Towards Least Privilege Principle: Limiting Unintended Accesses in Software Systems

by

Beng Heng Ng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

Doctoral Committee:

Professor Atul Prakash, Chair
Professor Kang G. Shin
Associate Professor Vineet R. Kamat
Associate Professor Zijiang James Yang

© Beng Heng Ng 2013
All Rights Reserved

For my wife, Haoyi, and daughter, Reann.

ACKNOWLEDGEMENTS

The journey towards writing this thesis had not been an easy one, and I will forever be indebted to the kind people around me for their guidance and support.

This thesis would not have materialize without the unrelenting support, enormous patience, and encouragement of my advisor, Prof. Atul Prakash. He has taught me the importance of rigorous research methodologies, critical thinking, as well as the need to always keep an open mind. His advice was not limited to science, but also included life, especially during one of the most challenging periods in my life.

I am also extremely grateful to Prof. Shin, Prof. Kamat, and Prof. Yang, for their invaluable suggestions, perspectives and insights, which have helped shape this thesis.

I also thank my previous and current colleagues, Billy Lau, Hu Xin, Alex Crowell, Earlence Fernandes, and Ajit Aluri, for the numerous intense and thought-provoking discussions.

I gratefully acknowledge the funding provided by the Government of Singapore, thus allowing me to focus on my research that leads to this thesis.

Above all, I would like to thank my wife, Hao Yi, for putting her dreams on hold so that I can go after mine. I don't think I will ever be able to understand the sacrifices that she has gone through. I also cannot thank my parents and brothers enough for their support. They make my every trip back home worthwhile. And of course, I thank my daughter, Reann, for being the sunshine of my life.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES x

ABSTRACT xii

CHAPTER

I. Introduction 1

1.1 Problem Statement 1
1.2 Terminologies 2
1.3 Why is Access Control Hard? 2
1.3.1 Email Address Leakages 3
1.3.2 System Permission Gaps 4
1.3.3 Software Code Re-Use 4
1.4 Thesis Statement 6
1.5 Contributions 6
1.5.1 Detecting Email Addresses Leakages 6
1.5.2 Detecting and Mitigating Permission Gaps in `SSHD`, `auditd`, and User Groups 6
1.5.3 Detecting Binary Code Re-Use 7
1.6 Thesis Organization 7

II. Literature Review 8

2.1 Access Control Mechanisms 8
2.2 Enforceable Security Policies 9
2.3 Disposable Email Addresses – SEAL 10
2.4 Tightening System Permissions – DeGap 12

2.5 Software Similarity Research – Exposé 13
2.5.1 Syntactic Approaches 14
2.5.2 Semantic Approaches 16
2.5.3 Other Techniques 17

III. Mitigating Impact of Email Address Leakages with SEAL 19

3.1 Introduction 19
3.2 User's Perspective 22
3.2.1 Lifecycle of a Semi-Private Alias 24
3.2.2 Affiliation Validation: Aliases as Proof of Affiliation 27
3.2.3 Requesting an Alias 27
3.3 Architecture 30
3.3.1 Account Creation 31
3.3.2 Alias Request 31
3.3.3 Managing the Alias Lifecycle 32
3.4 Evaluation 33
3.4.1 Effectiveness of Partly Restricting Aliases 34
3.4.2 Affiliation Validation 35
3.4.3 Leakages 37
3.4.4 Timing Performance 41
3.5 Discussion - Security and Usability 42
3.6 Conclusion 47

IV. Reducing System Permission Gaps with DeGap 49

4.1 Introduction 49
4.2 Limitations 53
4.3 Relationship between Permission Gaps, Permission Creep, and Attack Surfaces 54
4.4 Tightening System Permission Gaps 55
4.4.1 Gap Analysis and Traceability 56
4.5 System Architecture 60
4.5.1 Overview 60
4.5.2 Principles 62
4.5.3 Database Model 63
4.5.4 Permission Gap Analyzer 64
4.5.5 DB Schema and Query Mapper 70
4.6 Evaluation 73
4.6.1 Case Study: `SSHD` 73
4.6.2 Case Study: `auditd` 77
4.6.3 Case Study: Tightening `/etc/group` 82
4.7 Improving Log Parser Performance 83

4.8 Conclusions . . . 85

V. Discovering Potential Binary Code Re-Use with Exposé . . . 87

5.1 Introduction . . . 87
5.1.1 Security Implications of Binary Code Re-Use . . . 87
5.1.2 Other Applications of Detecting Code Re-Use . . . 88
5.1.3 Possible Approaches . . . 89
5.2 Assumptions and Scope . . . 91
5.3 Approach . . . 93
5.3.1 Pre-Filtering . . . 94
5.3.2 Computing semantic matches (`IS`-pairs) . . . 96
5.3.3 Syntactic function matching (`MAY`-pairs) . . . 98
5.3.4 Distance Score . . . 101
5.4 Results and Evaluation . . . 104
5.4.1 Quality of Ranking of Applications . . . 104
5.4.2 Library Versions and Compiler Options . . . 108
5.4.3 Timing Performance . . . 110
5.5 Conclusion . . . 111

VI. Conclusions and Future Work . . . 116

6.1 Contributions . . . 116
6.2 Future Work . . . 120

Bibliography . . . 122

LIST OF FIGURES

Figure

2.1 Partial function call graph of a shared library. 15

2.2 Partial function call graph of an executable. 15

3.1 Overview of SEAL. 22

3.2 State diagram for alias. 24

3.3 Lifecycle scenarios of three aliases. 25

3.4 Example email sent by Bob to request an alias. 28

3.5 Example response to Bob's alias request. 28

3.6 SEAL architecture. 30

3.7 Example of using *hint*. 32

3.8 Simplified SEAL database. 32

3.9 Number of emails received daily for the control and subject aliases. 34

3.10 Number of emails processed daily. 36

3.11 Number of active aliases per day. 36

3.12 Histogram of the number of aliases for different number of unique sender domains. 39

3.13 Values of `Received` header fields for an email 43

4.1 Conceptual model for DeGap. 60

4.2 Database model for DeGap. 63

4.3 Algorithm for Config. Evaluator, E. 65

4.4 Greedy Algorithm for Discovering a Maximal Patch. 66

4.5 Configuration specification format and examples for `PermitRootLogin` and `AllowUsers` for `SSHD`. 67

4.6 An example of a query for `auditd`. 70

4.7 General form of a query. 70

4.8 BNF for constraint expression $\mathcal{C}$. 71

4.9 Configuration generation rules used as input to DeGap. 74

4.10 Partial configurations used by `SSHD` for Server 1. 75

4.11 Tightened partial configurations for Server 1. 75

4.12 Partial configurations used by `SSHD` for Server 2. 76

4.13 Tightened partial configurations for Server 2. 76

4.14 Decision trees for determining file role type, i.e. owner, group, or other. 79

4.15 Query used for finding the number of files that have permission gaps for other-write permissions. 80

4.16 Number of files and directories with permissions set and actually used. 81

4.17 Ratio of log sizes to database sizes for `auditd`. 83

5.1 Exposé overview. 93

5.2 Cumulative distribution of function sizes. 97

5.3 Cumulative distribution of function cyclomatic complexities. 98

5.4 Function grouping, given a set of matching pairs. 104

5.5 Cumulative distribution of distance scores. 107

5.6 Cumulative distribution of elapsed times. 111

6.1 Summary of approaches towards detecting and mitigating unintended accesses. 119

LIST OF TABLES

Table

1.1 Summary of permission-related entities for this thesis. 5

2.1 Sub-categories in software similarity research. 14

3.1 Commands used by SEAL. 25

3.2 Capability matrix between sender status and alias states. 25

3.3 Table of aliases used for website, mailing list and newsletter registrations, sorted in increasing number of unique sender domains. 38

3.4 Table of aliases used for classified advertising and forum postings, sorted in increasing number of unique sender domains. 40

3.5 Percentages for five groups of shortest delays. 42

4.1 Comparison between number of entries in log file and number of tuples in database for `SSHD`. 77

4.2 Comparison between average number of entries in log files and average number of tuples in databases for `auditd`. 83

4.3 Timings for various benchmarks when `auditd` is activated, and when the modified kernel for improving `auditd` is used. 84

5.1 Percentage of binaries without symbols in various Linux distributions. 92

5.2 List of common functions excluded. 95

5.3 Typical x86 function prologue and two possible epilogues. 101

5.4 10 smallest distance scores using `libpng` as test library. 105

5.5 15 smallest distance scores using `zlib` as test library. 106

5.6 Distance scores of applications compiled with different `zlib` versions compared with `zlib v1.2.3`. 108

5.7 Distance scores between the 11 applications with smallest distance scores and different `zlib` versions. 109

5.8 Number of tuples for the top 11 binaries. 110

ABSTRACT

Towards Least Privilege Principle:
Limiting Unintended Accesses in Software Systems

by

Beng Heng Ng

Chair: Atul Prakash

Adhering to the least privilege principle involves ensuring that only legitimate subjects have access rights to objects. Sometimes, this is hard because of permission irrevocability, changing security requirements, infeasibility of access control mechanisms, and permission creeps. If subjects turn rogue, the accesses can be abused. This thesis examines three scenarios where accesses are commonly abused and lead to security issues, and proposes three systems, SEAL, DeGap, and Exposé, to detect and, where practical, eliminate unintended accesses.

Firstly, we examine abuse of email addresses, whose leakages are irreversible. Also, users can only hope that businesses requiring their email addresses for validating affiliations do not misuse them. SEAL uses semi-private aliases, which permits gradual and selective controls while providing privacy for affiliation validations.

Secondly, access control mechanisms may be ineffective as subject roles change and administrative oversights lead to permission gaps, which should be removed expeditiously. Identifying permission gaps can be hard since another reference point besides

granted permissions is often unavailable. DeGap uses access logs to estimate the gaps while using a common logic for various system services. DeGap also recommends configuration changes towards reducing the gaps.

Lastly, unintended software code re-use can lead to intellectual property theft and license violations. Determining whether an application uses a library can be difficult. Compiler optimizations, function inlining, and lack of symbols make using syntactic methods a challenge, while pure semantic analysis is slow. Given a library and a set of applications, Exposé combines syntactic and semantic analysis to efficiently help identify applications that re-use the library.

CHAPTER I

Introduction

1.1 Problem Statement

As early as 1975, Saltzer and Schroeder noted that every program and every user of a system should operate using the least set of privileges necessary to complete the job [107]. Strict adherence to the *least privilege principle* prevents security issues such as privilege creep [3, 121]. Unfortunately, violations of the principle are all too common [83, 87], with the common reasons being (i) security unawareness, (ii) hedging privileges for future usability, and (iii) insecure defaults. Using manual analysis, Felt et al. found that 15% to 30% of Chrome extensions are over-privileged, while the same is true for 10% to 20% of the most popular Android applications [42]. They also note that permission systems are effective in protecting against vulnerabilities in benign-but-buggy applications. However, users are required to make decisions about permissions on almost every download. This can quickly lead to security fatigue when security is perceived as a barrier rather than an enabler [43]. In their study on Windows users, Motiee et al. found 69% of the participants did not apply the user account control (UAC) approach correctly, all used administrator accounts, and 91% were unaware of the need to adhere to the least privilege principle [87]. Thus, checking for violation of the principle plays an important role in enhancing the security of a system.

1.2 Terminologies

In various literatures, the distinction between a *privilege* and a *permission* often depends on the context. For example, in the hardware domain, the term "privilege" is preferred over "permission". In some literatures, such as a manual for an Operating System [9], a privilege overrides a permission in the event of a conflict. In this thesis, I will use the two terms interchangeably.

In Lampson's seminal work on protection [70], he described a permission as an access *right* granted to a *subject* to access an *object*. Thus, a permission is represented by a tuple (s, o, r) that denotes subject s as being authorized to access object o using access right r. We will use the same terminologies in this thesis.

1.3 Why is Access Control Hard?

Administrating access control is an iterative process of granting and reviewing; the administrator creates an access control policy to grant the necessary permissions, and reviews the policy over time. However, in practice, this process can be fraught with challenges, which I will list before providing examples and motivating three works, SEAL, DeGap, and Exposé, for this thesis.

Irrevocability The object owner may not have the ability to revoke a granted permission. In other words, the effect of granting the permission is irreversible.

Changing Security Requirements As the saying goes, "Security is a process, not a product"[1]. The security requirements of a system may change. While this is typically not a cause for concern on its own, when coupled with irrevocability, the security implications can become harder to manage.

Lack of Access Control Mechanisms An access control mechanism may be lack-

[1] The saying is often attributed to Bruce Schneier of Counterpane Security Systems.

ing or impractical. This can lead to access rights being acquired, sometimes implicitly and without an object owner's knowledge.

Permission Creep Permission creep occurs when a subject's role changes, but the permissions associated with the previous role are not revoked [121].

1.3.1 Email Address Leakages

The Spamhaus Project characterizes the traditional problem of email spam as an issue of consent rather than content [122]. With current email addresses, once an address leaks without its owner's consent, it becomes effectively compromised, creating a struggle for the user to keep their addresses out of the hands of new spammers. This is because the senders' knowledge of email addresses is sufficient for the senders to email the owners of the addresses, i.e., knowledge of the email addresses cannot be revoked. Besides, technically, the only form of access control exists between the owner and first degree contacts. Beyond that, owners have no means to control who may acquire knowledge of the addresses.

