩ | ‘,’ ⟨*constantArgs*⟩

⟨*compOps*⟩ ::= ‘<=’ | ‘<’ | ‘>’ | ‘>=’ | ‘=’

⟨*variable*⟩ ::= ‘?’ | ‘_’ | ‘?’⟨*symbol*⟩

⟨*constant*⟩ ::= ⟨*domainConstant*⟩ | ⟨*stringConstant*⟩ | ⟨*symbol*⟩

⟨*domainConstant*⟩ ::= ⟨*symbol*⟩'⟨*symbol*⟩'

⟨*stringConstant*⟩ ::= "⟨*symbol*⟩"

⟨*symbol*⟩ ::= ‘`[a-zA-Z0-9]+`’

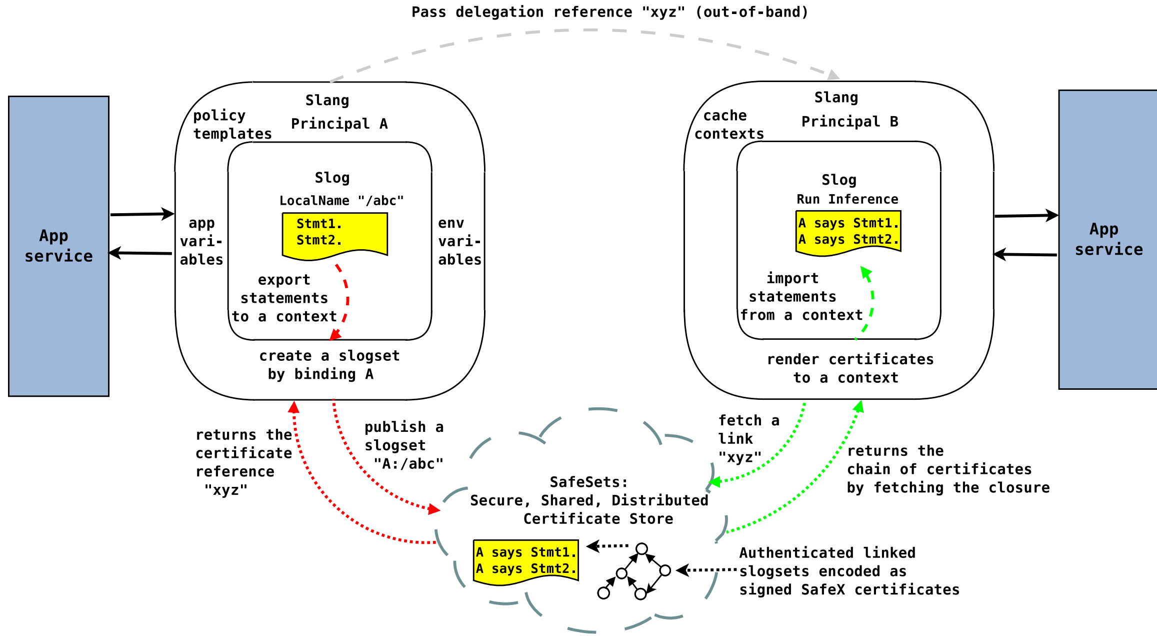


FIGURE 6.1: Overview of functionality of slang. Slang is based on functional logic programming that provides interfaces for interacting with service applications and SafeSets. These APIs include issuing and retrieving certificates, creating slogsets from templates, passing application and environment variables to slog programs, building a tailored proof context per request, and invoking slog with an appropriate proof context.

6.3 Functional Logic Programming

Slang programs are functional-logic programs written in extended logic with `def*` keywords. A consequence of treating functions as logical rules is that an invocation to a function may return multiple values since there may be more than one rule with the same head, and hence the same function definition. However, in general, the slang programs follow a deterministic path with no backtracking since most slang goals have at most one rule with a matching head.

A slang rule that starts with a `def*` keyword declares the rule as a function that returns the value of the last atom on that rule. The various slang rule types have additional behaviors to integrate with the application and with SafeSets, and to extend the scripting primitives. Table 6.1 shows the common idioms of slang

Table 6.1: Common idioms of slang and slog highlighting the differences from standard logic programming systems [WSTL12].

Syntax	Semantics	Example	Description
`:`	says	`Bob: coworker(Charlie).`	Bob says (signs) Charlie as his coworker.
`:=`	assignment	`?Value := +(2, 4).`	Evaluates the right expression and assigns the value to left variable, i.e., `?Value = 6`.
∼	statement suffix denoting a retraction	`Bob:coworker(Charlie)`∼	Bob retracts Charlie from his coworker's list.
`?`	prefix of a named variable or suffix of a statement denoting a query	`Bob:coworker(?Who)?`	Bob queries for his coworkers list.
_	anonymous variable	`Bob:coworker(_)?`	Bob queries for his coworkers list.
`$`	environment variable or a bounded variable passed from slang to slog	`?Who:coworker($Self)?`	List the speakers who said I am their coworker.
`‘‘ ’’`	string interpolation	`‘‘My identity is $Self’’`	Interpolates the string by substituting the variable `?Self` with a bounded value in scope.
`r‘‘ ’’`	regular expression	`coworker(r‘‘^C.*?’’)?`	List all coworkers starting with letter C.
{ }	mutable slogset with a given name	`‘Bob/coworkers’`{ `Bob:coworker(Charlie).`}	A credential with local name as ‘Bob/coworkers’ in Bob's namespace
{{ }}	immutable slogset with a derived name	{{ `[application/pdf] 0xcafe96f6e2f42`}}	A content object with the name derived as hash of its set contents.
`def*`	define rule as a function	`defenv EFF :- ‘const’.`	`$EFF` will resolve to a constant value string.

and slog highlighting the differences from standard Prolog syntax for logic programming. Slang provides some builtin functional features for rules prefixed with keyword tags `defenv` for initializing environment variables; `defcon` for set construction; and `defguard` for externally visible entry points to the slang program for authorizing incoming requests.

- Slang rules tagged with `defun` may include embedded native Scala code, which is compiled on-the-fly during load (or reload). This feature allows seamless interoperability with the host language and uses its libraries for implementing

complex functions. For example, the builtins in the slang library for string interpolation, regular expressions, crypto operations, and networking are implemented using `defun` and native scala code.

The function arguments are passed through the native Scala code enclosed in `` ` ` `` as string arguments similar to invoking the `main(args: Array[String])` function. Within the Scala code, the slang variables are accessed with the prefix `$` instead of `?` since all the variables are bound before invoking the native code. Following Scala conventions, the last expression is the return value of the function call, which is transformed into a constant atom in slang.

Example:

```
defun times(?X, ?Y) :-
  spec('multiply ?X and ?Y and return the value'),
  `
     $X.toInt * $Y.toInt
  `
end
```

- `definit` is invoked when a slang program is initially loaded. It invokes all the goals specified in its declaration.

```
definit ?X := times(2, 4), times(?X, 8).
// Results in evaluating the goal terms giving us the
// result 8, 32 respectively.
```

- `defenv` initializes environment variables via late binding, i.e., at the first reference to the variable. The environment variables declared through `defenv` are globally scoped but may be shadowed by the lexically scoped variables defined in rules with local scope.

Table 6.2: Builtin variables in SAFE and slang. The variable `?Selfie` is initialized using `defenv` in a slang program to set the issuer's keypair. The variables `?Self` and `?SelfKey` are computed based on the `?Selfie` value. The variables `?Speaker`, `?Subject`, `?Object`, and `?BearerRef` are passed as default arguments from a SAFE application to the guard function associated with a given request.

Variable Name	Description
`?Self`	hash of the issuer's public key
`?Selfie`	issuer's key pair
`?SelfKey`	issuer's public key
`?Subject`	hash of the requester's public key
`?Speaker`	hash of the server proxy (principal) that is making a request on the behalf of the subject (end-user) principal
`?Object`	object IID for which the access is requested
`?BearerRef`	set identifier passed by the requester

Example:

```
(* Selfie initializes $Self and $SelfKey, the hash of the public key
 * and the public key value respectively.
 *)
defenv Selfie :-
  spec('load the key pair for the issuer'),
  principal('issuer_keyPair.pem')
end
```

In general, the issuer/authorizer's keypairs are declared as environment variables by initializing the variable `Selfie`. `Selfie` initializes `Self` and `SelfKey`, which contains the identity/fingerprint of the issuer/authorizer and public key respectively. The builtin variables in slang are described in Table 6.2.

- A `defcon` rule creates or modifies a named slogset and returns its value—a *set constructor*. These rules should end with a set of slog statements enclosed in `name{}`, where `name` is the local name (label) of the slogset assigned by its is-

suer. The constructed slogset is materialized as a certificate upon a subsequent export, e.g., a post to SafeSets.

The slogset statements may contain some predefined predicates such as `link/1`, `subject/2`, `issuer/3`, `validity/3`, `signatureAlgorithm/1`, which are meta-predicates that are interpreted by slang for encoding slogsets as certificates. In particular, `link` is useful to reference another slogset as discussed in Chapter 5. The other predicates for certificate issuing and validation are discussed in Chapter 9.

Example:

```
defenv Charlie :- 'hash-of-Charlie-PK'.
defcon makeSlogSet() :-
  spec('create a simple set with a local name coworker'),
  "endorse/$Charlie"{ // the local name of this slogset 'endorse/Charlie'
     endorse("$Charlie", coworker).
     mkLink("controls/coworker-group").
  }
end
```

- A `defguard` rule is where the guard queries are defined and access-check operations are performed. The rule imports the assembled proof context for each query through the `import!` predicate, which takes a set reference as an argument, and performs certified evaluation through slog.

  Example:

```
defguard authorizer(?Subject, ?Object, ?Priv) :-
  spec('guard to check the authorization access for the subject'),
  ?Controller := rootID(?Object),
```

```
  ?ProofContext := fetch("?Controller:controls/?Object"),
  {
    import!("$ProofContext").
    grant($Subject, $Object, $Priv)?
  }
end
```

In addition, the `defguard` rule acts as an external entry point to the slang program, invoked via a REST API to check policy compliance (e.g., for an application-level request) when the slang interpreter is configured to run as a *s*ervice. We discuss running *s*lang as a service in Chapter 10.

6.4 Context-layer Functions

Slang provides several useful functions as a standard library to operate directly on slogsets. These functions include aggregate functions such as `length()`, `max()`, `min()`; parse functions operating on a safeset such as `getName()`, `getSpeaker()`, `getSubject()`, `verifySignature()`; functions to compute hash and load/generate keypairs; and support of `speaksFor` delegation. Table 6.3 presents the summary of context-layer functions supported in slang.

Many of the builtin functions can be directly coded in slang itself. For example, slang treats slogsets as collection types similar to Prolog lists. Code Snippet 6.1 and 6.2 show the `length()` function expressed in slang for lists and slogsets. `length()` computes the total number of elements present in the given collection. The syntax for a slogset collection is similar to a list collection type except the change of notation for representing the slogset { } rather than the bracketed list syntax `[ ]`.

Slang provides a builtin `for()` to verify `speaksFor` delegations. The verification follows a two-step process. First, slang checks whether the issuer and speaker are

```
length([], 0).
length([?H|?T], ?N) :- length(?T, ?N1), ?N := +(?N1, 1).
```

Code Snippet 6.1: length() function for lists expressed in slang.

```
length({}, 0).
length({?H|?T}, ?N) :- length(?T, ?N1), ?N := +(?N1, 1).
```

Code Snippet 6.2: length() function for slogsets expressed in slang.

different while fetching the set. If they are the same, then slang validates the signature and imports the statements with the speaker as the issuer. If they are different, then slang looks for a slogset reference that provides the proof context with this statement:

```
  ?Issuer: speaksFor(?Speaker, ?Issuer).
```

If such a delegation is found and is valid, then slang concludes that the issuer has an unrestricted delegation for the speaker. Slang then accepts statements in the slogset that are spoken by the named speaker.

In SAFE, `speaksFor` is useful for assigning multiple owners to a slogset, and delegating authority over principal hierarchies. For example, the post authorization of SafeSets relies on the `speaksFor` operation (see Chapter 9).

In general, slang is extensible and `defun` makes it straightforward to add new functions to the library. However, note that the flexibility can be misused; e.g., writing a function with an endless loop. But unlike slog code which can be arbitrary passed between participants across the network, the authorizer has complete control over the slang code running locally.

Table 6.3: Context-layer functions in slang.

Function Name	Description
`parseCertificate(?Certificate)`	parse a certificate to an internal representation and return a reference
`getName(?CertificateRef)`	get the slogset name on the certificate
`getIssuer(?CertificateRef)`	get the issuer's ID on the certificate
`getSubject(?CertificateRef)`	get the subject's ID on the certificate
`verifySignature(?CertificateRef)`	verify the signature on the certificate
`getID(?SubjectID, ?Name)`	get the slogset identifier with issuer as subject id and local name as name
`mkLink(?Name)`	get the slogset identifier with issuer as `Self` and local name as name
`import!(?Name)`	import the slogsets into the current proof context
`iName()`	generate a self-certifying name; used for assigning names to objects
`rootID(?IID)`	extract the root ID of the self-certifying identifier, i.e., the ID of the principal which assigns the name to the object represented by this IID
`fetch(?BearerRef)`	fetch a transitive closure of slogset ref by traversing all the links
`fetchSRN(?SetRef, ?SRN)`	fetch a transitive closure of slogset ref by traversing the links as guided by the safe resource name (SRN)
`post(?SlogSetRef)`	post the set contents referenced by ?SlogSetRef and return the certificate reference
`for(?A, ?B, ?PC)`	verify whether the subject `A` `speaksFor` the subject `B` given the proof context reference `PC`

6.5 Policy Templates

Slang provides a set of common policy templates via libraries that are useful to issue certificates and write guard queries. Table 6.4 provides the functional API for these templates specified in Scala. These APIs are directly implemented in slang as trust-logic functional rules.

The templates follow the conventions for slogset organization and naming as discussed earlier in § 5.4.2. For example, the issuer slogset types are: endorsement set,

Table 6.4: Policy APIs for slang specified in Scala. These APIs are implemented as trust-logic functional rules in slang.

API	Description
`endorse(entity: IName, attr: Symbol): IName`	endorse an entity by asserting an attribute
`endorseWithValue(entity: IName, attrName: Symbol, attrValue: Symbol): IName`	endorse an entity by asserting an attribute name-value pair
`delegate(subject: ID): IName`	issue delegation to subject via 'speaksFor'
`grantCap(subject: ID, object: IName, priv: Symbol): IName`	grant a capability for a subject on an object
`grantCapByPrefix(subject: ID, prefix: IName, priv: Symbol): IName`	grant a capability for a subject on all objects matching a prefix
`delegateCap(subject: ID, object: SCN, priv: Symbol): IName`	grant a delegatable capability for a subject on an object
`delegateCapByPrefix(subject: ID, object: SCN, priv: Symbol): IName`	grant a delegatable capability for a subject on all objects matching a prefix
`createGroup(): IName`	create a group and return the self-certifying name that identifies the group
`createObject(): IName`	create an object and return the self-certifying name that identifies the object
`createRole(roleName: Symbol, subject: ID, slogSetRef: IName): IName`	create a role for the subject and assign the credentials via 'slogSetRef'
`grantMembership(subject: ID, group: IName): IName`	grant membership for a subject in the group
`attachSubGroup(targetGroup: IName, parentGroup: SCN): IName`	grant membership for 'targetGroup' in 'parentGroup' upon which the 'targetGroup' becomes a subset of 'parentGroup'. The caller must control 'parentGroup'.
`resolveName(pathName: SRN): ID`	resolve a multicomponent pathname, which may cross domain boundaries
`attachPolicy(policy: Policy, object: IName): IName`	attach a policy to an object; the caller must control the object
`attachCredential(credential: Credential, object: IName): IName`	attach a credential to an object; the caller must control the object
`attachGroup(group: IName, object: IName): IName`	attach a group to an object; the caller must control the object
`attach(slogSetRef: IName, object: IName): IName`	attach a slogset to an object; the caller must control the object
`checkAccessCap(subject: ID, object: IName, priv: Symbol, slogSetRef: Option[ID]): Boolean`	check access capability 'priv' for a subject on an object. The subject may optionally provide the slogset reference set.

delegation set, control set, and revoke set. Here we provide the slang implementation for these templates.

6.5.1 Endorsement

An endorsement set is identified by its issuer and an entity for which the endorsement is issued. Issuing an endorsement asserts the subject attributes. Code Snippet 6.3 and 6.4 show the slang templates for endorsement set.

```
1 defcon endorse(?Entity, ?Attr) :-
2   spec('endorse an entity ?Entity as ?Attr'),
3   ''endorse/$Entity''{
4     endorse($Entity, $Attr).
5   }
6 end
```

Code Snippet 6.3: Endorse an entity by asserting an attribute.

```
1 defcon endorseWithValue(?Entity, ?AttrName, ?AttrValue) :-
2   spec('endorse an entity ?Entity as ?AttrName and ?AttrValue'),
3   ''endorse/$Entity''{
4     endorse($Entity, $AttrName, $AttrValue).
5   }
6 end
```

Code Snippet 6.4: Endorse an entity by asserting an attribute name-value pair.

6.5.2 Delegation of Authority

Delegation of authority can be done in a restricted form by issuing facts or rules for specific attributes (see § 4.6), or in an unrestricted form using `speaksFor`. For example, consider the endorsement from Code Snippet 6.3. Instead of directly issuing the endorsements, the issuer can assert a rule for an attribute-based delegation from a trusted entity. Consider line 4-6 in Code Snippet 6.5. The rule states that the

issuer delegates the authority on any attribute to a subject `?Delegator` possessing an attribute `?TrustedRoot`.

```
1 defcon endorseByDelegation(?TrustedRoot, ?Attr) :-
2   spec('delegate authority on an attribute to a trustedRoot'),
3   ''endorse/delegate/$Attr''{
4     endorse(?Entity, $Attr) :-
5       endorse(?Delegator, $TrustedRoot),
6       ?Delegator: endorse(?Entity, $Attr).
7   }
8 end
```

Code Snippet 6.5: Attribute-based delegation.

Slang provides a more powerful delegation via `speaksFor`, which may be used to delegate authority over all statements to a delegate. These delegations are issued as ordinary slog statements but are interpreted by the slang context layer to import the valid statements into the proof context, as described above.

A delegation set is identified by an issuer and a subject. A delegation set may grant unrestricted authority via `speaksFor`. Code Snippet 6.6 shows the slang template for delegation set.

```
1 defcon delegate(?Subject) :-
2   spec('delegate authority to a subject'),
3   ''delegate/$Subject''{
4     speaksFor($Subject, $Self).
5   }
6 end
```

Code Snippet 6.6: Delegate unrestricted authority to a subject via `speaksFor`.

The delegation policies are interpreted by slang at the context layer to import the relevant statements into the proof context as discussed earlier in § 6.4.

6.5.3 *Objects and Capabilities*

An object is a logical entity with a self-certifying name, about which statements may be made in the logic. In general, the object name is issued by a root entity that creates the object and has authority over it; any principal may assert the existence of objects and serve as the root entity (controlling authority) for those objects. The root entity may issue statements to delegate powers over an object to another subject and may attach policies on how the object access should be granted based on those statements. An object's root principal *"controls"* the object in the sense of BAN logic.

```
1  defcon createObject(?Subject, ?ObjectType) :-
2    spec('create an object and assign subject as an owner'),
3    ?Object := iName(),
4    ?ObjectPolicyRef := "policy/standardObjectPolicy",
5    ''controls/$Object''{
6      endorse($Subject, $Object, owner).
7      object($Object, $ObjectType).
8      mkLink($ObjectPolicy).
9    }
10 end
```

Code Snippet 6.7: Create an object by generating a self-certifying name, assign the subject as the owner, and also attach the standard object policy"

The root entity can define and attach standard policies on an object. For example, consider the Code Snippet 6.8. The policy delegates controlling authority to the `owner` using a `controls` predicate and then issues a rule via attribute-based delegation. The rule asserts that the `delegateCap` is inferred transitively based on the delegation chains starting from the object's owner.

A capability is a privilege issued by a subject that controls the object to another subject. The issuance of capabilities is captured by grant set types. A grant set is identified by an issuer, a subject, and an object for which the capability is granted.

```
1  defcon createStandardPolicy() :-
2    spec('create a standard access policy on an object'),
3    ''policy/standardObjectPolicy''{
4      // An owner controls the object
5      controls(?Subject, ?Object) :- owner(?Subject, ?Object).
6
7      // The delegation privileges are transitive and attribute-based
8      delegateCap(?Subject, ?Object, ?Priv) :-
9        delegateCap(?Delegator, ?Object, ?Priv),
10       ?Delegator: delegateCap(?Subject, ?Object, ?Priv).
11   }
12 end
```

Code Snippet 6.8: Standard access policy on an object.

Code Snippet 6.9 shows the grant privilege template for a single object. Code Snippet 6.10 show the grant privilege template for a group of objects under a directory. `grantCapByPrefix` finds all the objects matching a prefix and invokes `grantCap` for member in the group. The `for` expression is used here to traverse the list of objects in the group. We discuss `findMembers` in § 6.5.4.

```
1  defcon grantCap(?Subject, ?Object, ?Priv) :-
2    spec('grant privilege ?Priv to subject ?Subject on object
     ↪  ?Object'),
3    ?RootID := rootID(?Object),
4    ?SetRef := getID(?RootID, "controls/?Object"),
5    ''grant/$Subject/$Object/$Priv''{
6      grantCap($Subject, $Object, $Priv).
7      link($SetRef).
8    }
9  end
```

Code Snippet 6.9: Grant privilege ?Priv to subject ?Subject on object ?Object.

In addition to granting a capability on a subject, the issuer can also grant *delegate* capability to the subject for a specific privilege that is granted. A subject with `delegateCap` privilege can further recursively delegate the capability to other

```
defcon grantCapByPrefix(?Entity, ?Prefix, ?Priv) :-
  spec('grant privilege ?Priv to a subject or a group on all objects
   ↪  matching the ?Prefix'),
  ?RootID := rootID(?Prefix),
  // get the object list under the prefix
  ?GroupID := getID(?RootID, "controls/?Prefix"),
  ?ObjectList := findMembers(?GroupID, ?RootID),
  // issue grantCap for each object
  for ?X in ?ObjectList
    case member(_, ?Subject, ?GroupID):
      grantCap(?Subject, ?Object, ?Priv)
  end
end
```

Code Snippet 6.10: Grant privilege on all objects matching a prefix for the given subject or a group.

subjects. Code Snippet 6.11 shows the `delegateCap` template policy.

```
defcon delegateCap(?Subject, ?Object, ?Priv) :-
  spec('grant privilege ?Priv to subject ?Subject on object ?Object'),
  ''delegate/$Subject/$Object/$Priv''{
    delegateCap($Subject, $Object, $Priv).
  }
end
```

Code Snippet 6.11: Grant delegate privilege ?Priv to subject ?Subject on object ?Object.

6.5.4 Groups

A group is a virtual object in SAFE with a self-certifying name. Groups are fundamental to represent access control lists, roles, and hierarchies of resources and principals. Groups are created similarly to other objects as in Code Snippet 6.7 with the object type passed as **'group'**. Two common queries on groups are: (i) check whether a subject is a member of a group (ii) list all the members given the group identifier. Code Snippet 6.12 and 6.13 illustrate the membership templates on a group.

The access control policy for a standard group is similar to the access control

```
1  defguard isMember(?Subject, ?GroupID) :-
2    spec('check whether ?Subject is a member of the group ?GroupID'),
3    {
4      import!($GroupID).
5      member($Subject, $GroupID)?
6    }
7  end
```

Code Snippet 6.12: Check whether a subject is a member of a group.

```
1  defguard findMembers(?GroupID) :-
2    spec('find all members of the group'),
3    {
4      import!($GroupID).
5      member(?Subject, $GroupID)??
6    }
7  end
```

Code Snippet 6.13: Find all members in a group.

for the standard object as in Code Snippet 6.8. An owner is a member of a group and possesses `'invite'` privilege to request other subjects to join the group. The `delegateCap` is a transitively defined using attribute-based delegation. Code Snippet 6.14 shows the standard group policy template.

A group may contain subgroups which are references to other groups. Hence, a membership in a group recursively assigns membership in all the subgroups that are contained within the group. Code Snippet 6.15 shows example policies for creating subgroups and granting membership for a subject and a subgroup within the parent group.

6.5.5 Attach Policy

The objects can have access policies defined by the subject that controls the object. These polices are `attached` to the object similar to access control lists (ACLs) in Unix. Attaching a policy may simply involve adding a `link` that contains the actual

```
1  defcon createGroup(?Owner, ?Group, ?GroupPolicy) :-
2    spec('create a group, assign ownership, and attach the group
     ↪  policy'),
3    "controls/?Group"{
4      owner($Owner, $Group).
5      object($Group, group).
6      link($GroupPolicy).
7    }
8  end
9
10 defun createStandardGroup(?Owner) :-
11   ?Group := genCertifiedName($Self),
12   ?GroupPolicy := standardMembershipPolicy(),
13   createGroup(?Owner, ?Group, ?GroupPolicy)
14 end
15
16 defcon standardGroupPolicy() :-
17   'policy/standardGroupMembership'{
18
19     // A owner is a member of the group
20     member(?User, ?Group) :-
21       owner(?User, ?Group).
22
23     // owner possesses capability to invite other members
24     delegateCap(?User, ?Group, invite) :-
25       owner(?User, ?Group).
26
27     // A ?User has capability ?Priv if
28     //   there is a ?Delegator that has capability ?Priv
29     //   and ?Delegator has granted delegateCap on ?Priv to ?User
30     delegateCap(?User, ?Group, ?Priv) :-
31       delegateCap(?Delegator, ?Group, ?Priv),
32       ?Delegator: delegateCap(?User, ?Group, ?Priv).
33   }
34 end
```

Code Snippet 6.14: Create group with standard delegatable capabilities.

```
defcon grantMembership(?Subject, ?Group) :-
  ''controls/$Group/member/$Subject''{
    member($Subject, $Group).
  }
end

defcon subGroupPolicy() :-
  ''policy/standardSubGroupPolicy''{
     member(?User, ?Group) :-
       subGroup(?TargetGroup, ?Group),
       ?SubGroupOwner := rootID(?TargetGroup),
       ?SubGroupOwner: member(?User, ?TargetGroup).

     delegateCap(?User, ?Group, ?Priv) :-
       subGroup(?TargetGroup, ?Group),
       ?SubGroupOwner := rootID(?TargetGroup),
       ?SubGroupOwner: delegateCap(?User, ?Group, ?Priv).

   }
end

defcon createSubGroup(?TargetGroup, ?ParentGroup, ?TargetGroupPolicy) :-
  ''controls/$ParentGroup/member/$TargetGroup''{
    subGroup($TargetGroup, $ParentGroup).
    link($TargetGroupPolicy).
  }
end

defcon createStandardSubGroup(?TargetGroup, ?ParentGroup) :-
   ?TargetGroupPolicy := subGroupPolicy(),
  ''controls/$ParentGroup/member/$TargetGroup''{
    subGroup($TargetGroup, $ParentGroup).
    link($TargetGroupPolicy).
  }
end
```

Code Snippet 6.15: Grant membership for ?TargetGroup in ?ParentGroup.

policy or directly asserting the policy itself. Code Snippet 6.16 shows the `attach` template.

6.6 Limitations

The implementation of `speaksFor` is not fully transitive in that the onus is on the issuer to supply the proof context so that the speaker is verified. In this sense, the `speaksFor` implementation follows the approach of PCA, where the requester should assemble the proof a priori. From our experience, in practice, this restriction is not

```
defcon attach(?Policy, ?Object) :-
  ''controls/?Object''{
    link($Policy).
  }
end
```

Code Snippet 6.16: Attach policy to an object.

an issue since full delegation of authority is limited to one level for most scenarios. Where delegations of larger depths are required, SAFE recommends the attribute-based delegation which can scale to arbitrary delegation chains. An advantage of our approach is that `speaksFor` is treated as an ordinary slog predicate at the slog layer, and so does not affect the complexity of the prover. Slang applies speaksFor by accepting imported statements as spoken by the issuer even if they are signed by the delegate, on the condition that the delegate indicates in the certificate that it is speaking for the issuer.

The application's slang code controls the construction procedure for set linking. The standard procedure for fetching and importing proof contexts is not a fully general solution for credential discovery. For example, it presumes that endorsements and capability delegations are *explicit*—the delegate knows that it has been issued and receives a linkable identifier. More generally, the linking structure in the slang program constrains how an authorizer may ultimately use the credentials, by bounding the credential scope for each authorization decision.

There are several alternatives to address these issues to the extent that they are problematic for specific applications. An authorizer may link any additional sets into the proof context. For example, it may link *anchor sets* of implicit endorsements issued by trusted anchors, even if the endorsee is not aware of them. For example, the authorizer might query a locally trusted whitelist service to validate a user or identity provider. Note that our model also supports a limited form of set query: an authorizer who knows a set's label template, parameters (values substituted in the

template), and issuing key may synthesize the identifier and fetch the set. We leave open the possibility that the authorizer might augment the context with one or more standard queries, if needed by its policy. Note also that an issuer can easily salt its slogset labels if privacy is a concern. We discussed the alternatives for hybrid based access control in § 5.3.

6.7 Summary

Slang is a hybrid functional-logic programming language providing common policy templates and operators for fetching, pruning, and publishing slogsets. Slang is attractive since it abstracts the crypto operators and provides a simplified extended logical interface for writing polices. Slang makes credential discovery and pruning proof contexts programmable. In addition, slang is extensible: it is possible to add to the library of slang functions by implementing new slang functions in native code (Scala) through `defun`. The integration of the host language opens up wide range of libraries available to slang, which makes it a powerful extension language for integrating with application code.

7

Policy Expressivity in SAFE

In this chapter, we highlight common security and trust policies that can be expressed in slog naturally. A natural question that arise is: whether slog is powerful enough to express the common security policies? This chapter demonstrates that slog is sufficiently powerful in expressing RT_0, policies with constrained domains, and other policies such as mandatory access control and discretionary access control.

7.1 ABLP "for" operator

The `for` is expressed in slog as simple logic statement:

```
B: for(A) :- A: speaksFor(B, A). // B for A if A says B speaksFor A.
```

However, the verification in SAFE follows a two step process. First, slang checks whether the issuer and speaker are different while fetching a slogset. If they are same, then slang validates the signature and imports the statements with the speaker as the issuer. If they are different, then slang looks for a slogset reference that provides the proof context with this statement:

```
1   ?Issuer: speaksFor(?Speaker, ?Issuer).
```

Slang runs certified evaluation through slog to verify the existing of `speaksFor` delegation. If such delegation is found and the inference is valid, then slang imports all in that slogset as spoken by the speaker rather than the issuer.

7.2 SDSI's linked namespaces in SAFE

Consider an extended name in SDSI or a linked role in RT with the certificate of the form $P_1\ n_1\ n_2$, where P_1 is the principal (public key), n_1 and n_2 are local names bound to that key. The expression implies the set of all members bound to $P_x\ n_2$ such that P_x is bound to $P_1\ n_1$. Informally, the expression can be viewed by splitting into following steps:

- There is a group of principals bound to $P_1\ n_1$ with group name as P_x.
- There is another group of principals bound to $P_x\ n_2$ with group name as P_y.
- The combined expression, $P_1\ n_1\ n_2$, states that all principals of the group with name P_y are subset of the group with name P_x.

Alternatively, the group $P_x\ n_2$ (implicit) is *linked* to group $P_1\ n_1$ via $P_1\ n_1\ n_2$. The extended names in SDSI or linked roles of RT can be expressed in SAFE with explicit linking and explicit grant of membership. For example, the membership is granted by asserting a statement `member(`P_x`)` forming a credential. The credential is then linked to include the group membership policy.

Similarly, linking of non-local namespaces is also explicit in SAFE. For example, consider SDSI's certificate of the form $P_1\ n_1 \rightarrow P_2\ n_2$, where P_1, P_2 are principals and n_1, n_2 are local names bound to these principals respectively. The expression states that all members bound to $P_2\ n_2$ are also bound to $P_1\ n_1$, with P_x replaced by P_2. Unlike SDSI's *implicit* linking, in SAFE, the membership is asserted explicitly

```
// SDSI's extended linked name P1n1n2 expressed in SAFE

// Px's namespace
n2 {
  member(P2).
  member(P3).
}.

// P1's namespace
n1 {
  link(Px.n2).
  member(?X) :- trustAnchor(?Y), ?Y: member(?X).
  trustAnchor(Px).
}.
```

Code Snippet 7.1: SDSI's extended names and linking expressed in SAFE.

through the `member()` predicate (line 5-6) and linked using explicit predicate via `link()` (line 11) along with the membership policy (line 12).

7.3 Expressing RT_0 in Slog

RT_0 is the simplest of RT languages that subsumes SPKI/SDSI semantics. Li [LMW02] implemented RT_0 by translating to Prolog. To illustrate the simplicity of slog, we provide the slog semantics of RT_0 type rules as described in [LMW02, LWM03].

1. $A.r \longleftarrow B$

   Principal B is a member of role $A.r$.

   ```
   A: member(B, r).
   ```

2. $A.r \longleftarrow B.r_1$

   All members of role $B.r_1$ are also members of role $A.r$. Credentials of this form can be used to delegate authority over the membership of a role to another principal.

   ```
   A: member(?Subject, r) :- B: member(?Subject, r1).
   ```

3. $A.r \longleftarrow B.r_1.r_2$

   For each member of C of $B.r_1$, all members of $C.r_2$ are members of role $A.r$. Credentials of this form are used to delegate authority over the membership of a role to all entities that have the attribute represented by $B.r_1$. The expression $B.r_1.r_2$ is called a *linked* role.

```
A: member(?X, r) :- A: member(?X, r1), ?Y: member(?X, r2).
```

4. $A.r \longleftarrow f_1 \cap .... \cap f_n$

   Each principal that is a member of all roles $f_1, ..., f_n$ is also a member of role $A.r$. The expression $f_1 \cap .... \cap f_n$ is called an *intersection*[1] role.

```
A: member(?X, r) :-
  A: member(?X, f1),
  A: member(?X, f2),
  ....,
  A: member(?X, fn).
```

RT_1 adds parameterized roles to RT_0. The slog code will remain the same other than replacing the constants denoting a role r with variables $?R$.

7.4 Common Access Control Policies

- **Discretionary Access Control (DAC).** In slog, principals can pass their access rights to other principals at their own discretion. Consider the rule:

```
  Bob: read(?User, ?File) :- FileService: read(?User, ?File).
```

  Suppose if `FileService` grants permission to a user, `Alice`, then it automatically follows that `Bob` can also access the same `?File` which is read by `Alice`.

[1]In slog and in trust logics, intersection is same as conjunction.

- **Mandatory Access Control (MAC).** In MAC, any operation by any subject on any object is tested against the set of authorization rules to determine if the operation is allowed. For example, consider a set of users and files such that:

  - read access to the file by the user is allowed only if the level of the user is greater than the level of the file,
  - write access to the file by the user is allowed only if the level of the user is less than the level of the file.

  We assume every user and file is associated with a label from an ordered set of security levels. The constraint domain uses the function `level()` to retrieve these labels, and the relation `<=, >=` represents ordering.

  The assertions below implement the Simple Security Property from the Bell-LaPadula model [BL76, Lan81].

```
read(?User, ?File) :-
  FileService: tag(?User, user),
  FileService: tag(?File, file),
  level(?User) >= level(?File).

write(?User, ?File) :-
  FileService: tag(?User, user),
  FileService: tag(?File, file),
  level(?User) <= level(?File).
```

- **Hierarchical resources.** With slog collection type sequences, hierarchical resources are supported naturally. Consider the rule where Alice grants read access to all resources hosted under path to Bob.

```
// path 'alice.org/user/*' => seq"['alice.org', 'user', '*']"
Alice: read(Bob, path"alice.org/user/*").
```

```
// url'alice.org/user/' => seq"['.', 'org', 'alice', 'user', '*']"
Alice: read(Bob, url"alice.org/user/*").
```

Consider a query with a child operator `<`.

```
read(Bob, ?X) :- Alice: read(Bob, ?Y), ?Y < ?X.


read(Bob, path"alice.org/user/bob")? // OK
read(Bob, path"alice.org/user/bob/home")? // Not OK
```

Consider a query with a descendant operator `<<`.

```
read(Bob, ?X) :- Alice: read(Bob, ?Y), ?Y << ?X.


read(Bob, path"alice.org/user/bob")? // OK
read(Bob, path"alice.org/user/bob/home")? // OK
```

- **IP delegation.** Range queries are common across IP delegation. Consider an assertion from Alice to Bob delegating an IP subnet of 192.168.1.0/24.

```
// ipv4"192.168.1.0/24" => range"[3232235776..3232236031]"
Alice: delegate(Bob, ipv4"192.168.1.0/24").
```

Consider a query with in operator `<:`.

```
authorize(Bob, ?X) :- Alice: delegate(Bob, ?Y), ?Y <: ?X.


authorize(Bob, ipv4"192.168.1.100")? // OK
authorize(Bob, ipv4"192.168.1.300")? // Not OK
```

- **Constrained delegation.**

  In SAFE, there is no obvious way to constrain a delegation at issuing time. Any principal is free to issue a delegation to other principal. However, it is easy to constrain the delegation at query/context building time. For instance, the authorizer can provide the logic by passing the depth of the delegation chain as a constraint during credential discovery phase, i.e., fetch from SafeSets.

8

Architecture

The SAFE project builds on the earlier research in logic-based trust management by focusing on "logical trust as a systems problem". SAFE architecture is modular with four interlocking system components—slog, slogsets, slang, and SafeSets—as described in Chapter 3. To provide an end-to-end implementation using a single language, we chose Scala[1] [sca16], an expressive hybrid functional object-oriented language for the implementation. An advantage of implementing our own inference is that the inference engine is tailored to the needs of trust logics with relatively small footprint[2]. A prototype and overview of SAFE is presented in [CT14b, TC15].

The elements of SAFE project include: (i) a clear separation of logical inference from credential discovery and building proof contexts; (ii) a set of programming tools to create, publish, and import credentials—and policies—for logical trust; (iii) a decoupling from the external representations to transport the logic; (iv) a builtin encoder that translates logic into human readable certificate chains; (v) an integration

[1] A primary reason is that Scala runs on JVM and compatible with Java, which is the language of implementation for target applications.

[2] The size of SAFE inference engine is only ∼1500 whereas the size of the inference engine from comparable implementation (XSB with top-down inference) is ∼60,000 SLOC [XSB15].

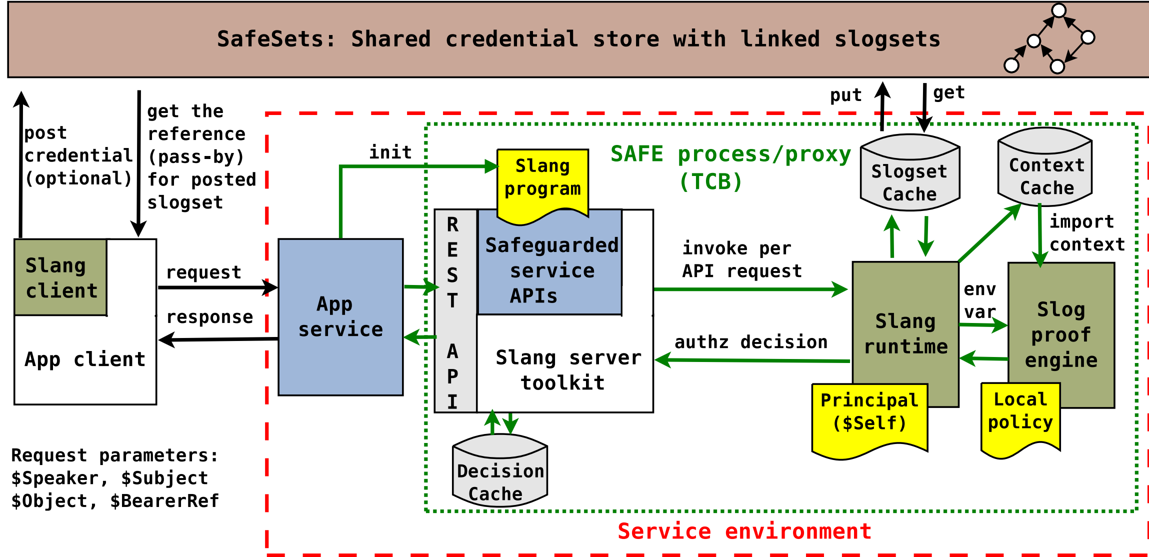


FIGURE 8.1: Server access control using SAFE. The SAFE instance runs as a separate process with a loaded slang program that contains context building procedure, the principal's signing key (`$Self`), and the authorizer's local policies specified in slog. The server application installs slang code in the SAFE process, which registers all the `defguard` APIs for access control checks. Credentials are passed as references to signed logic sets (slogsets) in a shared distributed store (SafeSets). The SAFE process fetches slogsets on demand, validates the signature and speaker, and caches them for use by the slog interpreter.

with application service frameworks; and (vi) a deployment structure that facilitates cross-language interoperability.