While advances in technology have seen successes in filtering spam, most of these techniques do not solve the root of the problem: once spammers have knowledge of the victim's email address, and can continue to spam the victim, or propagate the address to other malicious senders. Furthermore, according to industry reports, while the volume of spam has decreased, personalized spam is on the rise [64, 129], and current state-of-the-art anti-spam technologies, which rely on the bulkiness property of spam, may soon become insufficient. I will elaborate on SEAL, a method for distributing semi-private email aliases to senders in Chapter III. Aliases are assumed to eventually be leaked to spammers. The method allows a new alias to be created while the leaked alias is rendered useless.

1.3.2 System Permission Gaps

System objects are guarded by permissions that serve as the first line of defense. Ideally, only permissions that will be used should be granted. Unfortunately, this is often hard to achieve because information about which permissions will be needed is often unavailable. System administrators often tweak the permissions using out-of-the-box permissions as baseline. However, due to the large number of objects, it is tedious to configure every permission, such as whether all files on the system should be world-writable. Without security policies to state the permissions needed for every object on a system, there are no reference points for evaluating if all the security requirements for the system are met. Thus, the desired security requirements may differ from that actually configured for the system, leading to permission gaps.

Permission gaps can also occur on host machines when the roles of users change but the permissions granted for their previous roles are not revoked [96]. Over time, the extraneous permissions granted accumulate and result in permission creeps.

In Chapter IV, I will elaborate on DeGap, a tool for evaluating the permission gaps for system services. DeGap estimates a reference point for computing the permission gaps based on system logs.

1.3.3 Software Code Re-Use

Software code re-use is a common practice that allows developers to generate applications under time and knowledge constraints. However, with the lack of access control mechanisms, disparities between the author's intention on how a piece of code should be used and how it is actually being used can occur. Permissions to use a piece of code are commonly granted using software licenses. Often, this requires subjects to conform to the conditions specified in the license agreements. However, enforcing these conformances can be hard. Once a developer has physical access to a piece of code, it is hard to police its subsequent uses. An aggravating cause of

Table 1.1: Summary of permission-related entities for this thesis.

Tool	Subjects	Rights	Objects	Object Owner
SEAL	Unauthorized senders, e.g., spammers	Send email	Email addresses	Email address owner
DeGap	Processes, Users	Read, Write, Execute, Connect	Files, `SSHD`, User groups	System administrator
Exposé	Applications	Re-use	Software library	Code author

failure to conform to the conditions is changing security requirements. For example, when a piece of code is acquired by another company, its software license may be updated with new conditions. Thus, even when the security requirements change, the developer's ability to use the original code remains unaltered.

Security implications resulting from software code re-use include intellectual property theft. An application developer may re-use a piece of code without conforming (either knowingly or unknowingly) to the conditions specified by the software license, resulting in violation of the licenses, or worse still, intellectual property theft.

Existing efforts to solve the problem of detecting code re-use has largely assumed the availability of source code. Little work has been done to solve the problem for binaries compiled from the source code. Analyzing the binary code is important as it is the authoritative source of information for a software library or application [104]. However, the set of challenges between analyzing source code and binaries differs. When analyzing binaries, compiler optimizations, function inlining, and lack of symbols in binary code make the task challenging for automated techniques. On the other hand, semantic analysis techniques are relatively slow. In Chapter V, I will discuss Exposé, a tool that takes a library and a set of binary applications as inputs. Exposé combines symbolic execution using a theorem prover and function-level syntactic matching techniques to achieve both performance and order applications according to the likelihood of re-using the library's code.

Table 1.1 summarizes the permission-related entities for the three different levels

that will be discussed in this thesis.

1.4 Thesis Statement

Because access rights can be abused in email address distribution, system services, and software code re-use, thus leading to security implications such as email spam, increased attack surface area, and intellectual property theft, methods to detect potential unintended accesses, and thereby checking for violations of the least privilege principle, are needed to improve security.

1.5 Contributions

1.5.1 Detecting Email Addresses Leakages

This thesis describes the design, implementation, and evaluation of SEAL [91], a work on *semi-private aliases*. SEAL involves a mechanism that uses a life-cycle model for permitting gradual, selective controls on the use of an email alias by senders without requiring special infrastructure on the part of senders or receivers. Semi-private aliases can also be used to authenticate a user belonging to a certain organization and reveal only selected attributes to a service while hiding the real identity.

1.5.2 Detecting and Mitigating Permission Gaps in `SSHD`, `auditd`, and User Groups

DeGap is a framework for detecting permission gaps for system services. DeGap uses system logs to identify the permissions that are requested. Using that as an approximation, it computes the permission gaps. This thesis discusses DeGap's implementation and the logic for analyzing permission gaps, as well as the algorithm for proposing possible changes to the configurations towards minimizing permission gaps. We also present our evaluation in using DeGap to support three services, `SSHD`,

`auditd`, and user groups.

1.5.3 Detecting Binary Code Re-Use

Our initial study on code re-use for two libraries found that some of the re-used codes date back to more than 10 years ago [92]. Building on these results, this thesis examines Exposé [93], a tool that combines symbolic execution using a theorem prover and function-level syntactic matching techniques to achieve both performance and high quality rankings of applications. We discuss the algorithm used as well as our findings, including the performance and quality of the rankings for analyzing two libraries. We also present the effects of different library versions and compiler optimizations on the rankings.

1.6 Thesis Organization

Chapter II reviews the existing work on least privilege principle, access control mechanisms, enforceable security policies, disposable email addresses, tightening system permissions, and software similarity research. Chapter III elaborates on a method to limit the impact of email address leakages. Chapter IV examines a framework for discovering permission gaps at the system service level. Chapter V highlights the problem of illegitimate software code re-use and presents techniques for detecting binary code re-use while considering the effects of function in-lining, compiler optimizations, and lack of symbol information. Chapter VI concludes the thesis and suggests possible future work.

CHAPTER II

Literature Review

This Chapter reviews existing works in access control mechanisms, enforceable security policies, disposable email addresses, tightening system permissions, and software similarity research.

2.1 Access Control Mechanisms

Access control mechanisms are often used to ensure appropriate access rights, which are determined by some security policies, are granted to the subjects. Access control has been well-studied and several access control models have been proposed, with Role-Based Access Control (RBAC), Discretionary Access Control (DAC), and Mandatory Access Control (MAC) being the common models investigated [99, 61, 71, 97, 95]. The primary objective for access control is to restrict user activities, and is typically enforced with a reference monitor that mediates every attempted access by a subject to an object in the system [109, 108]. To deal with changing user roles, Bertino et al. propose Temporal Role-Based Access Control (TRBAC) that allows the access control policy to take effect for specific durations [27]. TRBAC requires information about the durations for granting a subject access to an object. However, this information is not always available. For example, it may not be clear when a new employer will relinquish her role.

2.2 Enforceable Security Policies

A great deal of work has been done in enforcing security policies and thus abiding the least privilege principle. However, most of the work is performed at the architectural level. For example, Levin et al. propose extending the separation kernel abstraction, such as those used in real-time embedded systems and virtual machine monitors, to represent the enforcement of the least privilege principle [72]. Schneider note that enforcing application-level security policies is a difficult problem in both theory and practice [112]. In particular, he highlights two questions that need to be addressed. Firstly, how can a reference monitor that intercepts all program actions and blocks those that cause compromise be implemented? Secondly, how can one define a suitable policy for the reference monitor to enforce?

Buyens et al. argue that the least privilege principle in software architecture is often neglected due to the lack of systematic rules as well as guidance to explain how the principle should be applied in practice [31]. They assume the worst case permission assignments for a given set of use cases and propose architectural changes to reduce violations of the principle. Scandariato et al. extend the work by proposing a two-phase approach, preparation and analysis, for automatically detecting least privilege violations in software architectures [110]. Their work assumes that proper documentation exists, e.g., in UML, for the software architecture being analyzed. While the assumption is expected to hold in a well-documented software architecture, this may not be the case for other scenarios, such as those discussed in this thesis.

A corpus of work examines enforceable security policies from a theoretical perspective [40, 113, 22, 74]. Enforceable security policies can be enforced by monitoring system execution and modifying the execution if policies are violated. This is achieved through the use of monitors, such as firewalls and security kernels (e.g. SELinux [120]), and a set of automata that describe the possible transformations to the execution. While enforceable policies play an important role in ensuring that

executions conform to the policies during runtime, our work focuses on checking for potentially abusable accesses.

2.3 Disposable Email Addresses – SEAL

Towards limiting the impact of email address leakages, disposable email aliases (DEA) have been suggested. Variants of DEAs are supported by several systems. We divide current DEA solutions into two groups, characterizing them as either *incomplete* or *overly restrictive*. The first category of DEA systems comprises specialized services that allow users to create DEAs but do not provide full email services. They can be sub-categorized into receive-only systems that do not allow a user to reply to emails, and temporary systems that only allow users to access their inbox for a limited amount of time [4]. Others, e.g., Mailinator [7, 6], provide a single shared inbox for all users, and so there is no notion of privacy; anyone with the email ID string can access any email belonging to that email ID.

In the second category, the DEA systems are overly restrictive, either only permitting the complete removal of an address to prevent spam or requiring that every correspondent solve a CAPTCHA. One example of such work is Inexpensive Email Addresses (IEA) [137]. IEA cryptographically generates exclusive email addresses for each sender that must be verified by CAPTCHA. This greatly limits the system's practicality, making it difficult to use with automated systems like mailing lists, newsletters, and password recovery services. For normal users, it requires them to go through an extra step of solving a CAPTCHA before being able to send an email. Yahoo Mail's aliases require removal of an alias to prevent spam, once traditional filters break down.

Ioannidis propose the concept of a Single-Purpose Address (SPA) [56], where an SPA has cryptographic properties and encodes security policies that can be enforced by receivers into the email address itself. When a receiver creates an SPA, an ex-

piration date is supplied that determines its lifetime. This encoding of policy into SPAs severely limits their usefulness; an encoded policy can never be changed during the life of the email address, so a single compromise means the owner must live with spam until the address expires or switch to a new key, thereby invalidating all of his existing SPAs. The unlikely event of a server compromise also poses a much greater problem for SPAs, since the leak of a key requires the owner to invalidate all of their SPAs and start from scratch. In our work on SEAL, we avoid these limitations by keeping state on the server, allowing the user to flexibly respond to address leaks by restricting specific aliases. A side benefit of this is that in the event of a temporary compromise of the server itself, once control is regained the user can restrict all of their preexisting aliases, avoiding a temporary complete loss of service due to the need to create and distribute entirely new email addresses.

The Tagged Message Delivery Agent (TMDA) is a challenge/response system that aims to mitigate spam [126]. One feature of TMDA is tagged addresses that can contain date information used in a similar manner to SPA for determining the expiration of the addresses. Similar to SPA, the expiration date has to be determined at the point of creation. Again, the tagged address may be leaked before it expires.

The free online classified advertising site Craigslist generates a random anonymous email address for the user when a posting is made. While this conceals the user's real address from email harvesters that scrape websites, Craigslist is often attacked by spammers who post fake advertisements. An unknowing user who replies to the fake email reveals his real address. With SEAL, a user can simply use a semi-private alias when responding to an ad on Craigslist, thus retaining an ability to block out spam to that address at any point.

Open ID [103] permits users to sign up for external services using an existing email ID, such as their Facebook ID or Google ID, while providing some privacy controls. SEAL accomplishes a similar goal but without requiring the service provider to use

a specific authentication infrastructure and having full control over the information that is disclosed with the email alias.

Spam filtering has been well-studied [105, 139, 18, 19, 114, 84, 94, 138, 101, 47, 28, 118, 48, 66, 14], and is complementary to our approach. While SEAL is not a spam filter, the life-cycle management controls provide an additional layer of spam control when traditional spam filtering fails, without requiring an alias to be completely disabled.

2.4 Tightening System Permissions – DeGap

The concept of permission gaps has been discussed previously in works on permission gaps for Android applications by Felt et al. [41], Au et al. [20], and Bartel et al. [21]. The phrase "over-provisioning of permissions" has also been used to describe a similar notion [63].

The key distinction between DeGap and these works is that in these works, the application is considered to be the subject, the set of granted permissions is considered to be the set of authorizations requested *from* the user when the application is installed and the set of used permissions is considered to be an upper bound on the set of permissions that could be used by the application, based on static analysis of the code (which tends to be exhaustive). DeGap is looking at systems in a different scenario where the set of used permissions is not computable by static analysis of the system. The DeGap idea could still be useful in the context of Android applications. It is possible that an application's code could make use of more restricted permissions than it does. In that case, log analysis with DeGap may help a developer identify opportunities for tightening the permissions used within the code.

DeGap is not an access control mechanism; it complements existing access control mechanisms by providing a means to verify the correctness of access control policies. However, a brief review of existing access control mechanisms may be useful. Access

control models, such as Role-Based Access Control (RBAC), Discretionary Access Control (DAC), and Mandatory Access Control (MAC), have been proposed. Access control is typically enforced with a reference monitor that mediates every access by a user to an object in the system [109, 108]. In practice, perhaps SELinux [120] and AppArmor [23] are the most widely accepted implementations of MAC. Both systems provide monotonic security, which is also a goal of DeGap. However, these mechanisms adopt a different philosophy from DeGap; SELinux and AppArmor begin by restricting accesses to objects and relaxing the limits when needed, while DeGap aims to identify accesses that are no longer required and then restricting these accesses.