8.1 Runtime

SAFE runs as an interpreter with one or more slang programs loaded into it. The slang code is executed in multiple ways: (i) it can be directly invoked from command-line tools (client-mode); (ii) it can be invoked within a co-located SAFE process through a REST API (server-mode); or (iii) or it can be invoked by an application server that relies on SAFE for its security policy implementation (integration-mode).

The behavior of the interpreter program is determined not just by the slang code itself but also by the external logic content passed to it: linked slogsets references.

The slang code contains the local policies of the authorizer, and logic for credential discovery and pruning. The slogsets contain the credentials, external policies, and trust structure. The slogsets may be passed at runtime without changing the slang program. In the integration mode, the participants may even formulate rules and exchange them over the network (via materialized certificates) as the system executes.

The interpreter is stateless, so participants may restart it and/or reload slang programs at any time: it affects only the access control for future requests. Slang programs are composable: it is easy to add code to customize the local behavior. Changing the program leaves other software and application state unchanged.

In general, the interpreter runs within its own process. Applications may call the REST API directly through hooks added to the programming framework. For example, as illustrated in Figure 8.1, consider a server that uses SAFE to apply access control checks for incoming requests. The server instance loads the slang program that contains the context building procedure, the principal signing key (`$Self`), and the authorizer's local policies specified in slog. The server application installs slang code in the SAFE process, which registers all the `defguard` APIs for access control checks. The server checks an incoming request by issuing a logic query (a goal as defined by the guard predicate) against the proof context assembled by the guard script. Upon each request, the server toolkit binds the requested method to a `defguard` method, and passes the standard parameters of the request. These parameters include the subject of the request, and the object for which the request is made, and an optional token referencing the caller's credentials (`$BearerRef`) in SafeSets.

As the slang program executes, the SAFE process may post and fetch slogsets from SafeSets. It may also fetch URLs or access other Internet services as directed by the program, but it presents no service interface to the external network. These choices assure that SAFE is portable and interoperable. They also enhance security

for server applications that use SAFE: the application's signing key-pair (`$Self`) is isolated from its Web-facing front end.

The SAFE runtime handles secure slogset identity generation, object identity generation, cryptographic operations automatically and transparently. In addition, the runtime provides builtin methods for publishing and fetching from Safesets.

8.2 Implementation

SAFE is implemented in Scala language [sca16] including the inference engine written from scratch in about 10000 lines of code [saf16]. Scala is a statically typed functional-object oriented language on the JVM with the support for distributed message-driven actor based concurrency through Akka library [akk15]. SAFE uses Akka for building the REST API and the SafeSets client implementation.

SAFE implementation is modular: slog is a standalone component upon which other layers are built. Slang extends slog parser and the inference engine. Slang incorporates the programming logic for handling certificates, caching slogsets, and communicating with external services (e.g., the application service, the requesting client, and the Safesets). In general, slang can be an independent language in itself with the parser and runtime decoupled from slog.

SafeSets is based on Riak key/value store [ria15]. The object authentication, identifying the ownership, and granting write privileges on a slogset on the SafeSets server is implemented in SAFE itself: each member node of a Riak cluster runs a SAFE proxy shim with a bulitin authorization policy.

On a post operation, the slogset ID serves as the key, and the named certificate is the value. The shim checks access for post operations: it verifies that the value (a certificate containing a slogset) is signed under a public key whose hash yields the slogset ID, when concatenated with the given local name on the slogset. The shim is implemented using SAFE itself: it is a SAFE process with slang code that

invokes ordinary certificate parsing and validation, queries the meta-attributes, and performs the guard check. SafeSets clients access the Riak store only through the shim, which serves the Riak request protocol. This is a simple example of using SAFE to "safeguard" a network service transparently, as an alternative to modifying the service or integrating with a service framework[3].

Because the slang scripting language abstracts the details of fetch/post with builtin constructs and hides them from applications, *"SafeSets is a replaceable component within the SAFE architecture"*. However, the idea of using a shared decentralized certificate store generalizes to other models for storing and authenticating the sets. In principle, slogsets could be stored in secure web directories maintained by the owning principals (e.g., the secure Web with TLS support can be used as a shared storage). In some scenarios, the slogsets can be stored in a centralized database, which is trusted rather than a distributed storage (e.g., in the case of a single service provider domain such as Amazon AWS database).

The SafeSets client is implemented using Akka actors for achieving distributed concurrency and scalability. Figure 8.2 shows the architecture of SafeSets implementation. The slang actor makes a request to the SafeSets client to fetch a reference or post a slogset. The SafeSets client sends to the master actor that distributes work to the workers. Typically, the distribution happens by pushing work to the workers via standard scheduling algorithms such as a balancing dispatcher. However, for SafeSets, we implemented our own dispatching algorithm to avoid back pressure. Back pressure occurs when the workers are heterogeneous or the master cannot distribute the work evenly. To avoid back pressure, the design of master-worker communication is reversed. Instead of master pushing the work, the workers in SafeSets are

[3] An earlier implementation of the SafeSets server used a key value store by implementing the autorization within the storage service itself that required several modifications to the source code. The present SafeSets approach is compatible with any key value store since it delegates the authorization procedure to SAFE.

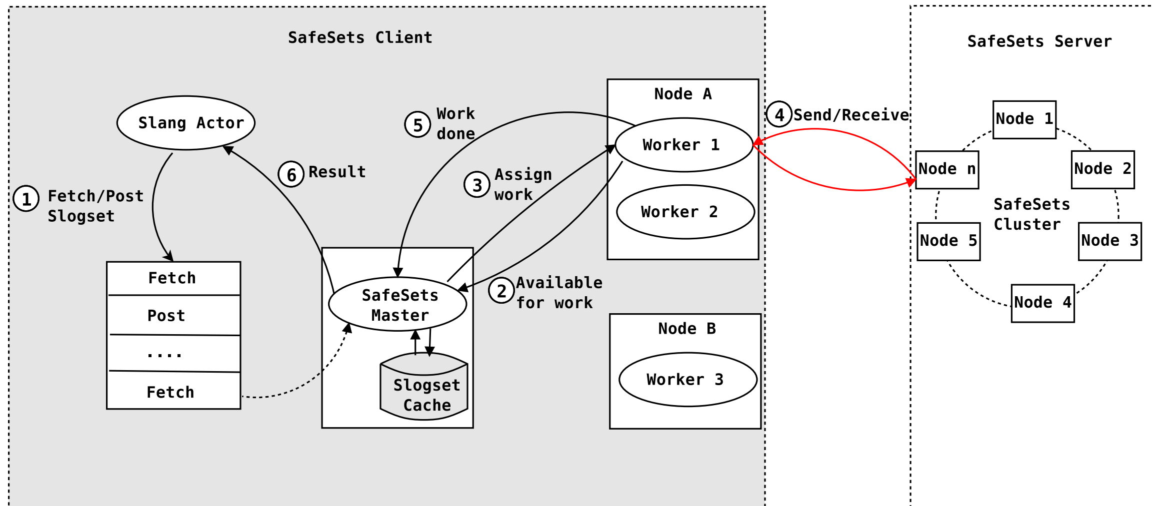


FIGURE 8.2: SafeSets implementation using an actor-based programming model for handling concurrent requests. The SafeSets client via a designated master actor distributes work via a pull based approach, i.e., the workers must request the work rather than the traditional approach where the master pushes the work. The pull-based approach avoids back pressure on the workers. The master is free to assign the work to multiple workers in parallel or re-assign the work in case of a timeout. The worker actors communicate to the SafeSets server nodes via an untrusted channel (over the Internet). The requests on the server are authorized via a configured proxy shim running SAFE inference for write validation. The server nodes form a ring cluster for balancing load.

required to *pull* the work from the master. The master is free to assign the work to multiple workers in parallel or re-assign the work in case of a worker timeout. The workers communicate to the SafeSets server nodes via untrusted channel (over the Internet). The server nodes form a ring cluster and communicate among themselves via gossip protocol. Each server node runs a SAFE proxy shim to authorize writes on the server.

9

SafeSets: A Secure Metadata Service

SafeSets is a secure, shared, distributed, and accountable credential store offering a metadata service. SafeSets stores certified objects signed under the issuer's keypair. SafeSets store two kinds of objects: immutable content objects, for which the local name is derived from the hash of its contents; and mutable objects such as credentials, resource objects, for which the principal owner provides an explicit name. Every object in SafeSets is identified by a globally unique self-certifying identifiers (`IID`s) as described in Chapter 3. In addition, an object can also be queried directly using a slogset identifier, which is the hash of the IID on the set. Set identifiers are serve as pass-by-reference tokens where privacy is a concern. The set identifiers are cryptographic unforgeable: it is computationally infeasible to obtain the local name of the set given the identifier unless the identifier was minted using the key.

SafeSets can be visualized as collaboratively editable DAG: each node in the DAG is controlled by its owning principals, and changes to a set by its owners are visible in other sets that link to it. The writes on mutable SafeSets objects are *idempotent*, i.e., the same write issued multiple times results in the same content including metadata, i.e., the logical statements are always merged to form a set union. Idempotence

implies every write to SafeSets is a read-modify-publish to a set. Although the writes incur additional cost of read, in practice, we found the idempotence keeps SafeSets's API simple and caching sets efficient.

9.1 Design

SafeSets requires high availability but the service itself needed not be trusted. SafeSets is designed as an accountable storage following the design principle: *trust but verify* [YC04]. We would desire SAFE not only detect server violations, but also prove to the client about the server misbehavior. With these considerations, we identify three desirable properties for SafeSets to satisfy accountability: integrity, serializability, and freshness.

Integrity is trivially satisfied since each object stored in SafeSets is cryptographically signed by its owning principal; any client can verify the authenticity of the object using the `IID`. However, without serializability on writes, and freshness on reads, the SafeSets server can arbitrarily reorder writes or provide stale data to clients compromising trust. For example, if a principal updates a credential and revokes a privilege, a malicious SafeSets server can conceal the revocation status by presenting the stale version of the credential. Here, the integrity is still satisfied but freshness is compromised. We define serializability and freshness informally.

Serializability. Serializability implies that there is a total order on the writes to the same set—that is each principal when committing an update is aware of the latest committed update to the same set.

Freshness. Freshness implies that read always return the data from the latest committed write. Note that we cannot guarantee that each set retrieved was the most recently received by the SafeSets—for example, on two parallel writes to the set, SafeSets can reorder the writes due to network delays of such requests. Instead, freshness guarantee that the last committed write (for which SafeSets acknowledged

receipt to the client) will be visible during read.

9.2 Implementation

We achieve these properties using server side *attestation*, i.e., reads and writes are attested by a SafeSets server node using its own key. The attestation procedure relies on "cryptographic hash chains and signing the client requests" to ensure non-repudiation. The attestation APIs involve fetch and post.

Fetch. Client sends a fetch request with a random nonce to a SafeSets server. Server attests the content by including a hash computed over the set, the nonce, and the version hash chain and sends it back to the client. Client verifies the attestation and server signature to accept the fetch as valid.

Post. Client sends the signed set for post. Server verifies the client signature on the set (post authorization) and accepts the set by incrementing the version and returns the attested version hash chain containing the new version. The client accepts the read as complete once the server attestation is verified.

If the server misbehaves, a periodic audit can detect and prove the violation, which may result in blacklisting the server. In SAFE, for simplicity, we configured the SafeSets server nodes as sub-principals to a single master; hence every attested object resides in a single master principal's namespace. To join a SafeSets cluster, every node should present a valid explicit membership credential from the master.

SafeSets together with slang offers an integrated solution for sharing authenticated logic sets in a networked system. Each authorizer's local SAFE runtime interacts with the SafeSets service to support the set abstractions of slang by fetching referenced sets on demand, caching their logic content, and assuring the freshness and validity of logic content passed to the proof engine. The SAFE runtime provides builtin methods for publishing and fetching from Safesets.

For example, when a program defines a slogset (using `defcon`), the builtin encoder

consumes the meta-facts and encodes the information they contain into the selected certificate format. When SAFE fetches a certificate, the builtin decoder validates the certificate, extracts the contents, and materializes it as an in-memory slogset. The slogset represents the relevant meta-information from the certificate as logic meta-facts. These facts are available to the slog inference engine if the slogset is added to the proof context for a query.

Note that SafeSets is designed to be used as a cooperative shared storage. However, if confidentiality is desired, the clients can use encryption on sets to conceal the credentials.

9.3 Safe Expressions (SafeX)

SAFE supports compact, reliable encoding of slogsets in X.509 certificates (using a string encoding within an attribute field) and also in a native SafeX format. The crypto layer represents all semantic content of any signed certificate internally in common logic, including builtin predicates for meta-attributes such as expiration time, encoding type, and so on. It generates the encoded cert from a logic set containing the required meta-attributes as facts, which are easy to specify directly in slang set constructors. The native SafeX certificate format is not subject to the arbitrary length constraints of X.509 certificates, and also improves compactness by hashing public keys embedded as principal names in the logic.

Consider an example policy from SafeGENI (see § 11.5), where the issuer (`Geni`) endorses a subject as a project authority as illustrated in Code Snippet 9.1. The corresponding slog code from line 6-8 contains three statements: one credential endorsing the subject as a project authority (line 7); one policy stating that the issuer accepts any subject as project authority if another principal with attribute `geniRoot` endorses the subject as project authority (line 6); and a link to another certificate/slogset within issuer's namespace with local name as `'policy/standardEndorsePolicy'`

```
1  defenv Selfie :- principal('geni_keypair.pem').
2  defenv PA :- u'qgU47-sQzOuOVZfDAJMg-OyRQBiRw6vHU8UT_pTVNFM'.
3
4  defcon endorse(?Subject) :-
5    "endorse/$Subject"{
6      endorse(?PA, projectAuthority) :- endorse(?Geni, geniRoot),
        ↪  ?Geni: endorse(?PA, projectAuthority).
7      endorse($Subject, projectAuthority).
8      mkLink('policy/standardEndorsePolicy').
9    }
10 end
11
12 definit post(endorse($PA)).
```

Code Snippet 9.1: The slang code illustrates an example policy from SafeGENI (see § 11.5), where the issuer `Geni` endorses a subject as a project authority. Upon post, the slogset is materialized as a certificate, which is shown in Code Snippet 9.2.

(line 8). Upon post (line 12), the slogset is materialized as a certificate. The Code Snippet 9.2 illustrates the corresponding encoded certificate with native SafeX encoding.

SafeSets objects contain metadata, which is useful for verifying the integrity, and ownership privileges on a slogset/certificate. This metadata include predicates such as:

- `Issuer`: The issuer is the principal that signs the slogset. Note that the issuer may sign the certificate on behalf of a speaker for which the speaker grants a `speaksFor` privilege. The predicate is of arity three: `issuer(issuer-ID, issuer-PK, issuer-SID)`, where `issuer-ID` is the ID for the issuer, `issuer-PK` is the full public key of the issuer, and `issuer-SID` is the reference to a slogset if the speaker and issuer are different. The `issuer-PK` and `issuer-SID` are optional fields.

- `Status`: The status of the certificate, i.e, whether valid or revoked. The status predicate is of arity four: `status(ok-or-revoked, published-date,`

```
1  cert #'application/slang;charset=utf8;hash=sha256' ->
    ↪  u'rP1oOBltJM1fBOKstppW4-PM4WnOPPn5tOLxfehOufU'(
2    status('ok', '2016-01-2611T00:50:00.000-04:00', 0, nil),
3    signedData(
4      version('1'),
5      issuer(u'RTQYM8q3J0-0J95sV_yoyG_shw41T87zPDEkvVIMDFw', nil ,
        ↪  nil),
6      subject(u'RTQYM8q3J0-0J95sV_yoyG_shw41T87zPDEkvVIMDFw', nil),
7      validity('2014-05-11T00:50:00.000-04:00',
        ↪  '2017-05-11T01:00:00.000-04:00', 'PT24H'),
8      cred #'application/slog;charset=utf8;hash=md5' ->
        ↪  'endorse/q-O1jJKpE7ZMVyf-3nKhYgEOjGvE16UJtA6h45DxgKE'{
9        link(u'qgU47-sQz0uOVZfDAJMg-OyRQBiRw6vHU8UT_pTVNFM').
10       endorse(?PA, projectAuthority) :- endorse(?Geni, geniRoot),
          ↪  ?Geni: endorse(?PA, projectAuthority).
11       endorse(u'q-O1jJKpE7ZMVyf-3nKhYgEOjGvE16UJtA6h45DxgKE',
          ↪  projectAuthority).
12     },
13     signatureAlgorithm('SHA256withRSA')
14   ),
15   signature(u'nIEwUdpOE82VTaoTFrHaRYMKkPexU3Tp9n5ik2Kksh2iX
      ↪  3sODa8_zki2fwrGqDqVEwwkHzlL1Xor3gE2DmoF63RKTOpau25gF-f
      ↪  M3xiaFJQg6UeMOAm3KuzZ4xH4VzX3qrEDdrZzmNH6RvlXwVgXkoOhD
      ↪  mGLoPIu__-EpKsE7ehsDjIjK-jLujbd-NU_2IpOoc20eQr6qFelYZ4
      ↪  Sr3st9PDerm4Mda6yC_kQ2RMG2_hUEndprKh6thnCYV6sfqnxwsSzt
      ↪  wtCr5TyQYsO57fvZBPKERfoYavXJPOm7XAoEVtVv-5tS7ztZQOmzwk
      ↪  MhA4EfyYW3oqIdJNtv7OUiOOOaw')
16 ).
```

Code Snippet 9.2: An example of certificate objects stored in SafeSets corresponding to the slang code from Code Snippet 9.1. Line 9-11 correspond to the slog statements lines 6-8 from Code Snippet 9.1. The other predicates are meta predicates that are filled with defaults. The certificate is encoded in native SafeX format, which makes it readable and navigable (by HMTL rendering the `link` predicates) to understand distributed trust structure.

`object-version, current-SID)`. `published-date` indicates the published time of the certificate that is provided by the SafeSets server node; `object-version` is an integer that is incremented by each subsequent post to the same slogset; and `current-SID` is a forward reference to an active certificate if this certificate is already revoked. The `current-SID` field is optional.

Object version is used for retrieving any previously revoked or poisoned certificates. The `IID`s of earlier version certificates are obtained by combined hashing of the current `IID` with "0" recursively until the required version is reached.

For example, consider an update to the slog statements from Code Snippet 9.2. Since an update does not change the local name and consequently the slogset ID, the update is in-place with a change of version and update of status to the old certificate. For example the status of the version 0 certificate is changed as follows:

```
1 cert #'application/slang;charset=utf8;hash=sha256' ->
  ↪ u'Aip7XiRxDGXN-0HOmAxak_quSlMG0gqW4CibsEKo1-c'(
2   status('revoked', '2016-01-2711T00:50:00.000-04:00', 0,
    ↪ u'rP1oOBltJM1fBOKstppW4-PM4WnOPPn5t0LxfehOufU'),
3   ....
```

where `u'Aip7XiRxDGXN-0HOmAxak_quSlMG0gqW4CibsEKo1-c'` is obtained by concatenating "0" to `u'rP1oOBltJM1fBOKstppW4-PM4WnOPPn5t0LxfehOufU'` and taking a cryptographic hash. The `current-SID` reflect the present active certificate ID.

The new certificate changes the `object-version` from 0 to 1, but retains the original slogset ID, i.e., `u'rP1oOBltJM1fBOKstppW4-PM4WnOPPn5t0LxfehOufU'`, for a direct fetch.

```
1 cert #'application/slang;charset=utf8;hash=sha256' ->
  ↪ u'rP1oOBltJM1fBOKstppW4-PM4WnOPPn5t0LxfehOufU'(
2   status('ok', '2016-01-2711T00:50:00.000-04:00', 1, nil),
3   ....
```

In this way, the `status` with `object-version` helps the client/authorizer to traverse the lineage of the SafeSets objects and identify the present active

version of the certificate. We revisit how Safesets solves the issues with the certificate management in § 10.2.

- `Subject`: The subject is the principal for which the credential is issued. The predicate is of arity two: `subject(subject-ID, subject-PK-opt)`, where `subject-ID` is the ID of the subject, `subject-PK` is the full public key of the subject. `subject-PK` is an optional field.

- `Signature`: The cryptographic signature to verify the speaker and the IID on the object. The predicate is of arity one: `signature(base64-encoded-signature)`.

- `Signature Algorithm`: The signature algorithm used in signing the certificate. The predicate is of arity one: `signatureAlgorithm(signature-algorithm-name)`.

- `Validity`: The validity range of the certificate including the issuer controlled refresh time after which the certificate needs to be re-fetched from SafeSets. The predicate is of arity three: `validity(not-before-date, not-after-date, refresh-time-period)`.

- `Version`: The encoding version specified by the issuer indicating the version used by the encoders/decoders to materialize a slogset into a certificate and vice versa. The predicate is of arity one: `version(version-number)`.

Apart from the status meta-predicate that is provided by the SafeSets server, the rest of the content is provided by the SafeSets client and signed by its issuer.