Other research efforts to improve the permissions of a system typically resulted in integrated tools that perform functions beyond permission checking, such as vulnerability analysis. COPS, a work by Farmer et al., contains tools called `dir.check` and `file.chk` that respectively check a list of directories and files for world-writability [36]. The Tiger Analytical Research Assistant (TARA) improved on COPS by including more analysis scripts [2]. Bastille Linux is a tool that guides the user through hardening the operating system [1]. While all of them may use different means, they return a list of files and directories with world-readable or writable permissions. This is also achievable using the `find` command on Linux-based systems [11]. We argue that the challenge is not in finding such a list of directories and files, since this is likely to be a huge list. Instead, the fundamental problem that we are solving with DeGap is in determining if the permissions are indeed unused.

2.5 Software Similarity Research – Exposé

The work on software code re-use falls under the category of software similarity research. Collberg and Nagra sub-categorized this field based on four applications: software birthmarking, software forensics, plagiarism detection, and clone detection [34]. We summarize the objectives of the four applications in Table 2.1. The objective of

Application	Inputs	Objective
Birthmark Detection	P, Q	Similarity between properties of P and Q that are invariant to common semantics-preserving transformations.
Software Forensics	R	Ordered list of possible authors for R.
Plagiarism Detection	S, T	Similarity between properties of S and T.
Clone Detection	U	Duplicated codes in U.

Table 2.1: Sub-categories in software similarity research.

our work on Exposé is most similar to that of birthmark detection.

2.5.1 Syntactic Approaches

2.5.1.1 n-gram

Many of the works in birthmark detection are designed for interpreted languages, typically Java (when compiled as Java bytecodes) [75, 55, 124, 123, 115]. There are other works that, while not specific to Java bytecodes, use them for evaluation, such as the work by Myles et al. [89]. The number of user-configurable options affecting the output bytecodes during compilation of Java source code is limited. For example, the original Java compiler `javac` does not provide any option that affects how the bytecodes may be generated. This implies that given the Java source code, compilations undertaken by different developers would likely yield highly similar Java bytecodes. In contrast, compilers for compiled languages such as C/C++ offer users different options, such as optimization levels and function inlining threshold, which introduce variations in the output native code. Since Exposé models the sequence of opcodes using n-gram, the variations can give rise to both false positives and false negatives. The effect of false negatives is generally smoothened out by true positives. False positives tend to be more problematic, which we mitigate using semantic analysis.

Outside the domain of birthmark detection, several proposals on using n-gram

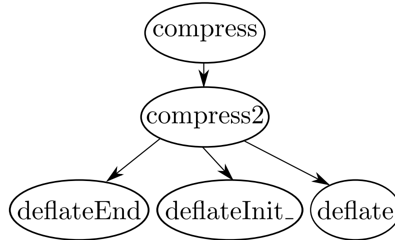


Figure 2.1: Partial function call graph of a shared library.

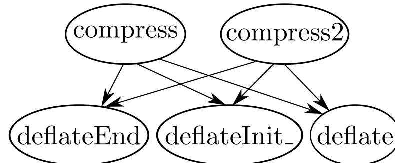


Figure 2.2: Partial function call graph of an executable statically linked with the shared library showing function inlining.

for malware detection and source code plagiarism have been proposed previously [73, 60, 15, 54, 59, 130, 111]. These works generate file signatures using n-gram, which are used for classification. Exposé differs from these works in that we apply n-gram at the function level, which provides us more precision without suffering from the consequences of basic block re-ordering if we had used the basic block abstraction level. To the best of our knowledge, we believe our work is the first to perform in-depth analysis of applying n-gram at the function level for solving the code re-use problem in binaries.

2.5.1.2 Approximate Graph Matching

Another approach is to identify syntactically similar binaries by using approximate graph matching on the function call graphs. Hu et al. propose SMIT, a scalable malware database management system for determining if a new malware is similar to one that is previously observed [52]. Hu et al. modified the Munkres algorithm under the assumption that if two nodes match, then their neighbors are also likely to match. However, if this assumption fails, such as when function inlining occurs,

the error may propagate to all neighbors, thus misleading the Munkres algorithm to produce an incorrectly biased set of assignments. Function inlining typically happens as part of the compiler's optimization process, which eliminates function calls, thereby improving runtime performance. Figure 2.1 shows the function call graph of a library, while Figure 2.2 shows the graph for an application that is statically linked with the same library. In Figure 2.1, `compress` calls `compress2`, which in turn calls three other functions. However after being statically linked with an application, `compress2` is inlined into `compress`. Exposé attempts to mitigate the effects of function inlining by allowing a library function to match with both another application function and its caller.

2.5.2 Semantic Approaches

2.5.2.1 API-based

In addition to n-gram, other possible birthmarks include API sequence, call structures, and frequencies. Choi et al. propose extracting a static birthmark based on the API call structure [33] while Tamada et al. extract the API information dynamically [125]. Wang et al. propose using System Call Dependency Graph (SCDG), also extracted dynamically, as birthmark [132]. The use of API and system call information is useful if such information is available and unique. However, for our purpose, this information may not necessarily be available. The symbols may be stripped and thus possibly removing the function names. Also, for self-contained executables that do not make system calls, such as compression libraries, these features cannot be applied.

2.5.2.2 Symbolic Execution

`BinHunt`, a work by Gao et al., attempts to find semantic differences between binaries using symbolic execution and theorem proving [45]. The authors propose

finding semantic differences instead of syntactic differences. They argue that semantic differences are more suitable for reflecting changes in program functionalities. They attempt to compute the matching strength between basic blocks within two functions across two binaries using symbolic execution to extract a set of constraints that are then submitted to the theorem prover. A backtracking algorithm, guided by the matching strength, is then applied to identify similarities between the control flow graphs of the functions. In their first case study, it required about an hour to complete the examination of several basic blocks. The authors explained that the long computation time is due to syntactic differences between the functions. In their second case study, analyzing two similar binaries of about 41,000 instructions took approximately 30 minutes. While `BinHunt` is able to identify semantic differences with high accuracy, it is less practical for analyzing huge number of files. Exposé aims to quickly rank a set of executables with high true positive rate. `BinHunt` may then be applied on the smaller set of files to determine if they are semantically identical with a vulnerable library file.

2.5.3 Other Techniques

It may seem immediate that the state-of-the-art techniques for identifying obfuscated malware could be trivially applied for our problem domain. We first examine static analysis approaches to malware detection. Moser et al. show that static analysis, which encompasses syntactic analysis, is ineffective in identifying obfuscated malware and propose combining static with dynamic analysis [86]. Malware static analysis generally identifies unique syntaxes such as section names and the legitimacy of the entry point for the program for hints. Other hints include entry point in the thread local storage section, shared libraries with no exported functions, low import function count, and modified import function table [127]. These techniques were effective until malware authors removed these hints. In our context, the absence of

such hints renders the techniques ineffective.

Recent works in identifying obfuscated malware have focused primarily on semantic analysis [17, 102]. While we also use semantic analysis as a building block, it is not entirely sufficient. One subtle difference between malware analysis and Exposé is that the former identifies equivalences between binaries while the latter focuses on discovering code re-use. In malware analysis, it is typically assumed that two functionally equivalent binaries do not contain extraneous functions so as to minimize file size. The function call graph usually remains mostly similar and has a smaller order. On the other hand, in addition to library functions, applications mostly comprise other functions, thus introducing substantial noise. Functions in the library may not be linked into the application if they are not called by the application, resulting in modified function call graphs. These interferences make our work an interesting and a challenging one. A second subtle difference is that semantic analysis may fail due the complexity of the functions and does not scale well if a large number of functions in a library need to be compared with functions in a large number of applications.

CHAPTER III

Mitigating Impact of Email Address Leakages with SEAL

3.1 Introduction

This Chapter discusses rogue accesses to email addresses (objects) by senders (subjects), and methods to detect and mitigate email address leakages.

Since its inception in the 1970s, electronic mail, or email, has come to largely replace snail mail for a variety of reasons, including its low cost, user convenience, ease of use, and high delivery efficiency. But for these very same reasons the problem of unsolicited bulk email messages, commonly referred to as spam, has grown along with email since the 1990s, spurring huge research efforts for finding tools to combat spam. Although the state-of-the-art in spam detection has been successful at detecting most obvious cases of spam, false positives and false negatives are still not entirely uncommon.

One primary cause of the spam problem is the way in which email addresses are typically used. In order to maintain a fixed address at which one can be reached, the vast majority of email addresses are treated as permanent by their owners and are only abandoned in very rare circumstances. As a result, once a user's email address leaks to spammers, it is nearly impossible to entirely prevent them from sending messages

to the user's inbox. And although one may take extreme care to prevent one's email address from falling in to the wrong hands, because anyone with the address could then leak it to a third party, either intentionally or unintentionally, such an effort can easily be futile. The situation becomes further complicated if we consider that users may later change their mind about whether they should have given their email address to a certain party.

To compound the problem, some services require organization email addresses as part of the proof of a user's affiliation with the organization before deeming the user as eligible for certain services or discounts. Early examples included Facebook, which required university affiliations when it started. More recently, piazza.com is used by many universities to host threaded forums between students and professors and, by default, requires users to sign in with an email ID that authenticates them to their university. This creates a potential dilemma for professors as to whether it is proper to require students to sign up for an external service with a different privacy policy than their university's. Companies offering corporate discounts on their services, for example, Sprint and AT&T in the U.S., also require that customers provide their work-affiliated email ID to receive discounts on their monthly bills. While alternative proof methods may be allowed, these are usually troublesome. This also creates concerns: a work address is potentially being used to receive non-work email, making it more susceptible to marketing use.

This Chapter describes a mechanism called *semi-private aliases*, a novel solution that attempts to blend the user control provided by disposable email addresses with the flexible nature of ubiquitous permanent email addresses to provide an email aliasing mechanism that can limit misuse without being overly restrictive to either the address owners or their trusted correspondents. Semi-private aliases are email addresses that can be attached seamlessly to a user's regular inbox, and serve as aliases for that inbox which can be distributed in the same fashion as Disposable Email

Addresses (DEAs) [81, 117]. These aliases make two significant contributions:

1. *Concept of Alias Lifecycle:* A *lifecycle*, in the form of a state machine, allows the user to adaptively add restrictions to an alias as the effects of an address leak begin to show themselves. Starting out as *unrestricted* and open to all incoming mail, an alias can be marked *partly restricted* when unsolicited email begins to be received, resulting in no added restrictions for those who have corresponded with the alias before the identified compromise. New senders sending to a partly restricted alias receive a CAPTCHA challenge, after which the user is prompted to accept or reject that sender's correspondence. Once an alias has reached the point where the user does not expect any new contacts on it, they may mark it as *fully restricted*, at which time only those contacts on the alias' whitelist are permitted to send. Finally, a user may *disable* an alias if it eventually falls entirely out of use.

2. *Affiliation Validation:* When deployed by an organization, the service can be used by individuals in the organization to validate their organizational affiliation (and, optionally, selected other information, such as their name or role) to external service providers without the risk of exposing private information to the providers by simply providing an email alias.

A challenge in designing a system for semi-private aliases, called SEAL, is how to make it work with existing email infrastructure and services as well as not put significant authentication steps (e.g., CAPTCHAs) along desired communication paths. SEAL achieves these goals.

We note that SEAL primarily aims to give user selective control over their privacy rather than provide complete anonymity over the web. We assume that the SEAL infrastructure is trusted, but we attempt to design it so that the theft of information in the databases maintained by SEAL for its functioning is of limited use to spammers.

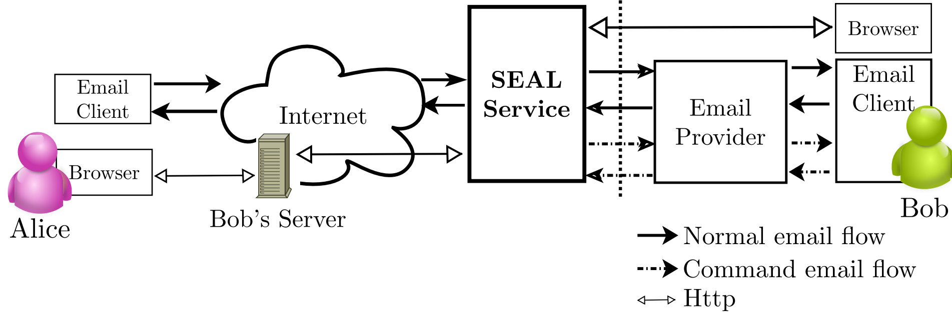


Figure 3.1: Overview of SEAL.

The Chapter is structured as follows. We first present SEAL from an end-user's perspective in Section 3.2. We also discuss the design and describe our prototype in Section 3.3. Next, we evaluate the effectiveness of the system in restricting aliases and tracing alias leakages, and the deployment of our system in a real world scenario in Section 3.4. Finally, we discuss potential limitations and defenses against potential attacks on the SEAL design in Section 3.5 before concluding in Section 3.6.

3.2 User's Perspective

Senders correspond with users of SEAL using semi-private email aliases. An example of such an alias would be `bob.89dtzx3r@sealserver`, where `bob` is the *alias name* and `89dtzx3r` is the *randomization string*. An alias is formed by joining the alias name and the randomization string with a delimiting character in between. The alias name is specified by the user while the randomization string is a randomly generated string, created by SEAL.

Figure 3.1 shows an overview of the email interactions between a SEAL user and a sender who wishes to correspond with that user. A user of the SEAL service sends mail through a SEAL server which processes the mail and forwards it to the recipient. A person sending to a SEAL user addresses their mail to the users semi-private alias, and the SEAL service performs any necessary restrictions, after which the mail can

be forwarded to the user at their normal inbox. Users can also manage their aliases directly over a provided web interface.

To become a user of SEAL, one creates an account with an email provider and configures the account to relay emails through SEAL. Theoretically, the system could also play the role of an email provider. However, segregating the roles has two practical advantages.

Reduced Attack Surface Leveraging existing email providers allows SEAL to obviate the need to provide message storage. This reduces the attack surface of SEAL and also insulates the user's emails from theft or corruption in the event of an attack.

User Familiarity Most users are already familiar with the user interfaces of their current email providers. Many email providers also provide other useful features. Using existing email providers eliminates the need for users to learn a new interface and allows them to continue using their favorite features.

To achieve this, we require the mail provider to support sending mails over authenticated SMTP. This is necessary to prevent masquerading attacks on SEAL.

In addition to normal emails, the user can also send command emails to two Service Addresses that allow the user to provide instructions to SEAL. Users can also receive feedback on the commands from these Service Addresses. Table 3.1 lists the commands supported. When an email is received at a Service Address, only the Subject line is parsed for commands. Table 3.2 summarizes whether a sender is allowed to send email to a semi-private alias for each of the different alias states. Figure 3.2 shows the state transitions of an alias.

3.2.1 Lifecycle of a Semi-Private Alias

After creating a user account, the user can request an alias name that is not in use by other users. Using the alias name, the user can request aliases for distribution

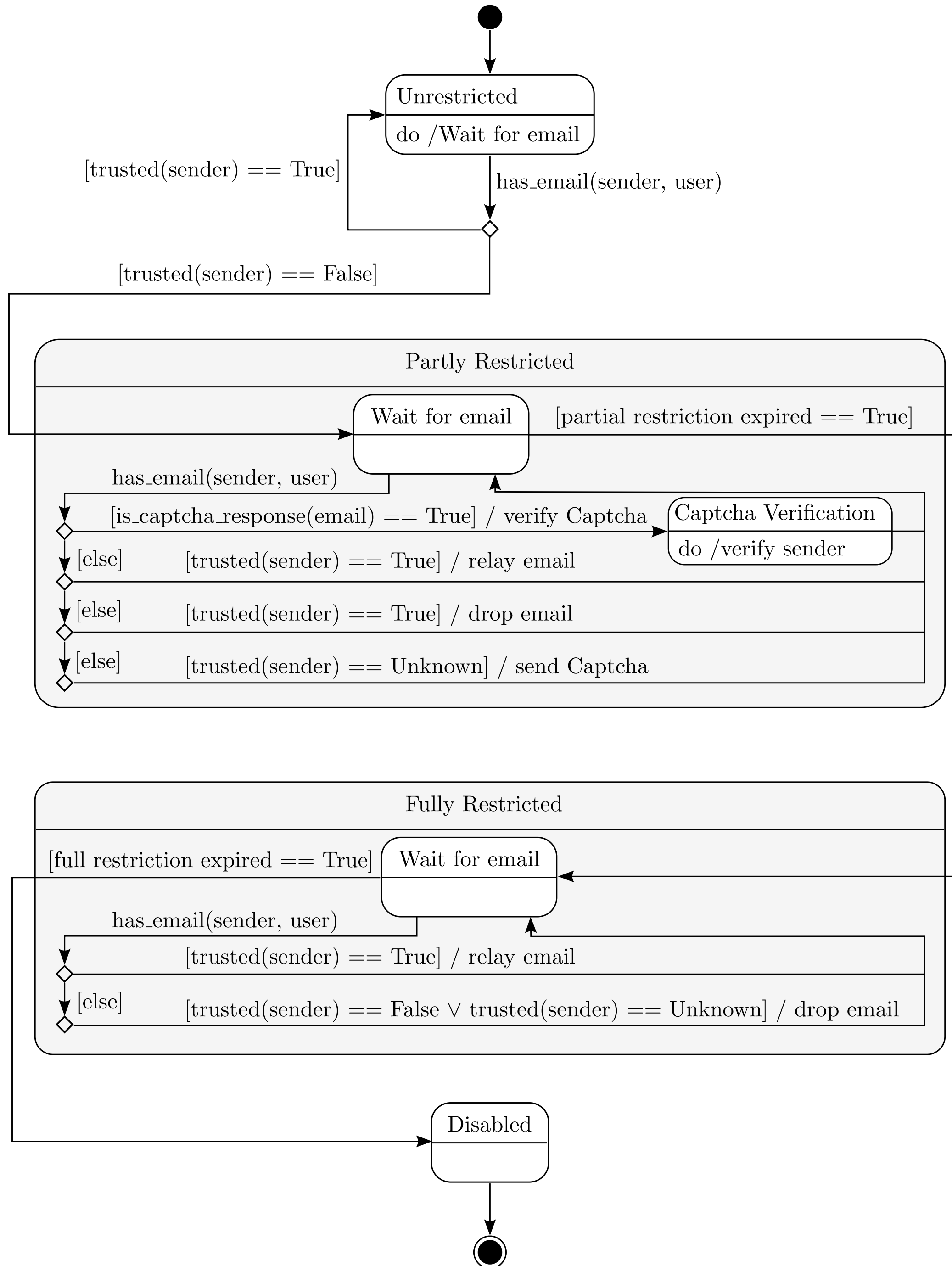


Figure 3.2: State diagram for alias.

Table 3.1: Commands used by SEAL. All commands are specified in the email Subject line. The contents of the message bodies are ignored.

Command	Service Account	Subject Line
Request alias	getalias@sealserver	⟨alias name⟩
Partly restrict alias	service@sealserver	restrict ⟨alias⟩
Fully restrict alias	service@sealserver	restrict full ⟨alias⟩
Trust sender	service@sealserver	trust ⟨email address⟩
Distrust sender	service@sealserver	distrust ⟨email address⟩

Table 3.2: Capability matrix between sender status (columns) and alias states (rows). A '✓' denotes that the sender is allowed to send to the alias, while a '✗' denotes the contrary. `CAPTCHA` denotes that the sender will be arbitrated to be trusted or not by solving a CAPTCHA challenge and receiving explicit permission from the user.

	Distrusted	Unknown	Trusted
Unrestricted	✓	✓	✓
Partly Restricted	✗	`CAPTCHA`	✓
Fully Restricted	✗	✗	✓
Disabled	✗	✗	✗

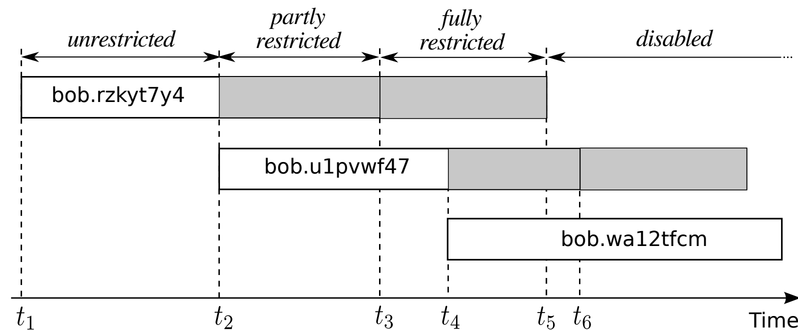


Figure 3.3: Lifecycle scenarios of three aliases. The unshaded part of a bar shows the alias is *unrestricted* while the shaded part shows that it has been leaked and becomes *restricted*.

to contacts. Figure 3.3 shows the lifecycles for three aliases. At t_1, the user requests a new alias for the alias name `bob`. We discuss the different methods for requesting a new alias in Section 3.2.3. The system returns the new alias `bob.rzkyt7y4` which can be distributed to the user's contacts. The user corresponds with the contacts on `bob.rzkyt7y4` until he observes that it has been leaked at t_2 and informs SEAL via a command email. SEAL then marks all senders prior to t_2 as *trusted*, marks the alias as partly restricted, and generates the successor alias `bob.u1pvwf47`.

At this point, it may be possible that spammers prior to t_2 are erroneously marked as trusted. However, this is reversible. The user can refine which senders should be trusted. Another possible approach may be to let the user decide the earliest time when the first spam to the alias is found and to mark all senders prior to that time as trusted. However, if the user makes a mistake in finding the first spam, mail from legitimate senders may be blocked, especially for automated systems like mailing lists. Therefore, we decided to take the more conservative approach.

Between t_2 and t_3, SEAL checks that the senders of emails sent to the leaked `bob.rzkyt7y4` are trusted before relaying the emails to the user. This also indicates that the sender has not updated his address book to send to the new unrestricted alias `bob.u1pvwf47` and so upon the user's reply, SEAL sends the original sender a reminder of the change. Email from untrusted senders is dropped. If the sender is neither trusted nor untrusted, we drop the mail and send a CAPTCHA to the sender. If the sender solves the CAPTCHA, a command email is sent to the user seeking permission to trust the sender. When the user agrees, the sender is added to the group of trusted senders and notified. Requiring the sender to solve a CAPTCHA first prevents the user from being overrun with requests, narrowing them down to requests only for senders who are likely to be human. Requiring the user's approval to add the sender ensures that consent is explicitly given.

At t_3, the first alias, `bob.rzkyt7y4` is changed to the fully restricted state. In this

state, only trusted senders can successfully send emails to the user. No new sender can become trusted as no CAPTCHA will be issued. At time t_4, `bob.u1pvwf47` is found to be leaked. The new successor `bob.wa12tfcm` is created. Trusted senders still sending to `bob.rzkyt7y4` will receive the notification to update the address of the user to the newest successor, which in this case is `bob.wa12tfcm`. At time t_5, the original alias is disabled and no emails will be delivered through it to the user.

3.2.2 Affiliation Validation: Aliases as Proof of Affiliation

As mentioned in the introduction, semi-private aliases can also be used as a means of providing proof of a user's affiliation with some organization in order to gain access to certain services or discounts. Trivially, our implementation could be trivially extended further, with the organization providing an additional service that allows its members to attach additional information profile to each alias and hosting that profile on a directory service as a more detailed proof of the identity associated with a certain alias address. With our current real-world usage in providing access to students to piazza.com forums, we have only needed to provide affiliation validation at this time.

3.2.3 Requesting an Alias

The user may need to distribute aliases under myriad scenarios. We broadly categorize them as *online* and *offline*. By online, we mean that the user needs to distribute an alias while she has network access to SEAL's server. In contrast, in an offline scenario, the user is not able to interact with SEAL over the network. However, we assume that there are opportunities for the user to access SEAL at some point in time prior to needing an alias. We sub-categorize online scenarios into alias requests and retrievals. An alias request creates a new alias that the user can distribute to a new contact. An alias retrieval refers to scenarios where the user wishes to retrieve a

```
From: bob@sealserver
To: getalias@sealserver
Subject: bobhome
```

Figure 3.4: Example email sent by Bob to request an alias.

```
From: getalias@sealserver
To: bob@sealserver
Reply-To: bobhome.b3f9cehd@sealserver
Subject: bobhome.b3f9cehd@sealserver
Body:
Your new randomized email is:
    "bobhome.b3f9cehd@sealserver"
Append this to your recipient list. We do not recommend using this
address for multiple recipients.
```

Figure 3.5: Example response to Bob's alias request.

previously requested alias associated with a website URL. To minimize the learning curve, it is important that minimal effort be required from the user when requesting new aliases under the different scenarios. We now give an overview of the most likely scenarios and describe briefly four mechanisms provided by SEAL to request aliases under different scenarios.

3.2.3.1 Request via Command Emails

In an online scenario, a user who has access to an email client can send command emails to the service address `getalias@sealserver` to request a new alias. This is the catchall mechanism since we can assume that users will normally have access to some email client. The server responds with an email containing an alias that the user can distribute to contacts. SEAL's response would be stored in the user's inbox. We also allow the user to specify a *hint* as a reminder to the context under which the alias is generated. Figure 3.5 shows a request example on the left and the server's response on the right.

3.2.3.2 Request via Browser Extensions

Another common online scenario requires the user to request an alias which may be subsequently needed for identification purposes. Specific examples of such a scenario include accessing the user's account information for a website and posting on forums. To cater to such situations, SEAL provides a browser extension that uses magic sequences for alias request and retrieval.

To detect the magic sequences, we use the same approach as PwdHash [106]. In that work, a browser extension transparently generates a unique password for the user. The other functionalities of the two extensions differ. We developed a Firefox extension that operates in two modes, *request* and *retrieval*, triggered by two magic sequences. For each browser session, when a magic sequence is detected, the extension authenticates the user with our system via their credentials. Once authenticated, a session key is generated and stored by the extension for the current session. Request mode automatically fetches a new alias from our server and is triggered by typing the magic sequence "`##[alias]#[hint]#`". A salted hash of the site's domain is stored. Retrieval mode is triggered by the magic sequence "`##$`" and is used when the user logs in to a previously registered site that requires an email address for authentication. The salted hash of the domain is used for looking up the previously created alias.

3.2.3.3 Request for Offline Distribution

The most challenging scenarios occur when a user is offline. For example, the user could be filling out a paper form at some place lacking an Internet connection. Though we have not implemented this, we envisage an SMS service that replies with a new semi-private alias whenever the user makes a request. In addition, a mobile application that caches several aliases while it has network access and dispenses them as needed could be used.