10

SAFE Service Integration

SAFE integrates with the application service frameworks by providing a RESTful service API for each of the `defguard` rules. Many server applications are built using general-purpose service frameworks, which handle the details of network communication, URL parsing, request dispatch, and multithreading. The application itself consists of an implementation of the service API: it is a set of methods that plug into the server framework. SAFE provides a similar server framework where the applications can directly integrate their service API.

SAFE extends the `defguard` rules in slang for the SAFE service API. In the server mode, these rules define external entry points to the slang program for access checking the incoming requests. The server toolkit is built upon Akka [akk15] actors as discussed in Chapter 8. We added SAFE "glue" code to the framework to collect the parameters of the request as key-value pairs. The SAFE service matches each request with a corresponding guard entry method in the slang code and supplies the bindings of arguments on that method with the user provided values. The standard parameters of the request includes the subject of the request, the object for which the request is made, the speaker of the request if the subject is speaking on the

behalf of someone else, and the bearer reference that the requester may carry. The parameters are passed into the slang program as named environment variables.

Upon a request for access, the slang program with the `defguard` rule corresponding to the request method is initiated. The slang code runs by fetching the certificates starting with the requester provided bearer reference, and then follow the chains of linked certificates until a closure is reached. Each fetched certificate is verified for its integrity by validating the signature, which are then converted into an authenticated slogset. The slogsets sharing the name/identifiers are merged into a single slogset. In addition, for each request, a proof context is built as the union of set of authenticated slogsets, which are either fetched following the bearer reference or found locally in the cache. These proof contexts are then supplied to the slog inference engine, which checks for a valid proof. On successful execution, the request is routed to the application servers. Otherwise, the request is rejected by SAFE itself without reaching the application servers.

Application server frameworks can use SAFE as a proxy or invoke SAFE via REST API to check access control for client operations on the objects they serve (see Figure 8.1). In order to convert a service API into a SAFE service API, each API method needs to register a guard rule in the slang program through `defguard` prefix. When a request enters the Web application or service framework, it invokes SAFE to evaluate a declared guard whose name matches the requested method, passing a list of variables named in the request. Code Snippet 10.1 illustrates a sample interaction between client, the application server, and SAFE for the policy listed in Code Snippet 4.1. Ideally there is no change to the application itself, other than defining slang guards for each method.

The conventions for web service integration build on the secure naming conventions for slang and SafeSets outlined above. Several environment variables with predefined names may be present when the guard is invoked, for example, `$Object`,

```
curl -XPOST
  --header 'Accept: application/safe'
  --cert-type pem
  --cert /usr/home/client-cert.pem
  --data '?ObjectRef=Sq3Q1eBAzv-YddnfOOPShr5HYEwV_yOoFS'
  https://alice.org/secret/sensitive.pdf

curl -XPOST
  --header 'Accept: application/safe'
  --data '?ObjectRef=Sq3Q1eBAzv-YddnfOOPShr5HYEwV_yOoFS'
  --data '?Subject=yLXRbMlQsmdqxmcVyu5bJsJR7JzpcpHftpXd'
  --data '?Speaker=yLXRbMlQsmdqxmcVyu5bJsJR7JzpcpHftpXd'
  https://safeclouds.org/read?'?Object=/secret/sensitive.pdf'
```

Code Snippet 10.1: Example client-server-SAFE interaction for the policy listed in Code Snippet 4.1. A client requests read access for a file `https://alice.org/secret/sensitive.pdf`. The application server of `https://alice.org/` authenticates the subject (extracts the subject and speaker from the client certificate, line 1-6) and routes the request to an internal SAFE service proxy `http://safeclouds.org/`. The SAFE proxy fetches the credentials using bearer reference and then cross check whether the subject has access to object (line 8-13).

`$ObjectRef` in Code Snippet 10.1. The guard may reference these special environment variables internally without explicitly passing them to the `defguard` API method arguments. These implicit variables are local to a request whereas the environment variables defined through slang trust scripts are common across requests. SAFE executes each request on a separate thread with its own bind of the variables.

In the rest of the chapter, we describe how to caching works in SAFE (§ 10.1) and how certificates are managed using the decentralized storage. We also discuss some challenges and open issues with certificate management (§ 10.3).

10.1 Caching Credentials

Caching is an optimization technique employed by SAFE to reduce the latency of each user request. SAFE fetches slogsets automatically on first reference to a token

or a slogset ID in the slang program. The client side code performs a cycle-safe recursive fetch and the requests are executed in parallel using a thread pool for reduced latency. After each subset is fetched, SAFE validates the signature, parses the certificate, authenticates the issuer of the slogset, and assigns the speaker for each statement in the slogset. If the certificate is valid, its contents are extracted into an in-memory slogset including meta-attributes. Slogsets created via a slang program or fetched directly from SafeSets are cached in memory using a *slogset cache*.

The raw certificates may be optionally cached on a local disk using a *cert cache*. A fetch on a token always checks through the slogset cache. For instance, each subsequent fetch on a token encountered during the traversal of a closure of a credential graph will check the local cache, and also fetch the certificate if it is not present or expired locally. If a certificate is not present locally, it is automatically fetched on the first use of the token, then converted into a slogset, and cached in the slogset cache.

For each request, SAFE assembles, renders, and dispatches a proof context to the inference engine. The rendering a context involves traversing the credential graph of in-memory slogsets and flattening all the statements into a single context required to substantitate the proof. Rendering statements may also involving building additional indexes (for example, secondary indexes where necessary), which may help to speed up the inference engine. The proof context is also cached under a *context cache*.

If a slogset or a proof context becomes popular, we anticipate those entries will be cached and found locally. To ensure the frequent items are cached, we implement the cache eviction mechanism using LRU algorithm. In particular, we rely on Guava caching library[1] that provides thread-safe implementation of LRU cache.

Cached copies may become out-of-date, and so the authorizer may assign them optionally time-to-live (TTL) values. In general, the default TTL value is provided

[1]https://github.com/google/guava

by the issuer that sets the refresh time under the `validity` metapredicate (see § 9.3). In addition, the expiration of entries is guided by the type of entires residing at the caching layer. For instance, the expiry date on the context cache is set to the *lowest* expiry dates from the collections of sub-contexts that are assembled. SAFE tracks the period of validity internally to expunge any expired content from the caches. A leaf subset expires from the slogset cache at the expiration time of its containing certificate. We also added support to invalidate a cached context at the earliest expiration time of any statement it contains, so the proof engine sees only fresh logic content. If an expired certificate is re-issued, then a subsequent fetch on that certificate token pulls the fresh certificate from the store automatically.

We also enhanced server integration by adding support for a query *decision cache* on the Web server side. The decision cache optimizes repeated operations for a given subject and an object by avoiding the inference check entirely within a configurable time limit.

10.2 Managing Certificates

SAFE together with SafeSets solves the long standing issues with certificate management. In contrast to the approaches such as CRL and OCSP, our approach does not need an authorizer to maintain huge list of revocation certificates or require other participants to run services to respond to status validation checks.

- **Renewal.** An issuer may renew a certificate by posting the new expiration time on a certificate to SafeSets with the same identifier or a token overwriting the existing certificate. Any subsequent fetch from the authorizer will see a renewed certificate. Alternatively, if a SAFE authorizer encounters an expired slogset, it uses the identifier to fetch a new version automatically and retries the query.

- **Revocation.** An issuer may modify a posted slogset at any time or "poison" the certificate to invalidate its contents. A modification or poison does not take effect if an authorizer uses an old copy of the slogset from its cache. An issuer may control the expiration times to bound the time that a set remains in an authorizer's cache by setting the `refresh` time on the published set as described in § 9.3. An authorizer may refresh slogsets in its cache at its discretion, even if they have not expired.

- **Rotation.** Rotation is a challenging issue with PKI. Keys may be lost or misplaced or may become outdated over time. In SAFE, to enable a third-party to verify credentials, we issue certificates to SafeSets by identifying the principals by their public key hashes. Although this approach has benefits in terms of global naming and identification, it presents challenges for fetching the credentials transparently upon a key rotation. If a principal loses or rotates its key-pair, then certificates that incorporate the stale key in their slogset identifiers must be regenerated. A fundamental issue is not only re-issuing all the credentials that correspond to the principal for which the new key is minted, but also updating the uplinks which may potential link to one of the slogsets owned by this principal.

  In SAFE, we avoid regenerating the upstream credentials involving a third-party by allowing the new principal to poison the old certificates with an optional link. The link references the new certificate generated using the new key-pair. In this way, a fetch on a certificate automatically finds the new certificate and retrieves the credentials.

  Alternatively, SAFE advocates each principal to create sub-principals which `speaksFor` the master principal on a given role (e.g., signing, encryption). We assume that the master principal is kept securely offline. The slogset ID genera-

tion is performed using the master principal hash rather than the sub-principal. In this way, we can avoid potentially expensive certificate regeneration involving slogset references of the rotating principal.

10.3 Discussion

The SAFE approach to managing credentials also raises some potential concerns.

Malicious content. Issuers may write malformed certificates to SafeSets or generate a malformed credential DAG, e.g., by creating cycles in the DAG. The SAFE fetch procedure rejects malformed certificates and detects cycles. Valid certificates contain only slog statements, which share the termination properties of pure datalog: all queries terminate. However, issuers may create very large or costly slogsets to mount a denial of service attack. An authorizer may bound the size of incoming logic sets and query contexts at its discretion.

Accountability. Policy mobility relies on participants to enforce the policy rules of others. In general, entities control their own authorization decisions and have power to do harm only to the extent that others trust them. For example, a GENI aggregate that ignores policy conditions may be unduly promiscuous with its own resources, but it cannot affect access to the resources of others. Moreover, all entities are strongly accountable (in the sense of CATS [YC07]) for certificates they post to SafeSets representing the result of access decisions. Accountability is an active research topic.

Confidentiality. Synthesized identifiers raise the issue of confidentiality of policy rules and other logic material stored in SafeSets. If an entity wants to protect a confidential slogset it may salt the name: it is infeasible to guess a hashed identifier that is effectively random. We emphasize that the protection for writing to SafeSets is stronger: a client must possess a principal's private key in order to write to a slogset that the principal controls.

Reclamation. Logic material may accumulate in SafeSets over time. SafeSets may delete any set after it has expired: all logic sets have expiration times. Even so, issuers may use unreasonable expiration times or simply post useless data to the store. SafeSets authenticates each issuer by its public key, but quotas are of no help if an issuer can mint new keys at will. One option is to apply a SAFE access check for posting. Another option is to arrange the store so that each issuer provides and manages its own storage (e.g., via a Web server).

SafeSets failure. Managing SafeSets as a decentralized key-value store can be a scalable and reliable solution. One or more entities may control the SafeSets servers. A faulty or malicious server can destroy content or block access to it. However, it cannot subvert the integrity of the system because all slogsets are signed by their issuers.

11

Applications & Evaluation

In this chapter, we present the applications of SAFE based on case studies drawn from practice (i) a secure name service similar to DNS that resolves names across multi-domain federated systems; (ii) a secure proxy shim to support rich access control in a key-value store; (iii) an authorization module for a networked infrastructure-as-a-service system with a federated trust structure (NSF GENI architecture); and (iv) authorization rules for a secure cooperative data analytics service that enables computation on sensitive data in compliance with secrecy constraints. We provide overview of each application and present empirical evaluation.

The evaluation methodology seeks to answer the following questions.

Q1. Does SAFE achieve acceptable performance? How SAFE compares to ACLs and capability-based access control where policies are attached to entities directly without involving credential retrieval? What is the overhead of invoking slog through slang?

Q2. Are trust logics practical for use and deploy in real applications? What is the programming effort required to build secure applications using SAFE as the

foundation for trust management and access control?

Q3. How does linking slogsets and caching contexts improve performance across space (cross-sharing of common slogsets among multiple simultaneous queries) and time (caching frequently accessed slogsets)?

Q4. What is the overhead of server-side access checks and attestation in SafeSets?

Q5. What is the overhead of using PKI for authentication?

All our authorizer experiments are conducted on an eight core Intel Xeon CPU E5520 @ 2.27GHz processor containing 8 GB of RAM and running CentOS 6.7 with hyper-threading enabled. The SafeSets cluster consists of four VMs, each with a single core Intel Xeon CPU E5620 @ 2.40GHz processor containing 1GB of RAM interconnected by a 1Gb network, all running Ubuntu 14.04. The authorizer accesses the SafeSets store over WAN. We use unmodified Riak 2.0 [ria15] as our key-value store for SafeSets guarded by SAFE as a proxy "shim" for authorizing writes to a slogset.

11.1 Microbenchmarks

We use microbenchmarks to measure the common operations of SAFE. Table 11.1 shows the overhead of basic operations in SAFE on a 1kB of payload per certificate. The slogset identifiers are configured to use SHA-256 for hashing, and Base64 for encoding and decoding, which resulted in tokens of fixed size 44 byte strings. For signing, we use 2048-bit RSA keys. The *null* inference is the minimum penalty measured in terms of latency when invoking the slog inference engine through slang. In general, the inference execution time is proportional to the size of the proof context and the number of global variables which are substituted from slang to slog.

Fetch a certificate in the Table 11.2 refers to a single certificate fetch—i.e., a certificate is retrieved from SafeSets without traversing any links—followed by signature verification and conversion to a slogset. The latency of a fetching a set includes: (i) retrieving a certificate; (ii) parsing the certificate; (iii) retrieving the public key of the issuer from the issuer reference on the certificate; (iv) validating the signature on the certificate; and (iv) converting the certificate to a slogset data structure.

Post a slogset in the Table 11.2 refers to posting a slogset to SafeSets by signing its contents. The latency of posting a slogset includes: (i) signing the slogset; (ii) sending the request to a SafeSets server over network; (iii) verifying the signature on the certificate by the SafeSets server; and (iv) issuing local post by the SafeSets server to its key-value store.

Attested fetch and *attested post* in the Table 11.2 refers to server-side attestation on the SafeSets cluster. For attestation, each request from the client involves sending a nonce in addition to the standard payload. The server in turn signs the response using its own key. The server-side attestation ensures accountability as discussed in Chapter 9.

Table 11.1 shows overhead associated with parsing a certificate is in the order of 2ms per 1kB payload. When we increased the payload to 5kB, i.e., each certificate contains more logic content (number of logic statements were increased from two to 600) while keeping the metadata constant, the parsing cost is increased to 40ms. The parsing relies on packrat parsing and parser combinators, the builtin parsers in Scala. The difference in efficiency between parser combinators and custom parsers is shown to be over 100x, which we believe is achievable in SAFE as well.

Note that the latency of post is 4X times the latency of fetch: post is expensive since each post is idempotent and performs read-modify-update to a slogset.

To analyze the cost of inference, we simulated delegation chains where the number of unifications exactly matches the proof length. We measure the latency by

Table 11.1: Microbenchmarks of basic operations in SAFE on a 1kB of payload per certificate. Certificates are signed using 2048-bit RSA keys in the native SafeX format. Hash function is configured as SHA-256. All cryptographic strings are Base64 encoded. The *null* inference is the minimum latency penalty for invoking slog through slang.

function	compute hash	verify signature	sign a slogset	parse a certificate	null inference
latency (ms)	0.13	0.56	12.14	2.2	0.09

Table 11.2: Microbenchmarks of SafeSets operations on a 1kB of payload per certificate. Fetch and post measurements involve network latencies over WAN for reading a certificate and writing a slogset from/to SafeSets. Certificates are signed using 2048-bit RSA keys in the native SafeX format. Hash function is configured as SHA-256. All cryptographic strings are Base64 encoded. The attested fetch and post refers to measurements where the SafeSets server signs each response to the client in addition to standard process.

function	fetch a certificate	post a slogset	attested fetch	attested post
latency (ms)	7.8	27.4	22.6	40.3

varying the length of number of unifications matching the goal from 1 to 1024 and the controlling the degree of backtracking, b_d. When b_d is set to zero, we have no backtracking and the length of the proof chain is linear in terms of the unified goals in the input query. Setting b_d to zero approximates proof-carrying authorization (PCA [BSF02]) where the length of the input is exactly the length of the proof chain. In our applications of SAFE, we observed that backtracking scenarios may occur with attribute-based-delegation, where the principal on a goal is variable and the proof context have multiple rule heads matching on the same goal. However, SAFE uses indexing on multiple parameters which often reduces the b_d to a small value. Index optimization is a work in progress. In our next experiment, we set b_d to 10, i.e., each goal can have at most 10 possible rule heads matching with the goal and the length of the proof chain may grow exponentially with the input size. Figure 11.1 shows the latency measurements of when varying proof chain length and b_d. When b_d is zero, latency grows linearly with the proof length, which is expected. However, when b_d is 10, it is interesting to note the latency remains linear upto 600

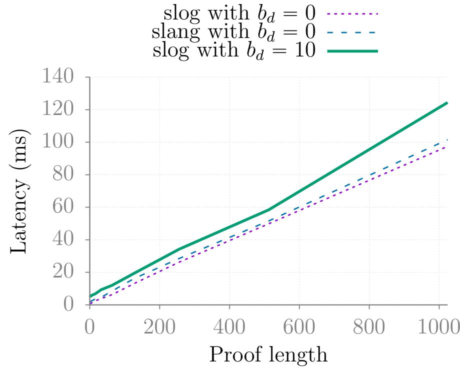


FIGURE 11.1: Cost of inference with varying proof length and degree of backtracking b_d. The latency measurements show that the inference cost scales linearly if b_d is kept low. The plot also shows the overhead of calling slog from slang program is minimal ($< 5\%$).

Table 11.3: Analysis of programming effort for building declarative trust applications in SAFE.

	# Rules	# SLOC	Delegation Pattern
SafeSets	2	15	Linear
SafeNS	7	40	Hierarchical/DAG
SafeGENI	30	110	DAG
SafeCoda	35	150	DAG

unifications. Thereafter, as the number of unifications matching the goal increase, the latency deviates from a linear scale. The result shows that set linking and tailoring proof contexts is important to keep b_d low, which will in turn help to scale the inference cost linearly with the size of the input.

The latency costs in Figure 11.1 show that SAFE inference takes 0.1 ms per unification, which is competitive with respect to identity based ACLs, which typically needs only one fact checked. Further, the plot shows that comparison of inference costs when input is fed directly to a slog interpreter vs. the same input provided via a slang program. Recall that slang program will in turn invoke slog interpreter after substituting all the environment variables. The plot shows that overhead of calling slog from slang is minimal ($< 5\%$).

We built authorization systems for four practical applications in SAFE. Table 11.3 shows the modest effort required to build applications using SAFE. The next chapters describe these applications in detail.

11.2 SafeSets: A Secure Proxy Shim for a Key Value Store

SafeSets provides a secure metadata service. The certificates stored in SafeSets are meant to be discoverable. Hence, any party with a token reference or a slogset ID can read the certificate and verify its integrity. But who can write to SafeSets? How to ensure that the certificate posted to SafeSets is not tampered before reaching the SafeSets server?

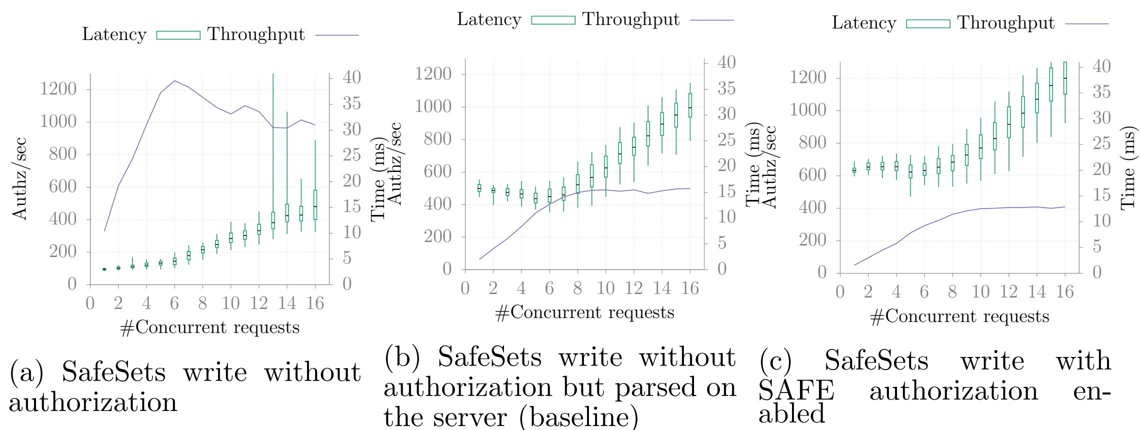
SafeSets uses SAFE as a proxy "shim" for guarding write access to slogsets. The post authorization for SafeSets involves validating the issuer that signed the slogset, who will assume the ownership responsibility for that slogset. The issuer/owner controls the life cycle of a certificate by setting the expiration date accordingly. In addition, the issuer/owner may revoke a certificate early or reissue an updated certificate with the same token or slogset ID. SafeSets is also a good example to illustrate the application of `speaksFor` delegation that is implemented at slang layer.