An even more challenging scenario is posting email IDs on web pages, printed

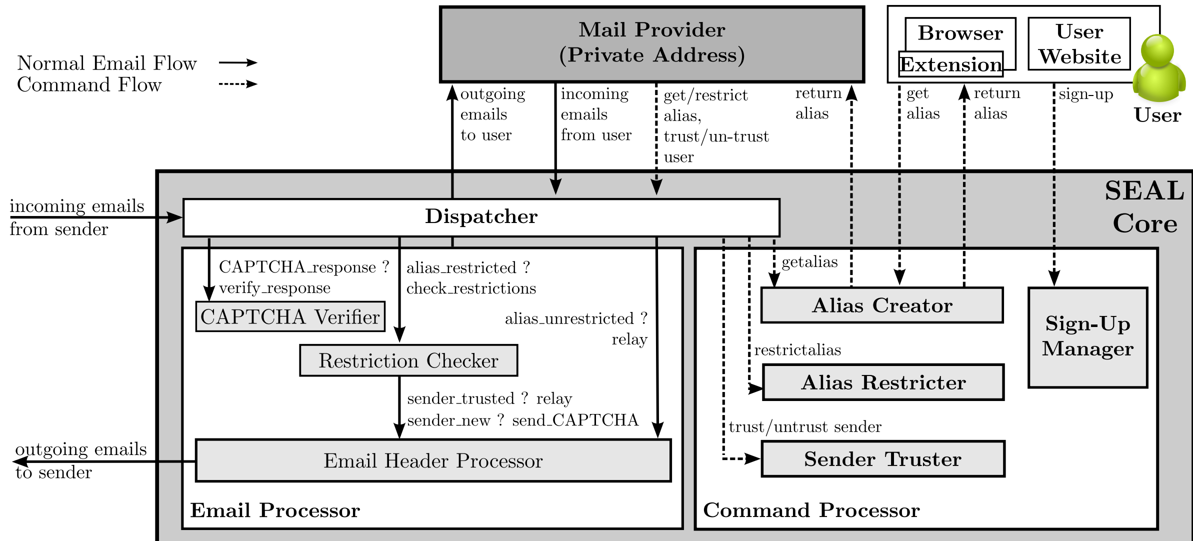


Figure 3.6: SEAL architecture.

documents, or on business cards. Since such IDs are widely disseminated, they are likely to generate both spam and legitimate use very quickly, even if the IDs are semi-private aliases. We discuss a potential solution to the problem in Section 3.5.

3.3 Architecture

SEAL's architecture is illustrated in Figure 3.6. The three main components in SEAL's core are the `Dispatcher`, `Email Processor`, and `Command Processor`. The `Dispatcher` receives emails over SMTP and passes them to the appropriate modules. If the email is a normal email, it is dispatched to the `Email Processor`. Otherwise, a command email is sent to the `Command Processor`. Other than using emails, it is also possible to interact with the `Command Processor` over HTTP/S. We discuss the components with reference to their functionalities.

Figure 3.8 shows a simplified version of our database. Each user has a salt that is used for hashing sensitive information, such as the sender's email addresses. This is to limit potential information loss in the event that SEAL is compromised.

3.3.1 Account Creation

The user creates a SEAL account by visiting SEAL's signup page and specifying the username, password, and relay address. The `Signup Manager` records this information to the database. The username and password are used for SMTP authentication by the user's mail provider when sending email through SEAL. All incoming messages to aliases and replies to command emails are sent to the relay address. By storing only the basic necessary information for SEAL's proper functioning, we aim to minimize the risks of theft of sensitive user data should our server ever become compromised. While our system works with any existing email account whose provider supports sending email as a user of another SMTP server, ideally, a new account should be created so as to start from a clean slate since the existing address might have already been leaked.

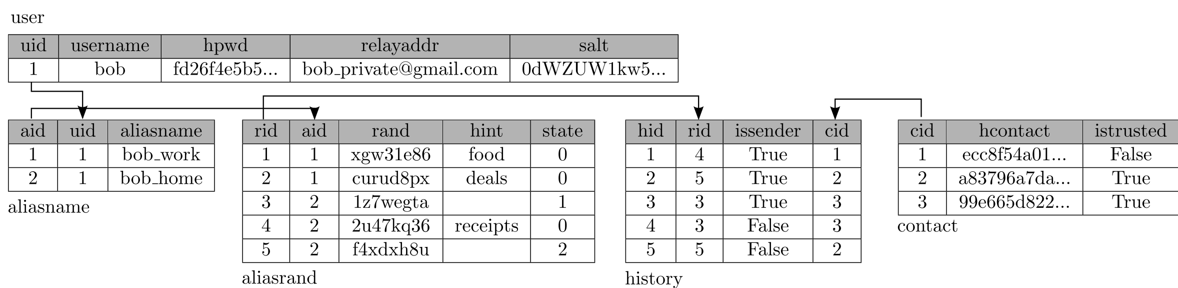
3.3.2 Alias Request

Requests for new aliases are sent to `Alias Creator`. This could be done either using a command email or an HTTP GET Request. The `Alias Creator` takes an alias name and an optional hint as inputs. If the alias name has not been taken by another user, `Alias Creator` creates a randomization string of length eight. We allow 32 possible alphanumeric case-insensitive characters (excluding '0', 'o', 'i', and 'l' to avoid potential user confusion) in the randomization string, providing a base entropy of 2^{40} bits for each alias address. This is one critical part that helps make it difficult for a spammer to correctly distinguish valid aliases from invalid ones. Implicitly, since the maximum allowable length of an email ID is 64 characters and a delimiter is used, this restricts the alias name to a maximum of 55 characters.

The optional hint replaces the user's name in the `To` header and can be used to remind the user of the context for which the alias is intended. Figure 3.7 shows an example of a hint "work". To prevent the original sender from observing the hint, it

```
From: alice@sealserver

To: "work" bob@gmail.com

Reply-To: alice@sealserver

Subject: Business Proposal

Body:

Dear Bob, ...
```

Figure 3.7: Example of using *hint*.



user

uid	username	hpwd	relayaddr	salt
1	bob	fd26f4e5b5...	bob_private@gmail.com	0dWZUW1kw5...

aliasname

aid	uid	aliasname
1	1	bob_work
2	1	bob_home

aliasrand

rid	aid	rand	hint	state
1	1	xgw31e86	food	0
2	1	curud8px	deals	0
3	2	1z7wegta		1
4	2	2u47kq36	receipts	0
5	2	f4xdxh8u		2

history

hid	rid	issender	cid
1	4	True	1
2	5	True	2
3	3	True	3
4	3	False	3
5	5	False	2

contact

cid	hcontact	istrusted
1	ecc8f54a01...	False
2	a83796a7da...	True
3	99e665d822...	True

Figure 3.8: Simplified SEAL database. In table `aliasrand`, the states 0, 1, and 2 mean unrestricted, partly restricted and fully restricted respectively.

is removed in the reply mail to the sender.

3.3.3 Managing the Alias Lifecycle

An alias' lifecycle begins when the user makes a request. `Alias Creator` then creates an entry in the database.

When a new email is received for an alias from a non-user, the `Dispatcher` checks the state of the alias. If it is unrestricted, `Email Header Processor` (`EHP`) replaces the `To` header with the user's relay address and appends the alias to the `Reply-To` header before sending it out. This causes the user to send their reply to the alias, which will result in SEAL processing the reply mail to appear as if it had been sent by that alias.

On the other hand, if the alias is restricted, the email is dispatched to the

`Restriction Checker`, which then checks if the sender is trusted. If it is, the email is relayed to the email provider via `EHP`. If the sender is new, the `Restriction Checker` tells `EHP` to generate a CAPTCHA response for the sender. Otherwise, the sender is untrusted and the email is dropped.

If an email arrives from a user, it is dispatched directly to `EHP`, which will replace the `From` header with the alias specified in the `To` header, so long as it is owned by that user.

If `Dispatcher` detects that the incoming email is a response to a CAPTCHA challenge, it forwards the email to `CAPTCHA Verifier`, which will validate the response and send a system message to the user to confirm the sender as trusted. While waiting for user validation, the sender will be treated as untrusted.

The user may mark a particular alias as partly restricted or fully restricted. This is done by sending a command email to `service@sealserver`, which will be dispatched to the `Alias Restricter`. Similarly, the user may mark a sender as trusted or untrusted. The command is dispatched to `Sender Truster`. Note that the restriction level for an alias is monotonically increasing. Once an alias is leaked, it cannot reach the unleaked state again. On the other hand, the trust level for a sender is reversible.

One possible method to automate the process of restricting leaked aliases is to leverage existing spam technologies. For example, when an incoming mail to a particular alias is flagged by a spam filter, we automatically restrict the alias. However, given the condition of current-state-of-the-art anti-spam technologies, false positives are still possible. Thus, to avoid erroneously marking an alias as leaked, we let the user perform the marking.

3.4 Evaluation

We implemented a proof-of-concept system using Postfix as the mail transfer agent and Dovecot Simple Authentication and Security Layer (SASL) for user authentica-

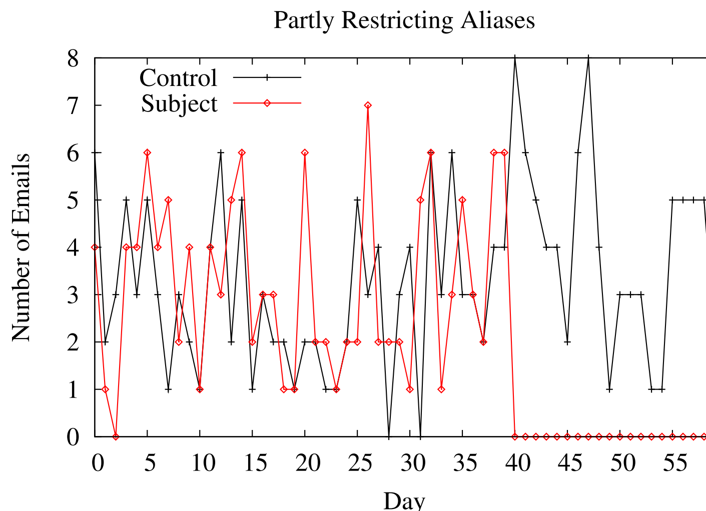


Figure 3.9: Number of emails received daily for the control and subject aliases.

tion [100, 39]. We implemented the system core as Postfix advanced content filter using Python scripts. This allows us to examine and modify email headers. Frontend web scripts provide account management functions for users. We also implemented a browser extension for Firefox by modifying PwdHash [106] so that the user can request reproducible email IDs for filling out web forms. There are three main parts to our experiments. In the second part, we offered the system as an option to a class that was asked to sign up with a discussion forum that requires their affiliation with the university to be validated using email addresses. Lastly, we registered with several websites and studied potential address leakages.

3.4.1 Effectiveness of Partly Restricting Aliases

To examine the effectiveness of our system in restricting aliases in a real world scenario, we registered twice with a website that promises users discounts and points on merchandises using two different aliaseson two email accounts, one of which served as a control. The website was chosen because it actively sends subscribers emails. On Day 40, we restricted both the control and subject aliasby transitioning it to a partly restricted state. Additionally, we marked the website as untrusted from the

subject account. The number of emails sent to each of the aliases were plotted in Figure 3.9, which shows that in contrast to the control alias, no emails were sent to the restricted alias after Day 40. This allows us to conclude that the website was able to continue sending mails on a partly restricted alias when it is not marked as untrusted. Conversely, once marked as untrusted, it is not able to send emails to the subject alias. Note that the difference in the number of emails received by the aliases on the same day are due to the site sending different emails to each alias.

3.4.2 Affiliation Validation

To study the system being used in a real world scenario where a web service requires an official email address for validating the user's affiliation, for one semester, we provided the students of a class an option to use the system for receiving updates from a discussion forum that accepts only email addresses having university domains. Including three instructors, there were 68 potential participants. The students were neither incentivized nor disincentivized to use SEAL. They would have been able to register with the discussion forum using their academic email addresses. 55 (80.9%) people proceeded to create aliases and used SEAL actively for the whole semester. Five of the users created two aliases, one created three aliases, and another created five aliases. The others created one alias. Figure 3.10 shows the number of emails processed by SEAL per day while Figure 3.11 shows the daily number of aliases that were active. The days with low email transactions coincide with weekends, school break, and public holidays. While not all aliases may be active daily, the figures show that the number of users using the system remains relatively constant throughout the semester. Even though the system is only a prototype, there were no participants who stopped using the system prematurely, demonstrating the practicality of the system prototype.

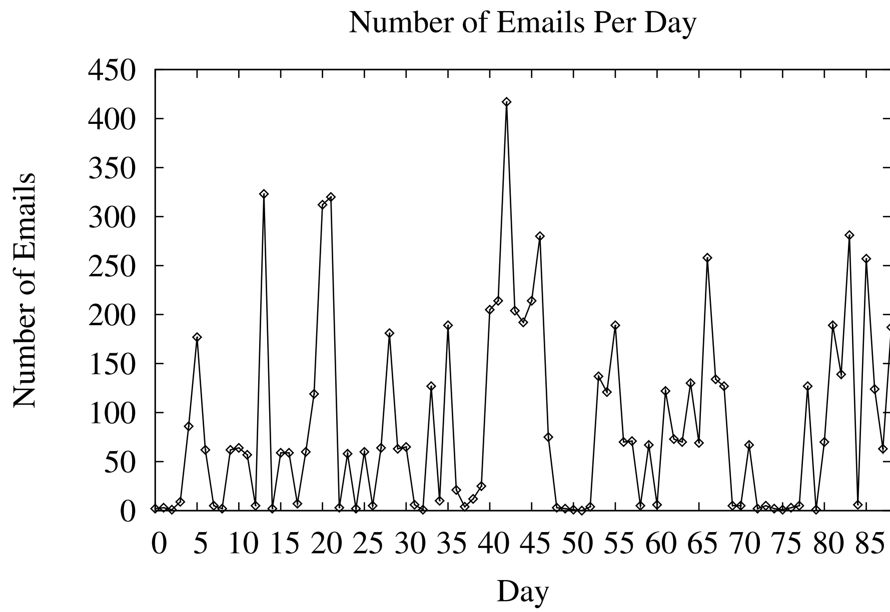


Figure 3.10: Number of emails processed daily.

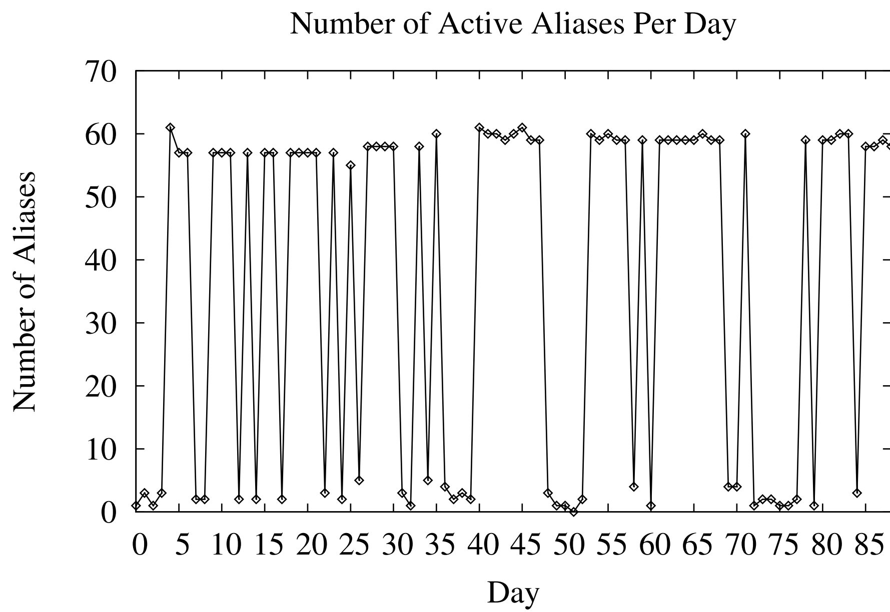


Figure 3.11: Number of active aliases per day.

3.4.3 Leakages

In this part of the experiments, we examine potential leakages of email aliases.

3.4.3.1 Leakage by Websites

To examine whether websites and mailing lists are sources of email address leakages, we created aliases and used them to register with 56 websites. In an attempt to diversify the websites instead of choosing only those ranked as highly popular by survey companies such as Alexa [16], the websites were chosen arbitrarily by searching for keywords including "shopping", "fast cash", "movies", "music", "cheap flights", and "education". We attempted to register with 70 web sites, succeeding in registering on 56 of them. Two of the websites initially rejected the registrations as they did not accept email id lengths exceeding 30 characters. However, we were able to register successfully after using a shorter alias name to satisfy that requirement. Three websites disallowed the period character in email ids. We do not view this as a limitation of our system since the RFC clearly states that the email id may be up to 64 characters long and the period character is allowed [53]. Remaining 10 failures were due to them requiring credit card information and real cellphone numbers.

Using aliases we also registered with another 101 websites from 15 categories listed by Alexa as the most popular sites using unique random aliases [16]. The 15 categories are arts, business, computers, games, health, home, kids and teens, news, recreation, reference, regional, science, shopping, society, and sports. In addition, we subscribed to 15 mailing lists. The mailing lists were ranked amongst having the most subscribers by L-Soft, the company that invented electronic mailing lists [76].

After fifteen days, we collected and analyzed the domains from which the senders are emailing each of the alias. We used the domains from the email envelopes instead of the email headers, since it is easy for an adversary to spoof the "From" header. Figure 3.12 shows the distribution for the different number of aliases for varying

Table 3.3: Table of aliases used for website, mailing list and newsletter registrations, sorted in increasing number of unique sender domains. The first column lists the Case IDs while the second column shows the alias. The third column indicates the number of unique domains of the senders and the last column lists some of the domains. In the interest of space, entries with senders from more than six unique domains are truncated.

Case ID	#	Sender Domains
A1	2	website.mlb.com, bounce.ed10.net
A2	2	emailconfirm.com.com, noreply.gamespot.com
A3	2	australia.care2.com, bounce.bluestatedigital.com
A4	2	connect.match.com, returnpath.bluehornet.com
A5	2	bounce.em.ign.com, bounce.mkt1839.com
A6	2	paypal.com, bounce.ed10.net
A7	2	signaturesurveys.com, server220.go-mama-hosting.com
A8	2	b.mypoints.com, mail.hpshopping.com
A9	2	envfrm.rsys5.com, animoto.com
A10	2	crosswalkmail.com, salememail.net
A11	2	email.decrease4u.net, QuickenLoans.com
A12	2	ebay.com, us.emarsys.net
A13	3	email-bounces.amazonses.com, facebookmail.com, bounce.game.e.playdom.com
A14	3	echineselearning.com, in.constantcontact.com, bmsend.com
A15	3	mail.christianmingle.com, ChristianMingle.com, believe.com
A16	3	pandaresearch.com, paidsurveysforyou.com, arcamax.com
A17	4	yourfreesurveys.com, surveyhelpcenter.com, myview.com, bounce.exacttarget.com
A18	5	rootsweb.com, email.ancestry.ca, email.ancestry.com.au, email.ancestry.com, email.ancestry.co.uk
A19	6	ssprd9.net, 5in5now.com, clearvoicesurveysmail.com, mailboto24.com, bounce.npdmr.com, mailboto21.com
A20	6	jangomail.com, ownattention.com, freebieape.info, royalofficials.com, weekenddefeat.com, galabenefits.com
A21	13	litmus.modulelaunches.net, squaresz.com, nast.zoncatalor.com, tourer.fillsavings.com, ecipwriver.com, tingly.muterdepordet.com, etc.
A22	14	downpours.net, berks.philosophersr.com, unff.neswooleston.com, brazil.lxxia.com, pinch.istrowesturase.com, paolo.flatstudio.net, etc.
A23	16	mydailymoment.com, list.cheapflights.com, inboxpays.com, bounces.lifescript.com, inboxdollars.com, mailboto21.com, etc.
A24	128	sellingprocess.info, theconfident.info, yourcouponworld.info, mycontentsite.info, mycrowdsourcecentral.info, emilestone.info, etc.

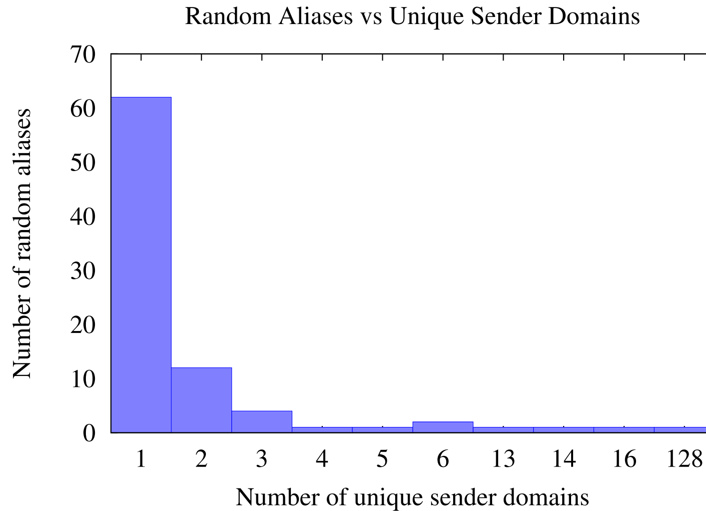


Figure 3.12: Histogram of the number of aliases for different number of unique sender domains.

number of unique sender domains. Table 3.3 lists examples of the sender domains. In the interest of space, only interesting cases are shown.

There were mails sent to 88 of the aliases. 62 (70.45%) of the aliases had senders from one domain. Having senders from multiple domains do not necessarily constitute leakages as some domains are *affiliated*. We define two domains to be affiliated if the registrants for the domains are the same or they are declared to be affiliated in the privacy policies or terms of service. Referring to Table 3.3, we examined the domain affiliations for Cases A1 through A20 and identified sender domains affiliated for each alias.

We noticed that for Cases A21 through A24, the number of unique sender domains ranged from 13 to 128. We returned to examine the private policies for these websites. All four policies stated that email addresses will be shared with other sites. An example of such clauses is, "We may share User information with third parties as reasonably necessary for us to operate this website and to provide offers and services to Users". It is easy for users to miss such clauses as they are usually obscured amongst lengthy privacy policies.

Table 3.4: Table of aliases used for classified advertising and forum postings, sorted in increasing number of unique sender domains. The layout is the same as in Table 3.3. Sender domains in **bold** are identified instances of leakages.

Case ID	Aliases	#	Sender Domains
B1	m4kkxa4d	1	**my.ohecampus.com**
B2	dpuxqbxg	1	**gmail.com**
B3	bob.m4kkxa4d	2	health.webmd.com, webmdmessage.com
B4	bob.n13va5ck	2	**adobe.com**, macromedia.com
B5	bob.qf11md51	2	alibaba.com, **hotmail.com**
B6	bob.wa12tfcm	8	maestro.independenttraveler.com, cruisecritic.com, tripadvisor.com, **gmail.com**, lists.sniqueaway.com, lists.airfarewatchdog.com, etc.
B7	bob.dpuxqbxg	14	**scmc050.net**, **ns2014560.ovh.net**, **kataros.com**, **fbi.gov**, **gmx.us**, **s15355439.onlinehome-server.info**, **muhleheidemusikanten.nl**, etc.

While investigating the sender domains, the ease with which we were able to find all emails sent to a particular alias was very encouraging. This demonstrated the advantage of using aliases for investigating potential leakages. Without using aliases, it might have been an extremely challenging task for the user to distill out bad websites such as for Cases A21 through A24. The user can then surgically mark the aliases for these cases as leaked without affecting the registrations for the good websites.

3.4.3.2 Leakages by Online Posts

To study the email address leakages through online message postings, we posted messages on seven forums and one popular classified advertising site. These were found from amongst the 45 sites we used in Section 3.4.3.1. For each posting, we generated a new alias and displayed it in clear in the message body. After 15 days, we examined the mails sent to these aliases. Table 3.4 shows the sender domains for aliases that had emails sent to them. We observed two leakages on two forums hosted by `tripadvisor.com` and `webmd.com`. These are Cases B1, B3, and B6. Cases B1 and

B3 are related. Clearly, the sender in Case B1 was intending to send to `bob.m4kkxa4d`. However, perhaps due to parsing error, the mail was sent to `m4kkxa4d` instead. We were able to observe this abnormality because we intercepted all emails sent to our server. The email in Case B1 was not forwarded to the user's email provider. The emails sent in Cases B4 and B5 were legitimate responses to our forum postings.

In addition, there were 24 emails from various senders sent to the alias that we used for posting an advertisement on a classified advertising site. These are Cases B1 and B6 in Table 3.4. Based on the email contents, seven of these appear to be legitimate queries, while the remaining 17 emails contained messages that were irrelevant to the original context. Six emails claimed to be from the administrator of the site, one from FBI and one from a reputable bank. Eight of them contained links to external suspicious sites.

3.4.4 Timing Performance

Email systems use a store-and-forward model. Numerous factors contribute towards the time taken for an email to be sent to its recipient, including network latency, spam or virus detection filtering, and overloaded relay servers. While delays are generally tolerable, any additional processing on the emails by servers such as SEAL should be reasonable. Towards understanding the timing overheads incurred by SEAL, we measured the arrival times of emails at the email relay servers as indicated by the time stated in the `Received` header field. While this is not ideal for several reasons including clock skew between different servers, incorrect date and time on some servers, and timing information having only a granularity of seconds, it allows us to approximate the overhead incurred by SEAL. Moreover, the lack of access to other servers does not allow us a detailed comparison of performance data.

We synchronized SEAL's clock with an NTP server and assume that other servers did the same. The `Received` header fields are added by each SMTP server as the

Table 3.5: Percentages for five groups of shortest delays.

Delay (secs)	Percentage	Number
0 to 1	81.383	103,189
1 to 2	9.465	12,001
2 to 3	1.779	2,255
3 to 4	0.900	1,141
4 to 5	0.605	767

email is accepted. Figure 3.13 shows an example of these fields for an email. We compute the differences between the timestamps of two consecutive entries and refer to them as *delays*. In the example, we use the delay between entries 2 and 3 as an indication of the processing time required by SEAL to analyze and forward the email. At entry 2, the email is marked as received by SEAL, after which it is processed. It is then sent to the outgoing queue with entry 3 added. While we could have timed the scripts, the lack of data from other servers would render any comparison meaningless.

Table 3.5 shows the percentages for the five groups of the shortest delays. 126,794 delays for emails were collected from one of the author's email accounts and a SEAL account. The mean and standard deviation for these delays is 105.116 seconds and 21,232.627 seconds respectively. 3,706 delays were incurred by SEAL, with the maximum and minimum delays being five and one seconds respectively. The average delay contributed by SEAL is 0.274 seconds. This is at 0.00494 standard deviation away from the mean. This can also be inferred from Table 3.5 that shows the percentages for the five groups of shortest delays. 81.383% of the delays are from zero seconds to one second, in which SEAL's average delay lies. The delay incurred by SEAL is thus insignificant in comparison to other delays.