The post authorization logic is as follows: (i) any issuer can write a slogset to their namespace, i.e., any principal can publish a self-signed certificate; (ii) a publisher can write to any slogset in subject's namespace if the issuer issues a `speaksFor` delegation; or (iii) a publisher can write to a specific slogset if the issuer issues a `speaksForOn` delegation on that slogset name.

Code Snippet 11.1 shows SafeSets post authorization is written in slang. The SafeSets server node receives the POST request with a signed certificate from the client. The server invokes the guard `safesetsPost()` by passing the slogset identifier and the certificate as a string. The `parseSet()` parses the certificate into a AST and provides the reference. The `getIssuer, getSubject, getIssuerKey,`

```
defguard safesetsPost(?SetId, ?PostSetAsString) :-
  spec('validate the principal and verify the integrity of a slogset
   ↪  before posting to a key-value store'),
  ?PostSet     := parseSet(?PostSetAsString),
  ?Issuer      := getIssuer(?PostSet),
  ?Subject     := getSubject(?PostSet),
  ?IssuerKey   := getIssuerKey(?PostSet),
  ?SetName     := getName(?PostSet),
  ?SpeaksForRef      := fetch(getIssuerRef(?PostSet)),
  ?IsSignatureValid := verifySignature(?PostSet, ?IssuerKey),
  {
    import!($SpeaksForRef).
    authorize() :- $Subject: speaksFor($Issuer, $Subject).
    authorize() :- $Subject: speaksForOn($Issuer, $Subject,
     ↪  $SetName).
    $IsSignatureValid, authorize()?
  },
  localPost(?SetId, ?PostSetAsString)
end
```

Code Snippet 11.1: SafeSets proxy shim using SAFE.



(a) SafeSets write without authorization

(b) SafeSets write without authorization but parsed on the server (baseline)

(c) SafeSets write with SAFE authorization enabled

FIGURE 11.2: Performance comparison of issuing a *write* to SafeSets store (write is a post with slogset signing excluded).

`getName` extracts the metadata attributes from the certificate. A fetch (line 8) retrieves the closure of the issuer reference slogset, which provides the proof context for the post evaluation query. The `verifySignature` verifies the signature on the

certificate. The slog query checks whether the issuer has received `speaksFor*` delegation from the subject. Note that by default, slang provides a builtin rule that every issuer `speaksFor` themselves. This rule ensures that all self-signed certificates are valid. For slogsets involving a different issuer and publisher, lines 12-13 ensure that the issuer delegated a `speaksFor*` for the write to be valid. Only if the query (line 14) is valid, then `localPost` (line 16) publishes the certificate into the SafeSets repository.

Figures 11.2a to 11.2c show the performance comparison of SafeSets writes with and without authorization checks respectively. The measurements show the peak throughput drops 21% due to write access check and median latency increases by 33% per post operation.

11.3 SafeNS: A Secure Name Service

Secure name resolution is important component of distributed and federated systems. For instance, domain name service (DNS) is a hierarchical decentralized naming system for services to convert human readable names to IP addresses. SPKI/SDSI proposed a naming architecture based on local namespaces and linking to resolve names securely. We implemented a similar secure name service, SafeNS, in SAFE. SafeNS resolves a multi-component name that may cross the domain boundaries. Using SafeNS, we emulate the SPKI/SDSI name resolution using a DNSSec and SD3 example in SAFE. Figure 11.3 illustrative the end-to-end workflow of credential discovery, slogset linking, context building and pruning using SafeNS as an example.

Given a name service request by the client, the browser/client-agent augments the request with a bootstrapped reference to the root (`ICAANN-ID`) slogset and passes it to the SafeNS resolver. The resolver uses the bearer reference to initiate the credential discovery process using a slang library function `fetchSRN()`. The resolver fetches the root set referenced by `ICAANN-ID` and matches the common name for

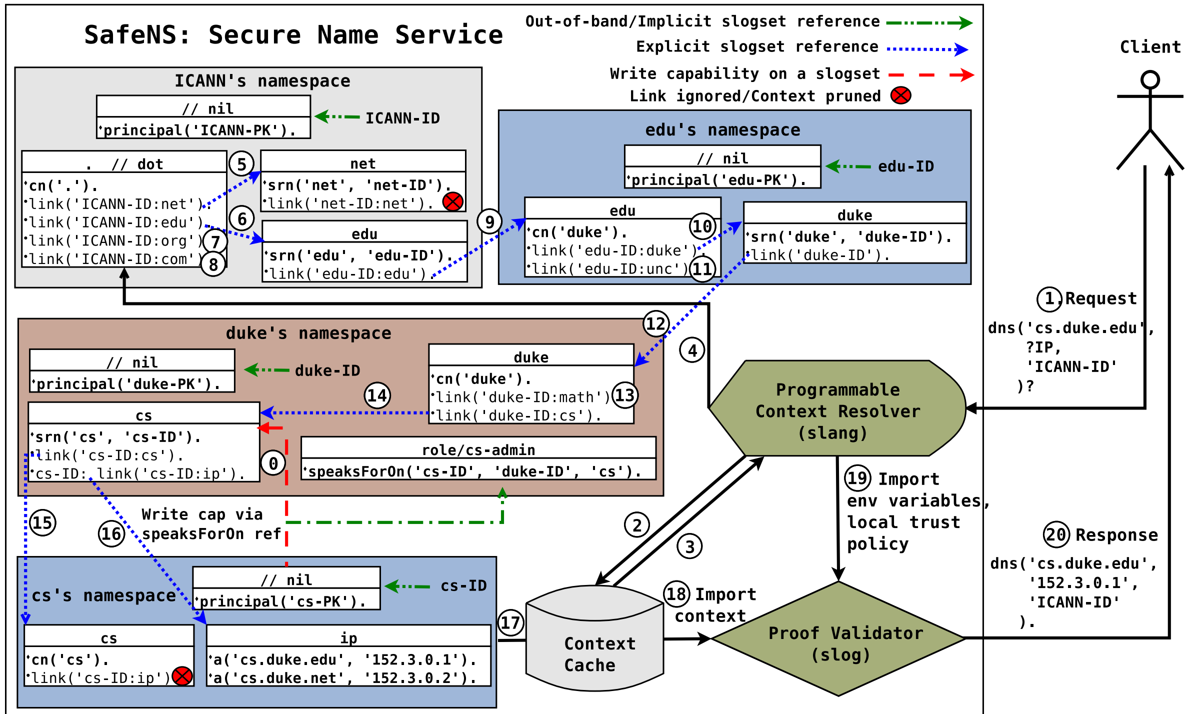
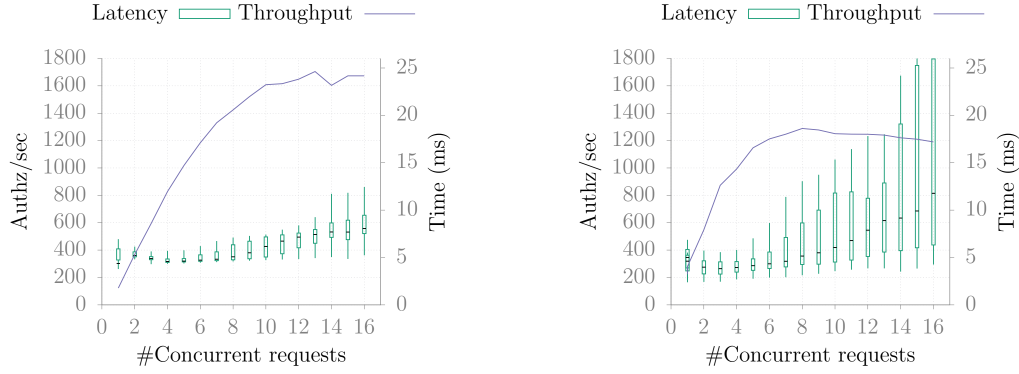


FIGURE 11.3: Workflow illustrating credential discovery, context building, context caching, and proof validation process for a secure name service, SafeNS, implemented using SAFE. The credentials are issued a priori (step 0) and materialized as slogsets in a shared distributed store (SafeSets). The principal `cs` writes to a slogset owned by `duke` due to a `speaksFor*` delegation capability issued by the owning principal (step 0). The SAFE process starts with a bearer slogset reference (`ICANN-ID`) provided by the client and builds the credential graph by traversing via linked references, and then tailors the context as per the resolver's programming logic written in slang (steps 1-17). The slang runtime invokes the slog interpreter by importing the relevant context. The slog interpreter validates the proof based on local trust anchors and policies, and certifies the response (steps 18-20).

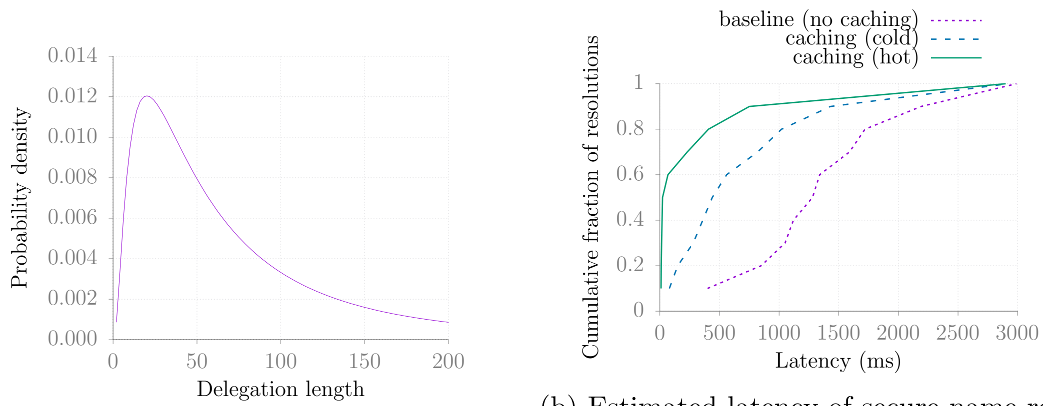
the root (`cn(.)`) with the first name token '`.`' by issuing a slog query. If the slog query returns true—i.e., the safe resource name (SRN) binds/matches with the slogset local name given by the `srn()` predicate—then the search continues further following the `link` predicates until a closure is reached. The slang runtime builds a tailored context based on the SRN, and once the search completes, slang invokes slog with the relevant proof context. The slog process validates the proof based on local trust anchors (`ICAANN-ID`) and policies, and certifies the response. The workflow also illustrates the use of `speaksForOn` issued by the principal `duke` delegating ownership



(a) SafeNS proof validation cost for delegations of length 4

(b) SafeNS authorization cost with delegations of length 8

FIGURE 11.4: Performance comparison of SafeNS with varying delegation lengths. The delegation length four is the average delegation length of a DNS service. The delegation length of eight is the observed maximum delegation length of naming systems from a sample DNS trace.



(a) SafeNS workload distribution with $\mu = 4$ and $\sigma = 1$

(b) Estimated latency of secure name resolutions in a simulated multi-domain naming system with varying delegation lengths following workload distribution as in 11.5a.

FIGURE 11.5: Impact of caching on secure naming resolution using SafeNS. The figure on the left shows the sample workload distribution we used to simulate the DNS traces with varying delegations following the lognormal distribution. The figure on the right shows latency measurements plotted against cumulative distribution function.

to the principal `cs` on a particular set named `cs`.

SafeNS resolver curtails the proof context at each stage of the credential discovery by using a constrained function `fetchSRN()` rather than `fetch()` (see Table 6.3). `fetch()` fetches the transitive closure name service endorsements starting from root

to the proof context, which is prohibitive if not curtailed to the relevant context. It is important to note that `fetchSRN()` is implemented as a slang library function (in 30 lines of code) rather than a native implementation. The context resolver is programmable: for example, the authorizer can add custom rules to accept content only from authoritative servers located in the US region.

If we assume that linking of slogsets is precise, i.e., fetching the closure of bearer reference would produce the credential graph, which is exactly equivalent to the final proof chain, then SafeNS emulates SD3 implementation for DNSSec. Both use two-level approach: credential retrieval followed by credential evaluation. Although in SD3, credential retrieval is equivalent of finding the proof and evaluation is an optional step, SafeNS requires both steps to certify the proof.

For analyzing the performance, we simulated a multi-component workload with varying delegation lengths. The workload is generated using a lognormal distribution with skewness (σ) as one and mean (μ) as four (see Figure 11.5a). The delegation length of four the average delegation length of a DNS service. Figure 11.4a and 11.4b shows the performance comparison of securing naming using SafeNS with varying delegation lengths of four and eight respectively. With full caching enabled, we achieved the peak throughput of 1600 for delegation length four and peak throughput of 1300 for delegation length eight respectively. The latency in both the scenarios is well within 20ms.

Our next experiment involves measuring the performance of the mixed workload under different caching techniques. Figure 11.5b shows the estimated latency plotted against the cumulative fraction of secure name resolutions. The baseline measurements fetches all the certificates over the network without caching them. The caching (cold) involves caching the common root prefixes in memory—such as `com, info, net, org, and edu`—upto two levels. The caching (hot) involves caching the verified certificates, i.e., slogsets, and proof contexts. With hot caching, all the popular

certificates are cached and there is minimal network overhead. Hot caching primarily measures the latency of certified validation. With no caching, each lookup is made for each part of the DNS name. The distribution of lookups is the same of distribution of the credentials, which are proportional to the name lengths. With prefix cache, the average lookups reduces to two per name resolution. For 54% of the requests, the latency improves over 3x. With full caching enabled, for 78% of the requests, the latency improves by 7x over no caching and by 3x over prefix caching.

11.4 SafeCoda: A Secure Cooperative Data Analytics

Consider a scenario where a group of data owners are willing to cooperate and share their sensitive data to a trusted third party either for mutual benefit or for global utility (e.g., sharing GENOME data) as long as the trusted party adheres to individual secrecy constraints of the participants. The trusted party may perform joint analytical computation on the data and may disclose the result only if there is an explicit mutual consent from all the participants involved. We refer to this scenario as secure cooperative data analytics. For example, participants in a medical research study may like to cooperative by sharing their data to a researcher (a trusted third party) as long as the researcher provide incentives to both the participants (i.e., diagnosing a disease or curing an illness) and does not disclose the individual data voluntarily. Secure support for cooperative analytics can help break down barriers to obtaining value from data.

One approach to solve cooperative analytics is to ensure the trust properties/attributes of each participant are compliant with their local policies at every level of the workflow. The participants tag their data with trust attributes and the trusted party (a software entity that is verified, attested, and trusted by all the participants) performs the joint computation of the workflow and reveals the results as per the participants policies.

SafeCoda supports access control based on named tags, which are authenticated security attributes. In particular, users label their objects with tags, and delegate *clearance* or *authority* for those tags to other principals and programs. Clearance for a tag confers permission to access data labeled with the tag. SafeCoda provides flexible delegation primitives and safe metadata storage to define tags and control permissions in the system.

SafeCoda implements *decentralized information flow control*, in which the system tracks to ensure that the security label for each object reflects the potential sensitivity of its contents. SafeCoda defines two types of tags: sensitivity tags and integrity tags. Sensitivity tags may have an associated numeric value that represents a *sensitivity level*. Clearance and authority permissions for these tags also have a value indicating the maximum clearance level for access or a minimum sensitivity level for declassified data. In SafeCoda, we focus on the sensitivity tags.

For secure cooperative analytics, the data owners tag the data with sensitivity tags and may delegate capabilities on these tags to other subjects. The requester for reading or writing the data must present the set of tags to the authorizer to gain access. The authorizer performs an inference by fetching all the tags on the data object (from SafeSets) and verifies whether the requester has access to all the tags by taking the union of tags. In SafeCoda, these authorizers are typically the "signed programs" executing on a secure "sealed service" that performs the analytics. SAFE achieves the strong identity of the signed programs by attesting the attributes of the programs by an independent code authority.

In our experiments, we use Spark as the computation engine that performs the data analytics. In particular, we launch a Spark cluster as a PaaS (Platform-as-a-Service) and run analytics applications on the platform. To protect sensitive data, we add a module called SafeRegistrar into the PaaS platform to collect sensitivity tags of source data and to maintain the sensitivity of all derived data sets throughout

```
defcon issueTag(?SetName, ?Subject, ?TagName, ?TagValue) :-
  "$SetName"{
    hasTag($Subject, $TagName, $TagValue).
  },
end
```

Code Snippet 11.2: Code run by SafeRegistrar on the behalf of each principal for issusing tags.

```
defcon delegateTag(?SetName, ?Subject, ?TagName, ?TagValue) :-
  "$SetName"{
    classifyTag($Subject, $TagName, $TagValue).
    declassifyTag($Subject, $TagName, $TagValue).
  },
end
```

Code Snippet 11.3: Code run by SafeRegistrar on the behalf of each principal for delegating tags.

SafeCoda. SafeRegistrar intercepts all requests received between the Spark and the file system (HDFS). Our initial evaluation shows the end-to-end latency for a joint computation in Spark using SafeCoda results in penalty of less than 10%. A detailed evaluation is left to future study.

We present the authorizatin logic rules for SafeCoda to demonstrate the flexibility of SAFE in implementing tag-based access control and decentralized access control models. In SafeCoda, each principal may posses a tag and a value. The tag is issued with a `defcon` rule as shown in Code Snippet 11.2. In addition, each principal can delegate the tag privileges by issuing the `classify` or `declassify` tag statements as shown in Code Snippet 11.3. Now, given a principal P with query label L and list of files F that the query will access, the SafeRegistrar will fetch all the tags assigned for the files F, computes the union of those tags U, and authorizers the query only if each tag in L is also present in U.

11.5 SafeGENI: Authorization in Federated IaaS Cloud Service

GENI is a NSF initiative for cooperative resource sharing among research institutions and organizations across the nation [GEN15]. Specifically, GENI is an infrastructure-as-a-service (IaaS) architecture for *multi-domain* clouds with multiple resource providers that enable users/experimenters to build custom combinations of resources for computing, networking, and storage. GENI is cooperative IaaS cloud where resource providers may authorize/control the allocation of resources as per their local policies. This is in contrast to traditional IaaS cloud providers such as Amazon AWS [AWS15], Microsoft Azure [Azu15], and Google Cloud [GCE15], which are administrated globally with a single enforceable policy.

GENI provides common abstractions through controllers, aggregates, and coordinators for managing users, resource providers, and mediating the interactions among them. Figure 11.6 illustrates the components of GENI architecture. A *controller* is a software entity that speaks for a user and provides common tools to access GENI services. An *aggregate* (or an aggregate manager) is a software entity that speaks for a resource provider. An aggregate may act as a "local GENI proxy" for some other resource service, such as a cloud site manager or standalone network testbed. A *coordinator* is an entity that mediates interactions among controllers and aggregates. Coordinators provide common services such as authorization and user activity monitoring. Resource providers via aggregates and coordinators interact and cooperate to serve users.

SafeGENI is an authorization system for GENI built using SAFE where the trust relationships are expressed in logic declaratively. We note that the existing GENI architecture is implemented using a structured URN naming scheme, which encodes various attributes and relationships into names directly. The GENI architecture relies on static policies and a global naming authority [CT14a]. However,

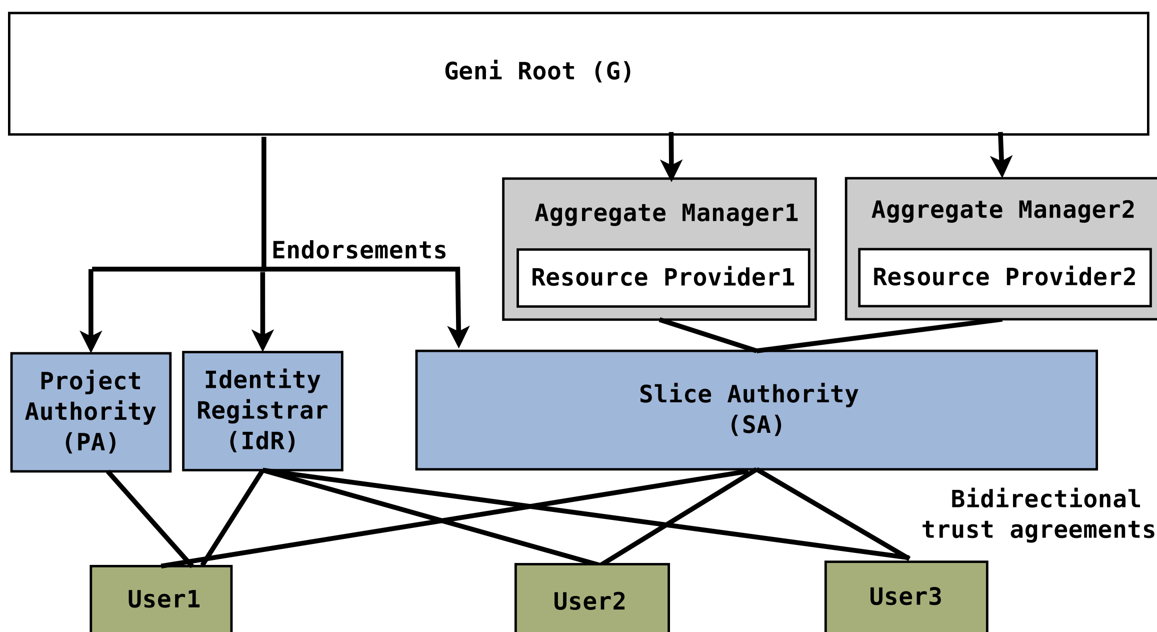


FIGURE 11.6: Overview of GENI components and its trust structure. All participating aggregates and coordinators (IdR, PA, SA) are endorsed by a common trust anchor called the federation root (G). Entities join the federation when they accept the root (G) as the trust anchor and are endorsed by the root. Entities install standard local rules to accept other entities endorsed by the root. The resource providers represent federation-approved cloud sites via aggregates. The trust structure emerges by inference from declarative statements of the participating entities.

one primary advantage of using SAFE is to decouple the trust logic from the system implementation. The approach of SafeGENI is flexible and extensible to any network architecture with federated trust structure. The design of SAFE was driven in part by the lessons of GENI experience and challenges encountered during the earlier research for authorization system for GENI.