3.5 Discussion - Security and Usability

Although our construction of semi-private aliases seeks to minimize inconvenience to legitimate senders, there are remaining issues, some of which also apply to existing

```
(6) by 10.231.190.83
    with SMTP id dh19csp37616ibb;
    Sat, 25 Feb 2012 07:02:20 -0800 (PST)
(5) by 10.50.178.73
    with SMTP id cw9mr7274127igc.23.1330182140761;
    Sat, 25 Feb 2012 07:02:20 -0800 (PST)
(4) from seal.eecs.umich.edu
    (d-110-235.eecs.umich.edu. [141.212.110.235])
    by mx.google.com
    with ESMTP id no10si2673927igc.10.2012.02.25.07.02.20;
    Sat, 25 Feb 2012 07:02:20 -0800 (PST)
(3) from seal.eecs.umich.edu (localhost [127.0.0.1])
    by seal.eecs.umich.edu (Postfix)
    with ESMTP id EE11954C72F
    for <johnsmith@gmail.com>;
    Sat, 25 Feb 2012 10:05:12 -0500 (EST)
(2) from backend.www.inm.smartertravel.net
    (backend.www.inm.smartertravel.net [75.98.73.172])
    by seal.eecs.umich.edu (Postfix)
    with ESMTPS id CA35354C722
    for <ads.j1pdkqa5@seal.eecs.umich.edu>;
    Sat, 25 Feb 2012 10:05:12 -0500 (EST)
(1) from smarter (helo=localhost)
    by backend.www.inm.smartertravel.net
    with local-bsmtp (Exim 4.76)
    (envelope-from
    <b-KEEXNPCTCQ-38936-2893808-AWDSubscriptionUtils
    @lists.airfarewatchdog.com>)
    id 1S1J8d-0007SB-QG
    for ads.j1pdkqa5@seal.eecs.umich.edu;
    Sat, 25 Feb 2012 10:02:19 -0500
```

Figure 3.13: Values of the `Received` header fields for an email, annotated with the order in which they were pushed onto the mail header. The receipt timestamps are highlighted in gray.

DEA systems. During the transition of an alias to the restricted state, there are some cases in which known legitimate senders may be treated as untrusted. For instance, in a more severe case, if a user is subscribed to a mailing list under a semi-private alias that the user later marks as restricted, and then the domain name of the mailing list's server is changed, the mailing list would then be treated as untrusted and would likely ignore our service's prompts to solve a CAPTCHA challenge, resulting in that newsletter being silently blocked. One simple mitigation would be to deliver these messages to the user's spam folder, instead of completely blocking them (this requires cooperation between SEAL and the mail provider). The user can then mark the senders that are incorrectly delivered as trusted.

Another concern is the potential that spammers could also misuse SEAL. For example, they could create aliases to be used in the From field of spam messages, providing a channel for the recipients of spam to respond (e.g. to spam advertisements). But it is not clear if this offers significant advantages to spammers. Spammers already have the ability to create multiple email addresses using mail servers they control and, as far as we are aware, this does not help them bypass existing spam defense mechanisms. This is an area for further investigation. Note that a spammer would still have to create an account with an email provider that is coupled with their SEAL account. Legitimate SEAL servers could be configured to permit only coupling with email providers that have checks against spammer registration or receipt of large amounts of bulk mail in short intervals (e.g., Gmail appears to have such controls). Illegitimate SEAL servers that are primarily designed to protect spammers would probably get blacklisted over time, just as mail servers do.

Spammers could also attempt to attack SEAL protocols directly. For example, a spammer could attempt to spoof a legitimate user and send commands to add themselves to the trusted set. But, to do that, the spammer would need several pieces of information that are not easy to get: email ID with the mail provider

and account userid/password on SEAL. Email sent to command addresses, such as getalias@sealserver, is rejected unless it arrives over an authenticated SMTP session and the commands are executed under the identity of the user that authenticated, rather than the content of the "From" field in the message headers.

Spammers could attempt to compromise SEAL infrastructure as well. While SEAL servers should be secured using best practices, one should minimize the damage that results in case the server is compromised. We consider two forms of attacks: (1) a one-time intrusion that simply steals all the data within the databases and (2) an active attack where the attacker compromises the code within the server. In the first case, the only email IDs that the attacker gets hold of are the user's email ID at the mail provider (Gmail ID in Figure 3.6. All other email IDs are stored as salted hashes, which should be difficult to reverse[1]. Our implementation recommends that the user create a fresh, private account on the Gmail provider. That email address should not be publicly used – all email from it is routed via the SEAL server by configuration of SMTP settings within the mail provider. Recipients only see semi-private aliases. If the email ID at the mail provider is ever leaked, it is easy to change, since it is only relevant to the owner and not shared.

In the second case, if a spammer compromises the SEAL servers, they can monitor emails flowing through the system and collect addresses. While this is serious, the addresses collected are limited to the time that the attack goes unnoticed. It is certainly less serious than the compromise of an email provider, where both older messages and future email are potentially accessible.

SEAL is not designed to provide anonymity against local network snooping. A government, for example, could monitor the network channels to a SEAL server and collect emails, since they could go over unauthenticated and unencrypted SMTP from arbitrary senders. As far as we are aware, this is not a typical attack used by

[1]Besides, if the spammer had a dictionary of email IDs, there are cheaper means of verifying them than trying to do a dictionary attack on the salted hashes.

spammers.

Despite our efforts to make SEAL easy to use and minimize impact on non-spam senders, we acknowledge that some users will still prefer permanent addresses to semi-private aliases. Permanent addresses have the advantage that they can be printed on business cards, are easy to remember, and thus hand out. With SEAL, a user could generate an alias on a mobile device and then write it by hand on a form or business card (which may not be too bad for one-on-one situations). For better scalability when the user is handing out the cards to a large number of users, a possible solution would be to publish a means for a requester to send a text message and receive the alias as a response. This ties the requester's cell phone number to the alias. Cell phones are sufficiently common now among email users that we don't see this as a significant usage barrier.

For publishing email IDs on web pages, we are currently experimenting with a mechanism that generates a semi-private alias on the web page based on the IP address from which the HTTP request was received. The reason for looking into this is to investigate if this provides additional means to identify servers that are used to harvest email IDs from web sites. We are still in the process of collecting data from this mechanism.

One significant usability concern with SEAL is that, over time, one person could appear multiple times in an address book. This would occur when email containing aliases in the From or To fields is sent to a group. When those aliases are added to an address book, one person may end up with multiple aliases in an address book. This occurs today also to some extent as people both have work and personal email accounts. As a result, many address books permit multiple email IDs to be associated with one person. With SEAL, being able to mark an email ID as the preferred or primary email ID will be useful. In our design, we require the alias name of an alias to be associated with a single account. As a result, a SEAL-compatible address

book could automatically associate all email IDs that have the same alias name (e.g., aliasname.*@sealserver) with the same person.

As mentioned before, the browser extension for using aliases as web usernames was adapted from PwdHash [106], and so it comes with some of the same limitations as their work, including lack of portability to all applications that render HTML, vulnerability to spyware coexisting on the same computer, and susceptibility to attacks on DNS to confuse the resolution of domain names. One potential improvement in usability over PwdHash comes from the convenient fact that the username field is not normally scrambled on login web forms, so that the user can more easily see the fetching and replacement of their username and know that it was successful. It is also notable that while the user must input a sensitive password when using PwdHash, the information being input for our extension is not nearly as sensitive, and so attacks such as focus stealing are not likely to pose as substantial a threat to web account security.

3.6 Conclusion

The current paradigm does not provide email address owners sufficient control of their addresses, leading to email address leakages, and thus rogue accesses to email addresses. In addition to traditional risks posed by underground crackers, some services require the users' official addresses to validate their affiliations with certain organizations. Current technologies do not allow users to provide alternative addresses that do not over-disclose user information to these services.

We propose the concept of semi-private email aliases and its embodiment, SEAL, a system that provides users more control over their email aliases and allows web services to validate the user's affiliation with an organization without having access to the user's private information. Semi-private aliases are randomized email addresses that can be restricted progressively when the user detects that they have leaked. This

is related to the common disposal feature of DEA systems. However, what distinguishes SEAL from other DEA systems is that it both includes a more advanced mechanism for managing finer-grained alias lifecycles, allowing for a more flexible approach to retiring compromised email addresses, and also that it integrates fully with current email systems while at the same time not being overly restrictive. Experimental results indicate that SEAL can be useful in controlling unsolicited email, while being compatible with existing email systems.

In organizational settings, SEAL also permits use of aliases to validate a user's affiliation, while preventing disclosure of the work-related email ID or associated information. This proved useful in a test deployment where an instructor of a freshmen course at our institution required students to use an online forum provided by Piazza.com but did not wish to require the students to disclose their university email ID to the service because of concerns about student privacy. Piazza.com's default sign-up mechanism uses student's email IDs to validate their university affiliation. Over 80% of students chose to use email aliases issued by SEAL rather than their university email ID, to sign up at Piazza.com.

CHAPTER IV

Reducing System Permission Gaps with DeGap

4.1 Introduction

Permissions for system objects (e.g. files and network sockets) on host machines are granted explicitly via configuration files or system data structures such as inodes. Discovering potential permission gaps for system services can be hard. As operating systems and software mature, clear-cut cases of vulnerabilities due to permission gaps in commonly used software, such as allowing everyone to read and write critical log files for syslog-ng [32], will likely be discovered and fixed. However, new or less commonly used software may not have been subjected to a sufficient level of scrutiny, potentially allowing permission gaps to arise and persist. Actual occurrences of this include file permission vulnerabilities in openswitch-pki [26], logol [25], and extplorer [24].

Furthermore, different systems often have different ways of managing object permissions. For example, file permissions in Unix are configured via mode bits. But, permissions in various network services, such as `SSHD` daemon, are usually controlled via configuration files. In `SSHD` daemon, a variety of settings are used, such as `AllowUsers` (a list of users) and `PermitRootLogin` (a Boolean flag), to control access.

One method for identifying potential permission gaps is to use static analysis of a system. In several works on Android apps [41, 20, 21], the authors compared two sets of permissions for an Android app: (1) permissions used by the app from the user; and (2) the permissions that could be used by the app, based on static analysis of the code. The extra permissions used could pose a potential risk, even though static analysis shows them to be unused because malware could potentially exploit them by, say, code injection.

Unfortunately, the technique of doing static analysis on a system to identify gaps is not broadly applicable. In [21], an example of a firewall with too liberal permissions is used to motivate the need for closing permission gaps. But, static analysis of services listening behind the port and comparing it with firewall permissions is unlikely to identify the gaps because, usually, only the firewall is responsible for doing the blocking of remote requests. There is only one reference point, the firewall policy, and there is nothing else to compare with to identify a potential gap. Similarly, a service like ssh daemon can use potentially all the available configuration settings, even permissive ones, and static analysis of ssh daemon is unlikely to identify permission gaps.

Furthermore, any permission gap analysis is likely to be computing an *estimate* of a gap. Security requirements change over time and may not even be precisely known. A user may have been granted access to a object at some point in time but may no longer need access. The administrator of the object may not be aware of the change. Even in the static analysis example, there could be a rare app that wants to acquire extra permissions from users, anticipating a future capability in the app.

The above leads to the question of what is the best we can do in an automated manner, given that permission gaps exist in systems but static analysis is not feasible and any analysis must necessarily be an estimate? We make use of a simple idea to derive an estimate of the permission gap. DeGap examines the past usage of

objects and compares them with the granted permissions to derive an estimate of the permission gap, i.e., opportunities that users should consider for tightening object permissions.

Information on past usage is often available in log files or can be collected by appropriately auditing the system. Unfortunately, there are two problems. First, logs typically contain overwhelming amounts of data. They are often poorly formatted for detecting break-ins [116], let alone unneeded permissions. While there are tools utilizing these logs for various purposes including file integrity checking, intrusion detection, and troubleshooting [12, 88, 77, 133], to the best of our knowledge, logs are not currently easy to use for identifying permission gaps. Second, different services, such as as file systems, firewalls, and services such as ssh, use different methods for configuring permissions. Providing a generalized solution that works for multiple services appears to be non-trivial.

In this Chapter, we describe DeGap, a common framework for discovering permission gaps and suggesting solutions for configuration settings towards reducing the gaps. We used DeGap to analyze logs generated by two very different services: `auditd` and `SSHD`, with respect to utilized permissions. Users can use the tool to help determine a lower bound on the set of permissions for a object that is consistent with its usage during a selected period. These permissions are then compared with the assigned permissions to expose potential permission gaps. DeGap then proposes a set of suggested configurations for achieving that lower bound. While we demonstrate the feasibility of our framework in the setting of system objects, it should be readily applicable to other scenarios, such as helping users to improve privacy settings in social networks.

Overall, we make the following contributions:

- We describe a framework and a set of principles that we used to design DeGap for identifying permission gaps. This framework was designed with applicability

to different services in mind, and we describe the framework's core components while also highlighting the design nuances for the components that need customization for different services.

- We implemented DeGap, and demonstrate the flexibility of the framework by developing components for extracting and analyzing permission gaps from logs and configuration settings for both `auditd` and `SSHD`.

- We show that DeGap is able to discover permission gaps by presenting an analysis performed using DeGap on the permissions of several actively used machines. For `SSHD`, we discovered that two users had not had their access revoked even though it was no longer required. Also, DeGap was able to identify that legitimate users only used the public key authentication method to log on to one of the servers, but password-based authentication was unnecessarily allowed. The server had been the target of password brute-force attacks, suggesting that revoking the password-based authentication method could serve as a low-impact means of tightening the authentication policy and enhancing security. For `auditd`, DeGap discovered a private key for one service that was mistakenly set to be world-readable. It also found two additional files on the servers that could be exploited to execute a privilege escalation attack. Finally, DeGap identified various user groups that were unused during monitoring and were candidates for tightening.