SafeGENI provide services to authorize users, authorize slices, and bind attributes to users and slices. The principals in SafeGENI are the users, providers, and the coordinators. More precisely, the principals are software entities controlled by various identities in the real world, i.e., individuals and organizations. In SafeGENI, a unit of resource allocation and authorization is called a *slice*. More precisely, a slice is a logical container for a set of virtual resource elements that can be named and

instantiated upon a user request. GENI also uses the term *sliver* to refer to a virtual resource instance to encompass a diversity of virtual resource types. Each sliver is a member of at most once slice. Since the authorization is at the granularity of slice, if a provider grants an entity privilege to control a slice, then that entity can control any local sliver in the slice.

The privileges of a user in GENI to operate on slices and slivers depends in part on the user's membership in groups associated with specific activities. In GENI, such groups are called projects. GENI defines three types of coordinator entities (see Figure 11.7) to manage the identity of users, projects/groups, and slices.

- Identity Registrar (IdR) is an entity that is trusted to assert attributes of a real-world identity. Every GENI user must possess a keypair that is endorsed by a GENI-approved IdR. A project investigator (PI) is a special user that has additional privilege to create a project.

- Project Authority (PA) approves the creation of projects/groups. A decision to approve a project is based at least in part on validated attributes of the requester. PA acts as the root entity for projects that it approves: it issues a credential declaring the project and binding it to a name and other attributes. The PA also issues a credential designating the requester as the owner of the new project. The owner of a project may delegate membership privileges to other users.

- Slice Authority (SA) approves the creation of slices. A decision to approve a slice is based at least in part on validated attributes of the requesting user and the user's association with a project. An SA acts as the root entity for slices that it approves: it issues a credential declaring the slice and binding it to a project, a name, and other attributes. The SA also issues a credential

designating the requester as the owner of the slice, conferring a privilege to control the slice.

In SafeGENI, a requester's right to create a new slice is determined by its membership in a project. Each slice is bound to exactly one project at the time the slice is created. The project and its owner are accountable for activity in the slice. Slice creation is permitted only for users wielding a specific name privilege (`instantiate`) withing the project. Upon creating the slice, other users may control operations on the slice (`start, stop, info`) based on its delegate privileges. Aggregates determine those privileges based in part on statements made by SA that approved the slice. In particular, the SA names the owner of the slice and the project responsible for the slice. A capability-based access control model enables the owner of a slice to delegate control privileges to other users. Table 11.4 shows the simplified API for GENI testbed.

11.5.1 Trust Structure

Figure 11.6 shows the trust structure in GENI with a federated root (`?Geni`). The root endorses the coordinators project authority (`?PA`), slice authority (`?SA`), identity registrar (`?IdR`), and aggregate managers (`?AM`). For example, consider how SafeGENI servers validate a principal is an "authorized" IdR by delegating the control to the GENI federation root and including the following rule in their policies:

```
endorse(?IdR, identityRegistrar) :-
  endorse(?Geni, geniRoot),
  ?Geni: endorse(?IdR, identityRegistrar).
```

This rule delegates to a GENI federation root (`?Geni`) the power to say who is an authority in the federation. It states that a principal is an IdR authority if the

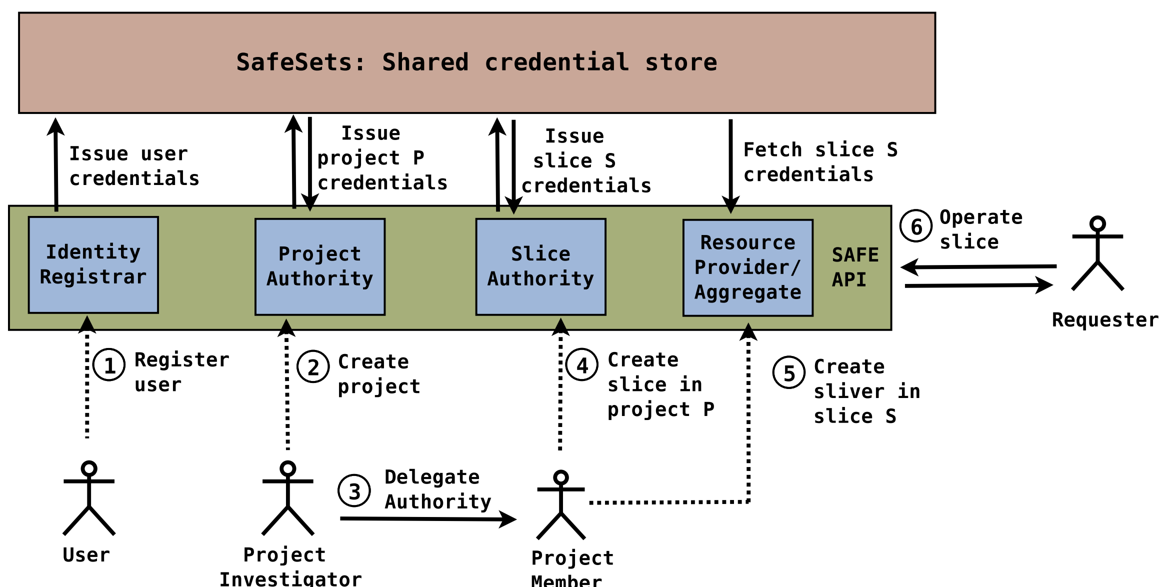


FIGURE 11.7: SafeGENI credential workflow for a typical resource request ("slice") in GENI. The users and project investigators register to corresponding coordinator entities IdR and PA and issue credentials to SafeSets. Each virtual resource instance ("sliver") is obtained from a single resource provider (via aggregate manager interface) and is bound to a project and slice that have been approved by authorities trusted by the aggregate according to its local policy. The authorities approve users, projects, and slices by issuing credentials. The aggregator guards the standard controls on a slice (`instantiate, start, stop, info`) through SAFE service interface. The requester passes a bearer reference and the aggregator fetches the corresponding credentials and runs SAFE inference to authorize the request. The downstream services that fetch from SafeSets credential store may cache the credentials locally.

federation root says/endorses the same. Similar rules can be made for servers that validate a principal as an authorized PA, SA, and AM.

To satisfy these rules, the federation root might simply issue endorsements to the IdR, PA, SA, and AM servers. Code Snippet 11.4 shows an example of GENI root issuing an endorsement for IdR.

An authority server includes any endorsements from the federation root in its subject set, and attaches it to the credentials it issues for inspection downstream (see Code Snippet 11.5). That is sufficient to implement the present GENI federation.

Table 11.4: Simplified trust API for GENI, a federated IaaS cooperative cloud testbed. These APIs are implemented in SafeGENI as trust-logic rules. With SafeGENI each API exposes a RESTful service interface and trust attributes/credentials are collected from the `$BearerRef` environment variable passed along with the request.

Caller	**Method**	**Description**
Root	`endorseAggregate(PID)`	Issue root endorsement for an aggregate (resource provider) with `PID` as the identity.
Root	`endorseAuthority(PID, authorityID)`	Issue root endorsement for an authority that certifies users (IdR), projects (PA), or slices (SA).
PA	`createProject(ownerPID, attributes): projectID`	Create a project for a subject `ownerPID` if the subject possesses valid privileges. This is an API call of a Project Authority (PA), which returns a newly created `projectID`.
User	`member(PID, projectID, delegatable)`	Delegate membership on a project `projectID` to a subject `PID` specifying whether the privilege is further `delegatable`.
SA	`createSlice(ownerPID, projectID, attributes): sliceID`	Create a slice with subject `ownerPID` in a project `projectID` if the subject possesses valid privileges. This is an API call of a Slice Authority (SA), which returns a newly created `sliceID`.
User	`delegateSlice(PID, sliceID, attributes, delegatable)`	Delegate named permissions/attributes to operate on a slice `sliceID` for a subject `PID` specifying whether the permissions are further `delegatable`.
Aggregate	`createSliver(sliceID, attributes)`	Check requester's permission to instantiate a virtual resource for use at this aggregate.
Aggregate	`sliceOperation(sliceID, operationType, attributes)`	Check requester's permission to perform a control action on a slice's resources at this aggregate.

Next, consider how the IdR itself endorses users. An identity provider's authority to issue credentials flows from its own endorsement by a GENI root, as shown in Code Snippet 11.4. To substantiate the credentials it issues, the endorser may link its own

```
1  defcon idpEndorsement(?IdR) :-
2    spec('endorse an IdR')
3    ''endorse/$IdR''{
4       endorse(?IdR, identityRegistrar).
5    }
6  end
7
8  definit post(idpEndorsement($1)).
```

Code Snippet 11.4: Code run by GENI federation root (G) to issue an endorsement for one of its coordinators IdR.

```
1  defcon addTokenToSubjectSet(?SetRef) :-
2    spec('add a link to subject set')
3    ''subject/$Self''{
4      link(?SetRef).
5    }
6  end
7
8  definit post(addTokenToSubjectSet($1)).
```

Code Snippet 11.5: Code run by IdR to add the endorsement link issued by Geni root from Code Snippet 11.4.

credentials into the endorsement set. The slang code in Code Snippet 11.6 endorses a principal as a geniUser and a geniPI, including a link to the issuer's own subject set. If the issuer is an authorized identity provider then its subject set contains its endorsement delegating that authority from the GENI root.

Code Snippet 11.6 is also an example of slogset linking in SAFE. Set linking via linked subject set makes the issuer's (IdR) credentials available for inspection by anyone who receives the endorsements that it issues.

As the federation evolves, the root may define additional rules to implement changes to the trust structure. These rules may be incorporated into the policies of other federation members and applied automatically, without changing their code. In particular, the root may add rules to peer with another federation according to various terms and policies. § 11.5.5 discusses some example alternatives.

11.5.2 Objects

```
defenv SubjectSet :- getID($Self, ''subject/$Self'').

defcon endorsePI(?User) :-
  spec('endorse a project investigator')
  ''endorse/$User''{
    endorse($User, geniUser).
    endorse($User, geniPI).
    link($SubjectSet).
  }
end

definit post(endorsePI($1)).
```

Code Snippet 11.6: Code run by IdR to endorse a user as a project investigator (PI).

Slices and projects are "global objects" in GENI. What makes them global is that multiple servers make access control decisions regarding these objects. For example, Code Snippet 11.7 shows how a Slice Authority requires a requester to have specific access rights (via possession of `instantiate` attribute) for a project in order to create a GENI slice in the project. The Slice Authority bases its choice entirely on statements about the project by third parties, including the project's root principal (a Project Authority) and its delegates. Similarly, a requester's right to create or control a sliver at an aggregate is determined solely by the privileges that it holds over the containing slice.

The `createSlice()` example illustrate the flexibility of guards in SAFE. In particular, they show how a service may delegate the policy rules for the privilege check on the target object to another principal. These guards delegate policy control over the target object to its root principal as returned by `rootID()` builtin. For example, the root principal for a slice is the Slice Authority who approved creation of the slice, assigned it a name, and endorsed it.

The policy delegation on the target object relies on mobility of policy rules, and the inherent ability of logical trust systems to reason from policy rules issued by other principals. An imported policy rule enables an authorizer to infer beliefs of the issuer, given some set of facts. For example, the `memberPrivilege` goal in a

```
1  defcon standardSlice(?Project, ?Owner, ?SlicePolicy) :-
2    ?Slice := genID(), // generate a new GUID
3    "capability/$Subject/$Slice"{
4      owner($Subject, $Slice).
5      slice($Slice, $Project, standard).
6      link($SlicePolicy) :- spec('link to standard slice control
        ↪  policy').
7    }
8  end
9
10 defcon sliceAuthorityPolicy() :-
11   spec('a policy for approving slice creation'),
12   'policy/standardSliceControl'{
13     approveSlice(?Project, ?Owner) :-
14       ?PA := rootID(?Project),
15       endorse(?PA, projectAuthority),
16       ?PA: project(?Project, standard),
17       ?PA: memberPrivilege(?Owner, ?Project, instantiate),
18       endorse(?Owner, geniUser).
19   }
20 end
21
22 defguard createSlice(?SliceCreatePolicy, ?SliceControlPolicy) :-
23   spec('verify whether the ?Subject has create slice capability
24         create a slice and post slice credentials to SafeSets'),
25   {
26     import!($BearerRef).
27     import!(?SliceCreatePolicy).
28     approveSlice($Object, $Subject)?
29   },
30   post(standardSlice(?Object, ?Subject, ?SliceControlPolicy))
31 end
32
33 definit ?SliceCreatePolicy := sliceAuthorityPolicy(),
34   // ?SliceControlPolicy is predefined and reference is obtained via
      ↪  command line args $1
35   createSlice(?SliceCreatePolicy, $1)
36 end
```

Code Snippet 11.7: Code run by SA to create a slice requested by the `?Subject` by passing the `?BearerRef` and `?Object` (project) references.

SA's `createSlice()` guard is satisfied if the PA itself has issued policy rules for `memberPrivilege` for the project, and the goal can be inferred from those rules and the credentials passed by the requester (via `$BearerRef`). The PA and SA may define the policy rules governing access for each project and each slice, and may specify different rules for different objects.

Using the SAFE conventions, objects are created as 160-bit GUID with its owner as the prefix—thus returning a self-certifying identifier that is globally unique. For example, the `scid()` function in Code Snippet 11.7 `standardSlice()` mints a new 160-bit GUID and prefixes the controlling principal `$Self` delimited by `':'`.

11.5.3 Capabilities and Delegation

When objects are created, the root principal may attach default policy rules on that object. For example, consider the `createSlice()` rule from Code Snippet 11.7. Once the requester is authorized to create a slice, a `defcon` rule called `standardSlice()` is invoked that creates a slice and links to the 'standard' policy set rules. The policy rules permit the owner to delegate control to other principals. Code Snippet 11.8 illustrates a `defcon` rule to construct the policy set.

The first two rules in the constructed policy set (line 5-6) states that the owner of a slice has a controls privilege. The `controlsStar` predicate indicates that the `controls` is delegatable. By transitive closure, the `controls` previlege is inferred via `controlsStar` (line 11-17). Any subject that `controls` a slice inherits the `instantiate` privilege on that slice (line 8-9). The rules with heads `grantCap` and `delegateCap` defines how an authorizer must interpret the delegation: if a principal `?Delegator` with a `delegateCap` says that another principal `?User` has a `delegateCap` privilege, then it is so (line 19-25). The `grantCap` assigns a non-delegatable privilege where as the `delegateCap` assigns a delegate privilege.

```
1  defcon standardSliceControl() :-
2    spec('policy rule set for standard slice'),
3    'policy/object/standardSliceControl'{
4
5      controls(?Subject, ?Slice) :- controlsStar(?Subject, ?Slice).
6      controlsStar(?Subject, ?Slice) :- owner(?Subject, ?Slice).
7
8      grantCap(?Subject, ?Slice, instantiate) :- controls(?Subject,
       ↪  ?Slice).
9      delegateCap(?Subject, ?Slice, instantiate) :-
       ↪  controlsStar(?Subject, ?Slice).
10
11     controlsStar(?User, ?Slice) :-
12       controlsStar(?Delegator, ?Slice),
13       ?Delegator: controlsStar(?User, ?Slice).
14
15     controls(?User, ?Slice) :-
16       controlsStar(?Delegator, ?Slice),
17       ?Delegator: controls(?User, ?Slice).
18
19     grantCap(?User, ?Slice, ?Priv) :-
20       delegateCap(?Delegator, ?Slice, ?Priv),
21       ?Delegator: grantCap(?User, ?Slice, ?Priv).
22
23     delegateCap(?User, ?Slice, ?Priv) :-
24       delegateCap(?Delegator, ?Slice, ?Priv),
25       ?Delegator: delegateCap(?User, ?Slice, ?Priv).
26
27   },
28 end
```

Code Snippet 11.8: Example rules demonstrating the capability-based access control implemented in trust-logic. These rules allow a slice owner to delegate control to another user.

Issuing a delegation is not different from issuing any kind of endorsement (see Code Snippet 11.4). Once the delegation is issued, the receiver must obtain the slotset reference by some out-of-band communication mechanism. The receiver then runs the `addTokenToSubjectSet()` as per the Code Snippet 11.5 and links the slogset

reference to its subject set.

11.5.4 Refining Capabilities

The object delegations in § 11.5.3 are a form of capability-based access control, specified and implemented in trust logic. The example in Code Snippet 11.8 shows *confinement*, the ability of a delegator to block the receiver from delegating the privilege to a third party. We now show how to expand the capability support for *refinement*: the ability of a delegator to constrain the access privileges under a delegation. Refinement is an important element of capability models. To support refinement, the slice privilege set adds the rules as illustrated in Code Snippet 11.9.

The first four rules confer the standard control privileges `instantiate, info, start, stop` to `?Subject` with `controls` privilege. The fifth rule permits a holder to delegate either privilege independently of the other.

11.5.5 Peering

Suppose the federation root wants to peer with another SafeGENI federation whose root is given by the environment variable (`$Peer`). To accept users from the peer federation, it could simply add the following policy rule to its subject set:

```
endorse(?IdR, identityRegistrar) :- $Peer : endorse(?IdR,
 ↪  identityRegistrar).
```

With the addition of this rule (and a link to the peer's subject slogset), all services in the local federation accept user endorsements from identity providers in the peer federation. It enables the peer's users to join projects and access slices, subject to additional rules for membership and access that require consent of the project leaders and slice owners. The root could go further by adding a similar rule to accept the peer's Project Authorities, enabling members of the peer's projects to create slices and request resources for them even without consent of local project

```
1  defcon standardSliceControl() :-
2    spec('policy rule set for standard slice'),
3    'policy/object/standardSliceControl'{
4       grantCap(?Subject, ?Slice, instantiate) :-
5         controls(?Subject, ?Slice).
6
7       grantCap(?Subject, ?Slice, info) :-
8         controls(?Subject, ?Slice).
9
10      grantCap(?Subject, ?Slice, start) :-
11        controls(?Subject, ?Slice).
12
13      grantCap(?Subject, ?Slice, stop) :-
14        controls(?Subject, ?Slice).
15
16      grantCap(?Subject, ?Slice, ?Priv) :-
17        delegateCap(?Delegator, ?Slice, ?Priv),
18        ?Delegator: grantCap(?Subject, ?Slice, ?Priv).
19   },
20 end
```

Code Snippet 11.9: Example rules demonstrating the capability-based refinement implemented in trust-logic. These rules allow a delegator to constrain the access privileges permitted under a delegation.

leaders. Accepting the peer's Slice Authorities enables slices from projects in the peer federation to request resources from local aggregates.

Accepting the peer's authorities in this way is a simple option for cross-federation, but it fully permissive and does not distinguish guest principals from the members of the local federation. A preferable alternative is to distinguish users, slices, and/or projects originating from the peer. For example, consider these root policy rules:

```
endorse(?User, guestUser) :-
  ?IdR: endorse(?User, geniUser),
  $Peer: endorse(?IdR, identityRegistrar).
endorse(?User, geniUser) :- endorse(?User, guestUser).
```

These rules enable peer users to act with the same privileges as local federation users, according to the SafeGENI guards. But the rules confer no PI privilege to the

peer's PIs, so they cannot, for example, create projects in the local federation.

Alternatively, if the second rule is deleted, then guest users have no access privilege unless other rules affirmatively grant it. In this way the guards may limit the scope of their privilege.

Note also that the first rule enables an authorizer to distinguish a guest user. For example, this rule may be useful if the authorizer's guard could deny access based on whether the user is a `geniUser` or a `guestUser`. Since SAFE allows limited form of negation via *deny conditions* (blacklists) in queries, we can rewrite the query for `defguard` rule `createSlice()` from Code Snippet 11.7 as shown in Code Snippet 11.10.

```
1  defguard createSlice(?SliceCreatePolicy, ?SliceControlPolicy) :-
2    spec('verify whether the ?Subject has create slice capability
3          create a slice and post slice credentials to SafeSets'),
4    {
5      import!($BearerRef).
6      import!(?SlicePolicy).
7      !guestUser($Subject), approveSlice($Object, $Subject)?
8    },
9    post(standardSlice(?Object, ?Subject, ?SliceControlPolicy))
10 end
```

Code Snippet 11.10: Example of blacklisting a user from peer federation (`guestUser`) from creating a slice in local federation.

A third option for cross-federation peering is for individual authorities or aggregates to accept upstream authorities from a peer federation. For example, an aggregate may simply accept multiple federation roots, and thereby grant access to all the users, slices, and projects of multiple federations on an equal basis.

It is an open question whether this permissiveness would be allowed under the rules of any individual federation. That is, by accepting foreign users, the aggregate may be violating its agreement with a sponsoring federation. The nature of the

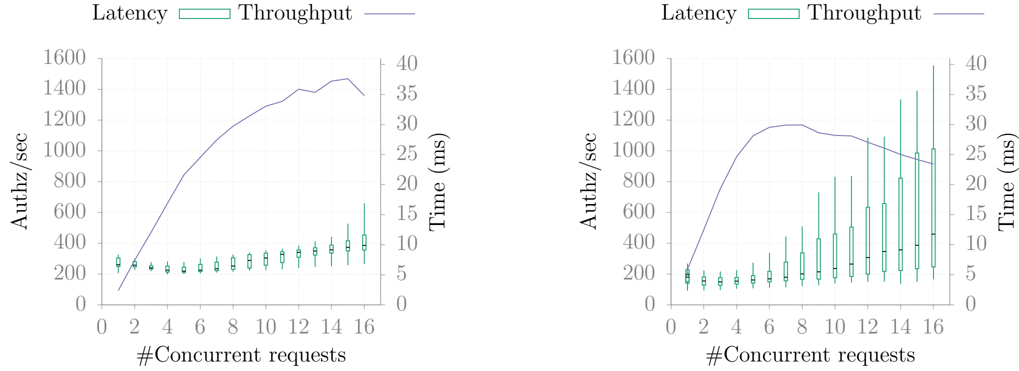
agreements that govern a priori trust relationships are outside the scope of SAFE. For example, SAFE can represent the trust policies to implement an aggregate's usage agreement with a sponsoring federation, but it cannot represent the agreement itself or force the aggregate to use only policies that comply with its agreement.

11.5.6 Evaluation

SafeGENI is implemented in less than 100 rules combining both slang and slog. A major motivation for SafeGENI is capture the trust agreements precisely and evaluate their complexity. Once the policies specified in logic, the authorization system is directly depolyable with SafeGENI without custom code modifications in the diverse GENI provider implementations.

To measure the impact of caching and context linking on performance, we benchmark SafeGENI using a standard GENI workload with a mix of N users and M resources. We set up delegation chains by following a geometric distribution as illustrated in Figure 11.10. In our experiment, half the users are reachable directly from the PI, i.e., these users are within a delegation of length one from the PI. Similarly, following the geometric distribution, 1/4th of the users are reachable within delegation of length two from the PI, and delegation of length one from the first user. The process continues until all the users are connected with at least one other user who has access to the GENI resource.

We measure the throughput and latency of the mix under three scenarios: (i) Baseline case: all certificates are processed in their entirety for each request with no caching involved. This includes retrieving all certificates from in-memory, validate crypto and speakers, render them to set cache, merge them to context cache, and querying the inference. (ii) Cold caching with monolithic contexts: the raw certificates are cached in memory but the slogset cache and the context cache are build on-demand. These context caches and slogset caches are monolithic in that



(a) SafeGENI authorization cost with delegations of length 2

(b) SafeGENI authorization cost with delegations of length 6

FIGURE 11.8: Performance comparison of SafeGENI with varying delegation lengths. The delegation length 2 is the average delegation length for the experiment setup with logarithmic delegation chains. The delegation length of 6 is significant due to power law distribution that demonstrates at most six-degrees-of-separation between human connections.

their life span tailored to a given request. (iii) Hot caching with set references and proof context cache enabled. Here we cache the proof contexts which enables shared credentials among queries is readily available through the context cache.

Figure 11.8 shows the performance comparison of SafeGENI with varying delegation lengths of two and six. The delegation length two is average delegation length for the experiment setup with logarithmic delegation chains. The delegation length of six is significant due to power law distribution that demonstrates at most six-degrees-of-separation between human networks. The plots show that a throughput scales linearly with the number of concurrent requests and achieves 100 authz/sec per core for the average delegation length of two. When number of delegations are increased to six, the throughput peaks at six cores and drops with increasing latency. This is due to increase in wait time per request at high concurrency.

Figure 11.11b shows the peak throughput measurements for the three scenarios with N set as 1024 and M set as 4096. The throughput for baseline case flattens out fast as expected since each request is processed in its entirety, i.e., by validating the

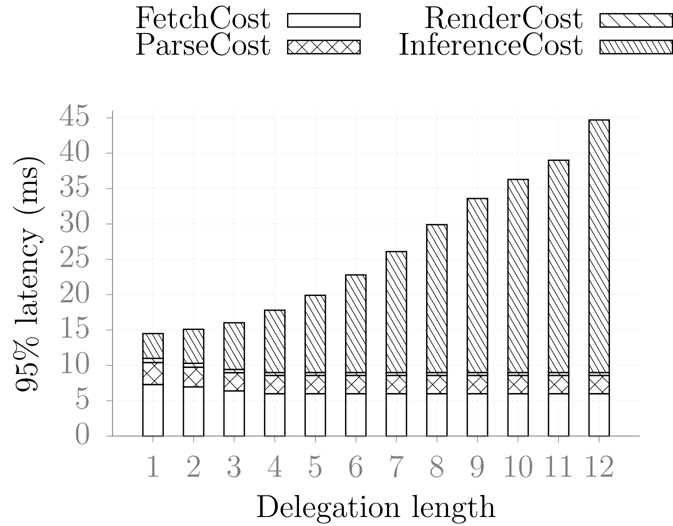


FIGURE 11.9: End-to-end authorization costs of SafeGENI with varying delegation lengths.

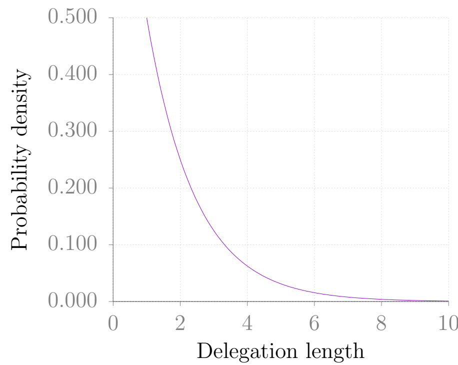
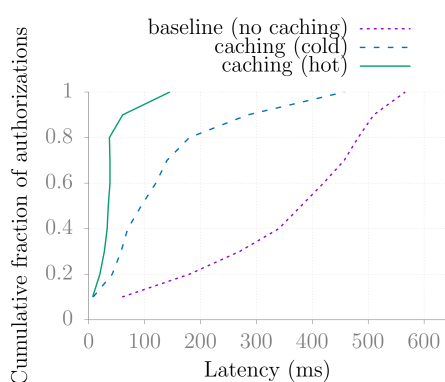
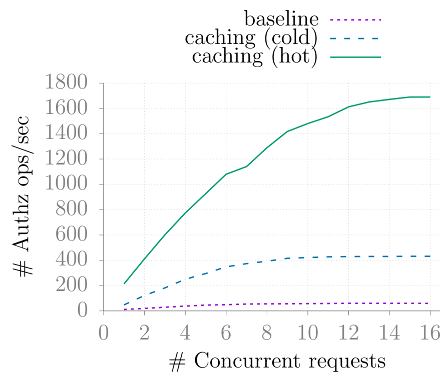


FIGURE 11.10: Delegation patterns of SafeGENI follows a geometric distribution with half of the users are reachable from the PI directly, i.e., within a delegation of length one.

certificates, ripping the contents, and evaluating the query. Caching the certificates improves throughput by 6x. However, the throughput flattens out after 10 concurrent requests. Figure 11.11a shows the latency measurements for the three caching configurations. With cold caching, the latency for 80% of the requests is improved by 2.5x and with hot caching, it is further improved by 4x. With hot caching, the proof contexts are shared across the queries resulting shared trust policies and credentials avoiding the re-rendering to proof context. The measurements show that



(a) Latency measurements of SageGENI.



(b) Throughput measurements for Safe-GENI.

Figure 11.11: Impact of caching on authorization in SageGENI. The figure shows the latency and throughput measurements plotted against the cumulative distribution function. For this experiment, we set N as 1024 and M as 4096, with delegations distributed according to the geometric distribution as illustrated in Figure 11.10.

throughput scales linearly with hot contexts and demonstrates the useful of caching proof contexts.

12

Summary

SAFE is a trust-agile system which supports delegation-driven credential linking using a high-level declarative language. SAFE provides an integrated system for end-to-end certificate management including credential discovery, context management, and certificate revocation. This chapter presents our experience with using SAFE (§ 12.1), the potential extensions and future work of SAFE (§ 12.2), and the summary of this thesis (§ 12.3).

12.1 Experience with SAFE

SAFE is born out of a need to address the challenges of managing trust and credential distribution for a distributed networked infrastructure-as-a-service initiative, the NSF's GENI system. SafeGENI served as a guiding example to abstract and extend the lessons learned into a generic framework resulting in a policy language "slang".

When we started the SafeGENI project, we identified credential discovery and context management are the open problems in managing trust. Initially, we attempted to use the Attribute-Based Access Control software from ISI [RFWS11], which is based on RT_2. We observed that the semantics of RT_2 are not well-defined,

except by their implementation. In our quest for identifying a simple and declarative trust language, we observed that there are many attempts in the literature that either (i) propose a powerful trust logic that is intractable, or (ii) solve an individual problem in the trust management subspace without a comprehensive end-to-end solution.

With that experience, we carefully crafted the features for SAFE. A key observation of our exercise is that we need two interoperable but independent layers for managing trust: (i) the inference layer that evaluates the proof locally (certified evaluation); and (ii) the context layer that provides a programming environment for writing trust policies using a scripting langugage. The inference engine must evaluate the proof quickly. In particular, termination on all inputs is a useful property to satisfy for practical inference. Slog enforces certain syntactic restrictions and designed without the standard features such as `speaksFor` delegation from the logic to remain tractable. The context scripting layer provides the credential linking outside the logic based on the delegation patterns of the application. In addition, it supplies the necessary proof context to run the inference. Slang is designed to support these features including non-tractable policies involving negation and aggregation outside the logic. Separating the proof context building from the proof evaluation is a salient feature of SAFE. These layers can be independently replaced with a more powerful trust logic or a different credential linking mechanism without effecting the overall functionality of SAFE.

Moreover, abstracting the set of credentials under a slogset made interfacing with the context layer and the storage layer transparent. In particular, slogsets can be published and encoded under varied certificate formats based on the application settings. Slogsets can be cached across the requests and can be consumed directly by the inference engine.

Credential linking solves the automated credential discovery and retrieval prob-

lem. Linking enables credentials to issued independently without a centralized controlling authority. The issuer may chose to chain the appropriate credentials via links based on the delegation patterns of the application.

We believe SAFE is a simple and an extensible language that is nevertheless sufficiently powerful for expressing practical trust policies. However, the flexibility of trust logics and in particular SAFE may demand further study. One main issue is that all the participant's must agree on the conventions used to express trust policies and credentials. For example, there is more than one way of stating the same attribute or a credential but all the participants must share a common understanding. In practice, this can challenging due to evolving and distributed nature of the participants. § 6.5 attempts to provide a simplified API and template library that address most of these issuers. In addition, the naming conventions in § 5.4.2 help in automated credential discovery.

Debugging is an issue with decentralized federated settings. It is not a surprise that the issue applies to SAFE as well. SAFE solves this partially by rendering `link` predicates as HTML hyperefs and making certificates navigable. Moreover, tailored proof contexts and certificated evaluation allows the user to manually observe the final end-to-end proof chain for which the query is satisfied. However, providing the precise proof context for an unsatisfied query is a challenging open problem. In summary, in our experience, programs written in SAFE are readable and easier to understand but not that easier to debug especially in the case of unsatisfied proofs.

12.2 Future Work

As discussed in § 12.1, SAFE may benefit from a more controlled API by tailoring to individual application domains at the slang layer. The flexibility of trust logics comes at a cost: all the participants must agree on the conventions used in expressing the credentials. The delegation patterns and trust structures vary across the application

domains. § 5.4.2 provides examples of general purpose APIs that capture groups, roles, and delegations for common applications. These APIs can be agumented to suit the needs of individual application domains.

SAFE benefits from a debugger that provides partial proofs if a query is unsatisfied. Ideally, the partial proof must be followed by a missing context that contains the required statements for the proof to be valid. Inferring missing statements is a challenging open problem as there can be exponential number of ways to reach the same proof path.

The performance of SAFE can be improved. In our current implementation, the parser is really slow (in the order of seconds per program) due to the use of parser combinators and packrat parser implementation in Scala. We believe the parser can be improved by an order of magnitude by choosing a hand-written parser or using standard parser generators such as ANTLR [ant16]. In addition, the unification module can be optimized by indexing terms.

SAFE relies on secondary indexes for queries that rely on attribute-based delegation. The API for supporting secondary indexes is not open to the programmer. We can use SWI-Prolog style "just-in-time" indexing to learn about the index on the predicates dynamically at runtime without relying on the programmer to specify them in advance. Restricting the slang API via templates allows some optimizations such as builtin secondary indexes for the required predicates.

Slang also may benefit from an optimizer that automatically translates the queries into multiple queries for slog and aggregates the answers. For example, aggregation and negation benefits from a builtin slang optimizer.

SAFE can be applied to secure other applications. For example, SAFE can be used beyond the access control for secure resource peering [FCC+03], where resources are virtual and dynamic but a global constraint (e.g., number of resources/leases per aggregator) needs to be met for cooperative decentralized resource management.

SAFE can also be used for secure networking: in particular, the access control policies on the control plane for software-defined networking [MAB+08, GFV16, BHG14], and software-defined Internet exchange [GVS+14] can be written in SAFE simplifying the policy logic.

We also posit SAFE is useful for expressing ontology's in networks. Currently, the ontology's are expressed using a different subset of logic such as OWL, RDF, and SPARQL with each having limitations in tractability and scalability.

12.3 Conclusion

Secure Access For Everyone is a trust-agile system that uses a declarative trust logic to represent policies, endorsements, and delegations. What is novel about SAFE is the integration of the trust logic with a scripting language ("slang") and shared storage abstraction for authenticated logic content. These elements work together to simplify and automate many aspects of networked trust. The SAFE abstractions *"delegation-driven credential linking"* and *"trust agility"* provide foundation for building secure network systems.

Slang provides programmable policy-driven credential discovery and solves the long standing open problem of identifying and accumulating the content that is relevant for a trust decision. Moreover, the agility of the authorizer to adapt to a user supplied content in the request enables hybrid access control models that are useful from centralized to federated settings. Slang also makes crypto operations transparent and supports programmable variables that makes policies easier to write and distribute via templates. Slog provides certified evaluation by validating each trust decision with an attested proof. Slogsets provide a new abstraction to capture the autenticated sets of logic statements.

With respect to implementation, we materialize slogsets as signed certificates, and store them in a scalable certificate store called SafeSets. Each stored certificate

is named by an identifier suitable for indexing and caching linked certificates or logic sets. SAFE also provides a service API for policies enabling easier integration of SAFE with existing application frameworks.

SAFE in combination with SafeSets address three fundamental problems: how to identify the content that is relevant to a given trust decision, how to manage the flow of credentials through the system including caching, and how to incorporate updates to outstanding certificates. Experience with wide range of applications from SafeGENI to SafeCoda shows that the approach is practical for a complex network trust system.

13

Glossary

This glossary presents relevant terms to this thesis grouped by their semantics—and not by their lexicographic order—intentionally.

- **Attribute.** An attribute is a name-value pair used to express some property of an entity.

  Example. "The sky is blue in color and vast in space."

  ```
  sky(color, blue), sky(space, vast).
  ```

  Here, 'sky' is an entity with two attributes: color and space with values 'blue' and 'vast' respectively.

  Note. The values of an attribute may not be constants: they can refer to other entities which in turn can recursively refer to other entities or constants—forming an entity graph. For example, 'blue' may be other entity with RGB values (173, 216, 230) and CMYK values (25, 6, 0, 10).

  ```
  blue(RGB, [173,216,230]), blue(CMYK, [25,6,0,10]).
  ```

  Alternatively, using SafeX, the same can be represented as:

```
sky(color -> blue), sky(space -> vast).
```

or in a closed system where the arity is well defined,

```
sky(color -> blue, space -> vast).
```

Question. Is it possible to have different interpretations for the same sentence?

Yes. Consider the example:

“The sky is blue and vast.”

```
sky(blue), sky(vast).
```

Three possible interpretations:

1. ‘sky’ is an attribute with values ‘blue’ and ‘vast’.
2. ‘sky’ is an entity with attributes ‘blue’ and ‘vast’ with both values set to ‘true’.
3. ‘sky’ is an entity with unspecified attribute names ‘color’ and ‘space’ with values set to ‘blue’ and ‘vast’ respectively.

In SafeX, attribute names can be unspecified, but the values must be symbolic constants or references to other entities. SafeX reflects logic programming approach where the predicates are populated with attribute values without explicitly naming the attributes. However, note that both approaches are sensitive to order of attribute values that makes the predicate.

In SafeX, these three are equivalent:

```
sky(blue, vast).
```

```
sky(?X -> blue, ?Y -> vast).
```

```
sky(?Y -> blue, ?X -> vast).
```

However, changing the order of attributes values changes the meaning of the predicate. Hence, the following expression is not equivalent to above.

```
sky(?X -> vast, ?Y -> blue).
```

- **Predicate.** Predicate are symbols used in logic to express some property of entities or some relation between entities. They are also sometimes called *relation* symbols. A predicate always evaluates to `true` or `false`.

  Example. "Duke is between UNC and NCState."

  ```
  between(Duke, UNC, NCState)
  ```

  Here, Duke, UNC, and NCState are the *arguments* of the predicate symbol 'between'. The *arity* of the predicate 'between' is 3, and written as 'between/3'.

- **Functor.** A functor is a non-logical symbol along with its arguments that forms a function. A function evaluates to a constant value unlike a predicate, which only evaluates to `true` or `false`.

  Example. "add values two and three"

  ```
  add(2, 3)
  ```

  Here, 'add' is a functor with arity 2.

  Sometimes, SAFE adds the suffix `!` to a functor to denote the side-effects such as writing to I/O.

- **Term.** A term is recursively defined as a constant or a variable or a functor with arity n.

  Example. "Duke is between east of some university and north of NCState."

  ```
  between(Duke, east(?University), north(NCState))
  ```

  Here, 'Duke', and 'NCState' are constant terms, '?University' is a variable term, and 'east(?University)', 'north(NCState)', and 'between(Duke, east(?University),

north(NCState)' are all predicate terms with arities one, one, and three respectively.

- **Atom.** An atom or an atomic formula is the predicate symbol applied to the arguments. An atom is the simple sentence with no logical connectives. Each argument of an atom is a term. Atoms are also referred as relational goals.

  Example. "Duke is between east of UNC and north of NCState."

  ```
  between(Duke, east(UNC), north(NCState)).
  ```

  Note: the difference between the predicate term and atom is that the atom is a simple sentence ending with '.'.

- **Ground atom.** An atom where all the terms are ground terms, i.e., the terms does not contain any variables. An ground atom is also called as *ground fact.*

  Example. "Duke is between east of UNC and north of NCState."

  ```
  between(Duke, east(UNC), north(NCState)).
  ```

- **Literal.** A literal is an atom (atomic formula) or its negation.

  Example. "Duke is not between west of UNC and south of NCState."

  ```
  !between(Duke, west(UNC), south(NCState)).
  ```

- **Sentence.** A collection of terms joined by logical connectives with no free variables. The logical connectives are *'and'* alias *','*, *'or'* alias *';'*, *'if'* alias *':-'*.

  Example 1. "Duke is east of UNC and north of NCState."

  ```
  east(UNC, Duke), north(NCState, Duke).
  ```

  Example 2. "There exists a university which is east of some other university."

  ```
  east(?X, ?Y) :- university(?X), university(?Y).
  ```

Example 3. [Incorrect sentence] "There exists a university which is east of something."

```
east(?X, ?Y) :- university(?X). // Not a sentence due to the
                                // presence of free variable ?Y
```

- **Statement.** A statement is the boolean value (true or false), which a declarative sentence asserts. We use the word "declarative" to denote the order of terms in a sentence does not matter, i.e., if the terms are reordered, the truth value of the statement remains the same. A statement can be made from two different sentences may assert the same. For example:

  Example 1. [Sentence 1] "Duke is east of UNC and north of NCState."

  [Sentence 2] "Duke is north of NCState and east of UNC."

  Two different sentences make the same statement.

```
east(UNC, Duke).
north(NCState, Duke).
university(?X) :- east(UNC, ?X), north(NCState, ?X), ?X = Duke.
```

- **Proof Context.** A set of facts and rules used as input to a prover for a given request.

- **Identity.** An identity is a uniquely distinguishable property of a principal, object, or a group.

- **Principal.** A principal is a unique identity that can make statements on the behalf of real world identities such as a individual, group, organization, or a party.

Note. A principal needs not always correspond to real world identity. For example, a software or a software agent with a keypair can speak on the behalf a real world identity. In SAFE, we do not differentiate between these types. Any entity that speaks with a keypair is called a principal.

- **Object.** An object is a property owned by some principal. The owner provides a local name to the object, which when combined with principal's identity gives a unique name to the object. An object cannot makes statements on its own. Only the owner or some delegated principal authority on the object can make statements on its behalf.

- **Entity.** An entity is either a principal or an object.

- **Resource.** A guarded/protected object by the authorizer that corresponds to a computing or a system component. Example: file access; database lookup; access to a virtual/physical computing node.

- **Group:** A group is a collection of principals who all share the same identity. Example: CS students at Duke; members of ACM.

- **Role:** A role is a collection of operations that can be performed on object by a given principal. A principal may assume a role temporarily for authorization purposes. The operations are specified as attributes or predicates in logic. Sometimes, the operations are also called as actions. Example: database administrator; project investigator.

- **Policy.** A policy is a specification of rules through logic statements for accessing a particular resource. Policies are either defined locally at the authorizer or imported by the authorizer from a trusted anchor.

- **Credential.** A credential is a statement, signed by some principal, asserting some property or attribute about another entity.

- **Authorizer.** An authorizer is the local authority that protects a resource by guarding all accesses to the resource—allowing access only after an appropriate possession of capability is shown. The authorizer determines the access capability by running a local proof based on the requester's credentials and local policy.

- **Requester.** A requester is a principal seeking access to a resource. A requester may pass credentials or other information to identify herself or to shown the possession of capability.

- **Certificate:** A certificate is a transport format for encoding credentials. Examples: X.509 and S-Expressions of SPKI/SDSI standard.

- **Issuer.** The principal that signs the credential granting some attribute for another entity.

- **Speaker.** The principal that makes the statement either directly by signing it or indirectly through an issuer that "speaks for" the speaker.

  Note. For self-signed certificates, issuer and speaker are one and the same.

- **Subject.** The principal for which the credential is issued.

- **Namespace.** Each principal owns and operates a namespace to keep track of resources they own, capabilities they receive, endorsements they issue, or bookmark references to other principal's namespace that may contain relevant trust policies for a later retrieval.

- **Local names.** Local names are the names assigned by the principal to other entities (principals or objects) in their namespace.

- **Revocation.** The retraction of subject's credential by its issuer.
- **Rotation.** The process of replacing a principal's keypair with a new one.
- **Renewal.** The process of replacing an expired certificate with a new one.
- **Credential discovery.** The process of finding the *right* credentials required to prove that requester posses access to the resource.
- **Delegation of authority.** The process of transfer of authority over policy from one principal to another.
- **Compound Principals.** The combination of one or more principals to agree on a some fact.
- **Domain.** The security/trust locality administered by a given authority.
- **Constraint Domain.** A domain over functions where relations among variables is specified as constraints. Example: Linear domains, hierarchical domains, and integer domains.
- **Self-certifying names.** A self-certifying names (IName) uniquely identifies an object in a federated domain by the pair <owning principal, object name >.

  Example: "Alice owns file named wonderland".

  `'Alice:wonderland'`

- **Self-certifying identifiers.** A self-certifying identifier (IID) uniquely identifies an object in a federated domain by the pair <owning principal, object identifier >.

  An object identifier is the hash of object name resulting in a fixed length token.

- **Safe Resource Name.** A Safe Resource Name (SRN) is a compound name that uniquely identifies an object with its fully qualified name starting from some 'root' trust anchor.

  Example: "DNS query: cs.duke.edu".

  ```
  'ICANN-ID:edu'.'edu-ID:duke'.'cs-ID:a'
  ```

- **Authentication.** Authentication is a process of identifying the requesting principal i.e., whether the request is originating from the principal who she claims to be.

- **Authorization.** Authorization is a process of determining whether a given requester possesses the necessary attributes to access a particular resource as mediated by local policy, based on well-defined semantics of policies and credentials.

- **Accountability.** Accountability is the ability to hold a principal, such as a person or organization, responsible for its actions, i.e., whenever the principal violates the policy, then with some non-zero probability, the principal could be punished.

- **Agent.** An agent is a software process or other principal that makes request on the behalf of original principal. Example: browser making requests for the user.

- **Actor.** An actor is a computational entity that can send and receive messages concurrently. Actor model enables asynchronous communication through message passing.

- **Distributed vs Decentralized vs Federated.**

  Distributed: components are located over network. Example: DNS

Decentralized: no central authority. Example: Gnutella, BitTorrent

Federated: a system composed of components from different administrative domains. Example: DNS

Bibliography

[Aba98] Martín Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, January 1998. (Cited on pages 23 and 26.)

[Aba08] Martín Abadi. Variations in access control logic. In *Proceedings of the 9th International Conference on Deontic Logic in Computer Science*, DEON '08, pages 96–109, 2008. (Cited on pages 20 and 29.)

[Aba09] Martín Abadi. Logic in access control (tutorial notes). *Foundations of Security Analysis and Design V*, pages 145–165, 2009. (Cited on page 45.)

[ABLP93] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Language Systems*, 15(4):706–734, September 1993. (Cited on pages 20, 22, 49, and 55.)

[ACMR02] Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman. Conchord: Cooperative sdsi certificate storage and name resolution. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 141–154, London, UK, UK, 2002. Springer-Verlag. (Cited on page 34.)

[AF99] Andrew W. Appel and Ed W. Felten. Proof-carrying authorization. In *6th ACM Conference on Computer and Communications Security*, pages 52–62, 1999. (Cited on pages 18, 31, 32, and 77.)

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communicatons of the ACM*, 53(4):50–58, April 2010. (Cited on page 4.)

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. (Cited on pages 25, 49, 63, and 65.)

[akk15] Akka actor toolkit for jvm. `http://akka.io/`, 2015. (Cited on pages 111 and 122.)

[AL07] Martín Abadi and Boon Thau Loo. Towards a declarative language and system for secure networking. In *Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases*, NETB'07, pages 2:1–2:6, Berkeley, CA, USA, 2007. USENIX Association. (Cited on pages 18, 25, and 28.)

[ant16] Parser generator for reading, processing, executing, or translating structured text or binary files. `http://www.antlr.org/`, 2016. (Cited on page 165.)

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM. (Cited on page 53.)

[AWS15] Amazon web servies (aws). `https://aws.amazon.com/`, 2015. (Cited on pages 45 and 144.)

[Azu15] Microsoft azure: Cloud computing platform and services. `https://azure.microsoft.com/en-us/`, 2015. (Cited on page 144.)

[BAN90] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990. (Cited on pages 19 and 49.)

[Bau03] Ljudevit Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton, NJ, USA, 2003. AAI3107865. (Cited on pages 31 and 32.)

[BCL+14] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds – Part I. (Cited on page 2.)

[BCR+08] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. A user study of policy creation in a flexible access-control system. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 543–552, New York, NY, USA, 2008. ACM. (Cited on pages 32 and 33.)

[BFG10] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, December 2010. (Cited on pages 18, 25, 28, 32, 36, and 63.)

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 164–, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on pages 18, 22, and 32.)

[BGR05] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on pages 32, 33, and 34.)

[BGR07] Lujo Bauer, Scott Garriss, and Micheal K. Reiter. Efficient proving for practical distributed access-control systems. In *Computer Security – ESORICS 2007: 12th European Symposium on Research in Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 19–37, September 2007. (Cited on page 32.)

[BHG14] Ilya Baldin, Shu Huang, and Rajesh Gopidi. A resource delegation framework for software defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 49–54, New York, NY, USA, 2014. ACM. (Cited on page 166.)

[BIK03] Matt Blaze, John Ioannidis, and Angelos D. Keromytis. Experience with the keynote trust management system: Applications and future directions. In *Proceedings of the 1st International Conference on Trust Management*, iTrust'03, pages 284–300, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on pages 18, 22, and 32.)

[BL76] D. Elliott Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA, March 1976. (Cited on page 105.)

[BL08] Ji-Won Byun and Ninghui Li. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal*, 17(4):603–619, July 2008. (Cited on page 31.)

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. (Cited on page 65.)

[BPE+14] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014. (Cited on page 29.)

[BS04] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 139–, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 28.)

[BSF02] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, pages 93–108, August 2002. (Cited on pages 18, 31, 32, and 133.)

[CEE+02] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, February 2002. (Cited on pages 23, 24, 32, and 33.)

[CFL+97] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. Referee: Trust management for web applications. *World Wide Web Journal*, 2(3):127–139, June 1997. (Cited on pages 23 and 34.)

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989. (Cited on pages 25 and 49.)

[CT14a] Jeff Chase and Vamsi Thummala. A Guided Tour of SAFE GENI. Technical Report CS-2014-002, Department of Computer Science, Duke University, June 2014. (Cited on page 144.)

[CT14b] Jeff Chase and Vamsi Thummala. Secure Authorization for Federated Environments (SAFE): Overview and Progress Report. Technical Report CS-2014-003, Department of Computer Science, Duke University, June 2014. (Cited on page 108.)

[DeT02] John DeTreville. Binder, A Logic-Based Security Language. In *IEEE Symposium on Security and Privacy*, pages 105–113. IEEE, May 2002. (Cited on pages 18, 25, 27, 28, and 36.)

[DH06] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, September 2006. (Cited on page 45.)

[EAH+01] Yassir Elley, Anne H. Anderson, Steve Hanna, Sean Mullan, Radia J. Perlman, and Seth Proctor. Building certifications paths: Forward vs. reverse. In *NDSS*. The Internet Society, 2001. (Cited on pages 10 and 67.)

[EFL+99] Carl Ellison, B. Frantz, Butler Lampson, Ronald Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693 (Experimental), September 1999. (Cited on pages 22, 23, 26, and 34.)

[Fag73] Ronald Fagin. *Generalized First-Order Spectra and Polynomial-Time Recognizable Sets'*. PhD thesis, 1973. (Cited on page 53.)

[FCC+03] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM Symposium on Operating System Principles*, October 2003. (Cited on page 165.)

[GCE15] Google cloud platform. `https://cloud.google.com/compute/`, 2015. (Cited on page 144.)

[GEN15] GENI: Global Environment for Network Innovations. `http://www.geni.net`, 2015. (Cited on pages 2 and 144.)

[GFV16] Arpit Gupta, Nick Feamster, and Laurent Vanbever. Authorizing network control at software defined internet exchange points. In *Proceedings of the 2016 Symposium on SDN Research*, SOSR '16, 2016. (Cited on page 166.)

[GJ00a] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *In Proc. 27th ACM Symp. on Principles of Programming Languages (POPL*, pages 316–329. ACM Press, 2000. (Cited on page 27.)

[GJ00b] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Softw. Pract. Exper.*, 30(15):1609–1640, December 2000. (Cited on pages 25, 27, and 32.)

[Gro10] Martin Grohe. From polynomial time queries to graph structure theory. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 2–2, New York, NY, USA, 2010. ACM. (Cited on page 53.)

[GVS+14] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 551–562, New York, NY, USA, 2014. ACM. (Cited on page 166.)

[Gö31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. [Translated to English] Kurt Gödel, "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I," in Jean van Heijenoort, ed., From Frege to Gödel: A Source Book in Mathematical Logic, 1879 - 1931 1967, Harvard University Press, 592-617. (Cited on page 48.)

[HFK+14] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, and Scarfone Cybersecurity. Guide to attribute based access control (ABAC) definition and considerations, 2014. (Cited on page 50.)

[HGL+12] Timothy L. Hinrichs, William C. Garrison, Adam J. Lee, Skip Saunders, and John C. Mitchell. Tba: A hybrid of logic and extensional access control systems. In *Proceedings of the 8th International Conference on Formal Aspects of Security and Trust*, FAST'11, pages 198–213, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on pages 18 and 30.)

[HK00a] Jon Howell and David Kotz. End-to-end authorization. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 11–11, Berkeley, CA, USA, 2000. USENIX Association. (Cited on pages 25 and 42.)

[HK00b] Jon Howell and David Kotz. A formal semantics for SPKI. In *Proceedings of the 6th European Symposium on Research in Computer Security*, ESORICS '00, pages 140–158, London, UK, UK, 2000. Springer-Verlag. (Cited on pages 21, 22, 24, and 26.)

[How00] Jonathan R. Howell. *An examination of keystroke dynamics for continuous user authentication*. PhD thesis, Dartmouth college, 2000. (Cited on page 25.)

[HVdM01] Joseph Y Halpern and Ron Van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1):105–142, 2001. (Cited on pages 23 and 26.)

[Imm82] Neil Immerman. Relational queries computable in polynomial time (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 147–152, New York, NY, USA, 1982. ACM. (Cited on page 53.)

[Imm95] Neil Immerman. Descriptive complexity: a logician's approach to computation. *Notices of the American Mathematical Society*, 42, 1995. (Cited on page 53.)

[Jim01] Trevor Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115. IEEE, May 2001. (Cited on pages 18, 25, 27, 32, 34, 36, and 42.)

[JS01] Trevor Jim and Dan Suciu. Dynamically distributed query evaluation. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 28–39, New York, NY, USA, 2001. ACM. (Cited on page 27.)

[kan09] A declarative applicative logic programming system. `http://kanren.sourceforge.net/`, 2009. (Cited on page 29.)

[KSMK03] Michael Kaminsky, George Savvides, David Mazieres, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 60–73, New York, NY, USA, 2003. ACM. (Cited on pages 31 and 34.)

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992. (Cited on page 20.)

[Lam04] Butler Lampson. Computer security in the real world. *Computer*, 37(6):37–46, June 2004. (Cited on page 2.)

[Lan81] Carl E. Landwehr. Formal models for computer security. *ACM Computiing Surveys*, 13(3):247–278, September 1981. (Cited on page 105.)

[LCG+09] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, November 2009. (Cited on page 28.)

[LGF03] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, February 2003. (Cited on pages 18, 23, 24, and 29.)

[LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*

- *Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 35.)

[LLFS+07] Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and M. Frans Kaashoek. Alpaca: Extensible authorization for distributed services. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 432–444, New York, NY, USA, 2007. ACM. (Cited on pages 18 and 29.)

[LM03] Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73, 2003. (Cited on pages 24, 25, 26, 54, and 62.)

[LM06] Ninghui Li and John C. Mitchell. Understanding SPKI/SDSI using first-order logic. *Int. J. Inf. Secur.*, 5(1):48–64, January 2006. (Cited on page 26.)

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 114–, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 22, 24, 55, 57, and 103.)

[Loo06] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006. (Cited on page 28.)

[LWM01] Ninghui Li, Willam H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *8th ACM conference on Computer and Communications Security*, pages 156–165, 2001. (Cited on page 32.)

[LWM03] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, February 2003. (Cited on pages 27, 32, and 103.)

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. (Cited on page 166.)

[Mar11] Moxie Marlinspike. Ssl and the future of authenticity. `http://blog.thoughtcrime.org/ssl-and-the-future-of-authenticity`, 2011. (Cited on page 5.)

[Mer78] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, April 1978. (Cited on page 45.)

[MKKW99] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 124–139, New York, NY, USA, 1999. ACM. (Cited on page 35.)

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, October 1997. (Cited on page 19.)

[MLM+14] Michelle L. Mazurek, Yuan Liang, William Melicher, Manya Sleeper, Lujo Bauer, Gregory R. Ganger, Nitin Gupta, and Michael K. Reiter. Toward strong, usable access control for shared distributed data. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 89–103, Berkeley, CA, USA, 2014. USENIX Association. (Cited on pages 18 and 30.)

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. (Cited on page 31.)

[PK06] Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 130–145, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on page 29.)

[Rev98] Peter Z. Revesz. Safe query languages for constraint databases. *ACM Transactions on Database Systems*, 23(1):58–99, March 1998. (Cited on pages 54, 62, and 63.)

[Rev02] Peter Revesz. *Introduction to constraint databases*. Springer-Verlag New York, Inc., New York, NY, USA, 2002. (Cited on page 54.)

[RFWS11] Mike Ryan, Ted Faber, John Wroclawski, and Steve Schwab. Attribute-based access control. `http://abac.deterlab.net/`, 2011. (Cited on pages 27 and 162.)

[ria15] Riak key value store. `http://docs.basho.com/riak/`, 2015. (Cited on pages 111 and 131.)

[RL] Ronald L. Rivest and Butler Lampson. Sdsi – a simple distributed security infrastructure. (Cited on pages 22 and 23.)

[saf16] SAFE code repository. `https://github.com/wowmsi/safe`, 2016. (Cited on page 111.)

[sca16] Object-oriented functional language on the JVM. `http://www.scala-lang.org/`, 2016. (Cited on pages 108 and 111.)

[SdR+11] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011. (Cited on page 18.)

[Sha49] Claaude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, October 1949. (Cited on page 45.)

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM. (Cited on page 35.)

[SPJF09] Soner Sevinc, Larry Peterson, Trevor Jim, and Mary Fernández. An emulation of GENI access control. In *Proceedings of the 2Nd Conference on Cyber Security Experimentation and Test*, CSET'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association. (Cited on page 34.)

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computuer Systems*, 2(4):277–288, November 1984. (Cited on page 42.)

[SS94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994. (Cited on pages 25 and 65.)

[SWS11] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information Systems Security*, 14(1):8:1–8:28, June 2011. (Cited on pages 21, 29, and 55.)

[TC15] Vamsi Thummala and Jeff Chase. SAFE: A declarative trust management system with linked credentials. *CoRR*, abs/1510.04629, 2015. (Cited on page 108.)

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990. (Cited on page 63.)

[Ull94] Jeffrey D. Ullman. Assigning an appropriate meaning to database logic with negation. Technical Report 1994-15, Stanford Infolab, 1994. A corrected version of a paper that appeared in "Computers as Our Better Partners" (H. Yamada, Y. Kambayashi, and S. Ohta, eds.) pp. 216–225, World Scientific, Singapore, 1994. (Cited on page 63.)

[Var82] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 137–146, New York, NY, USA, 1982. ACM. (Cited on page 53.)

[WABL93] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the taos operating system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 256–269, New York, NY, USA, 1993. ACM. (Cited on pages 20 and 32.)

[WJL09] Qihua Wang, Hongxia Jin, and Ninghui Li. Usable access control in collaborative environments: Authorization based on people-tagging. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 268–284, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 30.)

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. (Cited on pages xi, 25, and 83.)

[XSB15] XSB: Logic Programming and Deductive Database system (Tabled Prolog) for Unix and Windows. `https://sourceforge.net/projects/xsb/`, 2015. (Cited on page 108.)

[YC04] Aydan R. Yumerefendi and Jeffrey S. Chase. Trust but verify: Accountability for network services. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM. (Cited on page 115.)

[YC07] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong Accountability for Network Storage. *ACM Transactions on Storage (Selected papers from the 2007 Symposium on File and Storage Technologies)*, 3(3), October 2007. (Cited on pages 15 and 128.)

[Zim95] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995. (Cited on page 9.)

Biography

Vamsi Thummala hails from southern India. He received his B.S., Honors (2004) in Computer Science from International Institute of Information Technology (IIIT), Hyderabad; M.S. (2011) and Ph.D. (2016) in Computer Science from Duke University. Prior to arriving at Duke, he worked as a Systems Analyst for a telecommunications company.

His dissertation addresses key research challenges for building secure networked systems: end-to-end authorization, credential discovery, and declarative policy management. His work develops a practical software for managing trust in a collaborative hybrid cloud setting with multiple authorities. His other research contributions include robust query optimization and configuration management for database systems.