- We demonstrate DeGap ability to automatically suggest changes that can be made to configurations for reducing permission gaps. For both `SSHD` and `auditd`, DeGap generated correct suggestions for reducing permission gaps.

- For `auditd`, which generates extensive logs, we explore potential improvements to the auditing process that can speed up subsequent permission gap analysis.

The Chapter is structured as follows. Section 4.2 discusses the limitations of DeGap. Section 4.4 presents definitions and basic techniques that are used as building blocks for for discovering gaps and identifying fixes. Section 4.5 describes the system architecture. Section 4.6 presents evaluation results. Section 4.7 explores techniques for speeding up parsing of logs by modifying the way logs are collected. Finally, Section 4.8 presents conclusions and directions for future work.

4.2 Limitations

DeGap uses logs, which record past accesses to objects over the period of analysis. Clearly information about prior accesses provides no guarantee regarding patterns of future access. But in the absence of the ability to accurately predict future uses, DeGap provides a means for identifying potential permission gaps so long as access patterns remain unchanged.

An adversary may have accessed a object prior to or during an analysis of the logs by DeGap. DeGap is not able to distinguish legitimate accesses from illegitimate ones. Illegitimate accesses can be filtered out if they are known prior to analysis, for example using reports from an intrusion detection system. DeGap provides a database and a query engine that supports the exclusion of known illegitimate accesses. For unknown illegitimate accesses, an attacker's activity will be treated as normal; in that case, DeGap can still be useful in tightening the system to prevent other attacks that did not occur during the analyzed period.

We note that since accesses are logged after they have occurred, DeGap is by design incapable of preventing illegitimate accesses as they happen. Without reinventing the wheel, we leave this responsibility to existing tools, such as intrusion detection systems. The same limitation applies to other security alerting systems [57], such as Tripwire, a popular open-source system that detects changes to file system objects [13].

4.3 Relationship between Permission Gaps, Permission Creep, and Attack Surfaces

Eliminating permission gaps inherently stops permission creep and reduces the attack surface of a system. It may therefore be useful to understand the relationship between these terms, and the implications of this relationship.

Permission creep is an insidious accumulation of permission gaps over time [96], for example, due to granting employees additional permissions beyond their normal roles for a temporary need, but failing to revoke permissions when employees leave or the need disappears. Resolving the permission gap problem implies that the permission creep problem will also be resolved. However, the converse is untrue since permission gaps can still exist in the absence of permission creep. For example, in the case of a single user system, permission creep is unlikely to occur while permission gaps may exist. The term "permission creep" is also used by Vidas et al. [131]. However, the semantics of that term would better match that of "permission gap" in our context as well as Felt et al. [41] and Bartel et al.'s work [21].

The attack surface of a system is the set of methods available to a potential attacker for gaining access to a system with the potential to inflict damage [80]. Clearly, having a smaller attack surface roughly corresponds to having a more secure system. Granted permissions contribute to a system's attack surface and permission creep increases the attack surface "area". Eliminating the attack surface entirely by removing all permissions is impractical, but in pursuing an adequate level of security, it should be the system owner's objective to reduce their system's attack surface as much as is practical. DeGap provides necessary information to help achieve this.

4.4 Tightening System Permission Gaps

As a reminder, we represent a *permission* as a tuple (s, o, r) that denotes subject s as having the right r to access object o. In turn, system objects are collectively guarded by a set of *granted permissions* $P_{\mathcal{G}} = \{g_1, g_2, \ldots, g_n\}$, where each g_i is a permission tuple (s_i, o_i, r_i). A subset $TrueGap$ of granted permissions $P_{\mathcal{G}}$ are not required for all legitimate accesses to succeed. This subset constitutes the *true permission gap* for the system. These extraneous permissions potentially increase the attack surface of the system, and the goal is to help users discover these gaps and recommend ways of fixing them.

The problem is that computing $TrueGap$ is generally not possible for an automated tool in the sense that the precise set of legitimate accesses that should be allowed normally cannot be automatically inferred if the only reference point available is the set of granted permissions. A second reference point is needed to compute an estimate of the gap.

We attempt to compute an estimate of the $TrueGap$ by using past accesses, usually recorded in system logs, as a second reference point that can be compared with $P_{\mathcal{G}}$. Since permissions have to be granted explicitly for objects in system services to be accessible, we can simplify $P_{\mathcal{U}}$ to represent the set of permissions that were appropriately authorized, as per log files, i.e., $P_{\mathcal{U}} \subseteq P_{\mathcal{G}}$. Let P_{Δ} represent the set of permission gaps, where $P_{\Delta} = P_{\mathcal{G}} \setminus P_{\mathcal{U}}$. There is no permission gap if and only if $P_{\Delta} = \emptyset$.

In a static system (where $P_{\mathcal{G}}$ and $TrueGap$ do not change), the permission gap P_{Δ} from the above definition will be be an upper bound on $TrueGap$. How tight this bound is will generally depend on both the quality of the log files and the period of time that they cover. The approach we take is that since $TrueGap$ must be estimated, a reasonable choice, as good as any available for most services, is to estimate the gaps based on past usage. At least, that way, users have some well-defined reference point

when deciding whether to tighten permissions. Using this definition also permits users to ask the following question:

> Given a permission that is proposed to be revoked, were there actions in the past that would have been denied had the permission never been granted in the first place?

We believe that being able to answer the above question is important before revoking permissions as it may be indicative of denial-of-service problems that may arise with permission revocations. Using a firewall analogy, if blacklisting a sub-domain of IP addresses is proposed as solution to prevent attacks from a subset of nodes in that domain, it is important to consider whether there have been legitimate accesses from that sub-domain in recent past. If there were, then blacklisting the entire sub-domain may turn out to be an unacceptable option.

Our approach of using best-effort estimation of gaps is consistent with the approach in other areas of applied security. IDS systems and virus detection systems do not always guarantee correctness. They are still useful to administrators as an aid in securing systems.

Finally, we note that we used a model of subjects, objects, and rights to express permissions. Without loss of generality, the approach could be extended to support more detailed models of permissions, e.g., where subjects, objects, and rights have attributes and granted permission is viewed as a combination of approved attributes of subjects, objects, and rights. The key requirement is that there be a way to compute $P_{\mathcal{G}} \setminus P_{\mathcal{U}}$, given $P_{\mathcal{G}}$ and $P_{\mathcal{U}}$.

4.4.1 Gap Analysis and Traceability

Our goal in gap analysis is not only to determine whether there is a permission gap, but exactly the setting in a configuration file or the permission on an object that contributes to the gap, i.e., the question of mapping $P_{\mathcal{G}} \setminus P_{\mathcal{U}}$ back to specific

configuration settings; ideally, the administrator should be told the specific setting that contributes to the gap rather than having to make guesses. This is particularly important because gaps are likely to be reported at a lower level of abstraction by the tool than what an administrator is going to be used to.

The challenge in identifying the settings in a configuration file that are candidates for tightening is that a reverse mapping from P_{Δ} to a configuration setting may not always be available or straightforward to provide. For example, if logs indicate that remote root login to `SSHD` is not required, identifying the specific place to make the change (e.g., `PermitRootLogin` field or the `AllowUsers` field) will require a fair amount of domain knowledge. One potentially has to write two parsers, one to go from a configuration file to $P_{\mathcal{G}}$ and another from gaps to specific settings in a configuration file.

DeGap supports two approaches to the problem for discovering changes in configuration settings that lead to reducing the permission gap. In the first approach, the existence of a reverse map from gaps to configuration settings is assumed to be available for a service and can be provided to DeGap. We used this approach first for analyzing gaps with file permissions using logs from `auditd`.

In the second approach, the existence of a reverse map from gaps to configuration settings is assumed to be unavailable. We only require the availability of a one-way transform from configuration settings to $P_{\mathcal{G}}$. We used this approach analyzing gaps in configuration settings in `sshd_config` using `SSHD` authentication logs and later also applied it to `auditd` logs. The second approach requires less work in applying DeGap to a new service since a reverse map from gaps to configuration settings does not have to be defined. However, the first approach can sometimes be more efficient. In this Chapter, we will primarily discuss the second approach since we have found it to work sufficiently well in practice that the extra work of creating a reverse map is probably not worthwhile.

In the second approach, to identify changes in settings that could reduce the gap, the basic idea is to generate potential deltas to a configuration file that tighten permissions and map the modified configuration files to a set of granted permissions $P'_{\mathcal{G}}$. We then determine if $P'_{\mathcal{G}}$ helps close the permission gap with respect to a set of required permissions $P_{\mathcal{U}}$.

Permission gaps can be tightened (reduced) by restricting $P_{\mathcal{G}}$ to $P'_{\mathcal{G}}$ where $P'_{\mathcal{G}} \subset P_{\mathcal{G}}$. This involves eliminating permissions. To eliminate the gap P_{Δ} completely, one must choose $P'_{\mathcal{G}} = P_{\mathcal{G}} \setminus P_{\Delta}$. Realistically, this is not always possible as there are limits on the granularity of the granted permissions that can be influenced by the configuration settings. For example, for file permissions, changing the `other` mode bits in UNIX will impact all non-owner and non-group users.

In general, removing permissions by changing the configuration settings leads to three possible outcomes: under-tightening, over-tightening, or both. Over-tightening affects usability - some accesses in $P_{\mathcal{U}}$ would have been denied. Under-tightening exposes the system to accesses that are not in $P_{\mathcal{U}}$. For a static system (where $P_{\mathcal{G}}$ has not changed over the logging period), we claim the following propositions:

Proposition IV.1. *(Over-Tightening Rule)* $P'_{\mathcal{G}}$ is over-tightened with respect to required permissions $P_{\mathcal{U}}$ if and only if $P_{\mathcal{U}} \setminus P'_{\mathcal{G}} \neq \emptyset$.

Proposition IV.2. *(Under-Tightening Rule)* $P'_{\mathcal{G}}$ is under-tightened with respect to required permissions $P_{\mathcal{U}}$ if and only if $P'_{\mathcal{G}} \setminus P_{\mathcal{U}} \neq \emptyset$.

Proof. First, let $P'_{\mathcal{G}}$ be an over-tightened permissions configuration. This is equivalent to saying that there exists some permission u that has been requested but not granted; that is, $u \in PU \setminus P'_{\mathcal{G}}$ and hence $P_{\mathcal{U}} \setminus P'_{\mathcal{G}} \neq \emptyset$.

Similarly, let $P'_{\mathcal{G}}$ be an under-tightened permissions configuration. This is equivalent to saying that there exists some permission g that is granted but never requested, so that $g \in P'_{\mathcal{G}} \setminus P_{\mathcal{U}}$ and hence $P'_{\mathcal{G}} \setminus P_{\mathcal{U}} \neq \emptyset$. □

We note that it is possible for a configuration change to result in $P'_{\mathcal{G}}$ that is both over-tightened and under-tightened with respect to $P_{\mathcal{U}}$. This could theoretically occur if the configuration change results in revocation of multiple permissions that are not sufficient to close the gap, but some of the revoked permissions are in the set $P_{\mathcal{U}}$. For Unix files, removing group permissions on a file could result in such a situation.

The over-tightening rule is a simplification of the reality. Since $P_{\mathcal{G}}$ is really a snapshot of granted permissions, there is a possibility that $P_{\mathcal{U}} \setminus P_{\mathcal{G}}$ is not empty – permissions could have been tightened during or after the logging period, because of a change in security requirements and deletion of some objects, but before a snapshot of $P_{\mathcal{G}}$ was taken. If that is the case, we want $(P_{\mathcal{U}} \setminus P'_{\mathcal{G}}) \subset (P_{\mathcal{U}} \setminus P_{\mathcal{G}})$ to hold. If true, then the configuration change contributes to reducing the permission gap between $P_{\mathcal{G}}$ and $P_{\mathcal{U}}$; otherwise not. We accommodated this scenario by *normalizing* $P_{\mathcal{U}}$ (replacing it with $P_{\mathcal{U}} \setminus P_{\mathcal{G}}$) thereby removing requests in the log that pertain to objects that no longer exist or permissions that administrators have revoked. Once $P_{\mathcal{U}}$ is normalized, the over-tightening and under-tightening rules continue to apply.

To test if a configuration setting contributes to the permission gap, we can simply simulate a change to the setting to obtain $P'_{\mathcal{G}}$, and compute $P_{\mathcal{U}} \setminus P'_{\mathcal{G}}$. If $P_{\mathcal{U}} \setminus P'_{\mathcal{G}} = \emptyset$, the setting contributes to the permission gap. On the other hand, if new tuples show up in $P_{\mathcal{U}} \setminus P'_{\mathcal{G}}$, that means that changing the setting caused some actions in $P_{\mathcal{U}}$ to be denied. The setting does not contribute to the permission gap. Additionally, we check that $P'_{\mathcal{G}} \setminus P_{\mathcal{U}} = \emptyset$. Otherwise, some of the previously granted subjects will be denied access with the changed setting.

In general, if there are n possible atomic deltas to an existing configuration setting, it will require $O(n)$ checks to identify all the deltas that can lead to tightening the gap without over-tightening. The above idea also applies to finding potentially stale members in a group. As an example, for `SSHD`, the `AllowUsers` field specifies a list of authorized users. To detect users in the list that could be contributing to a permission

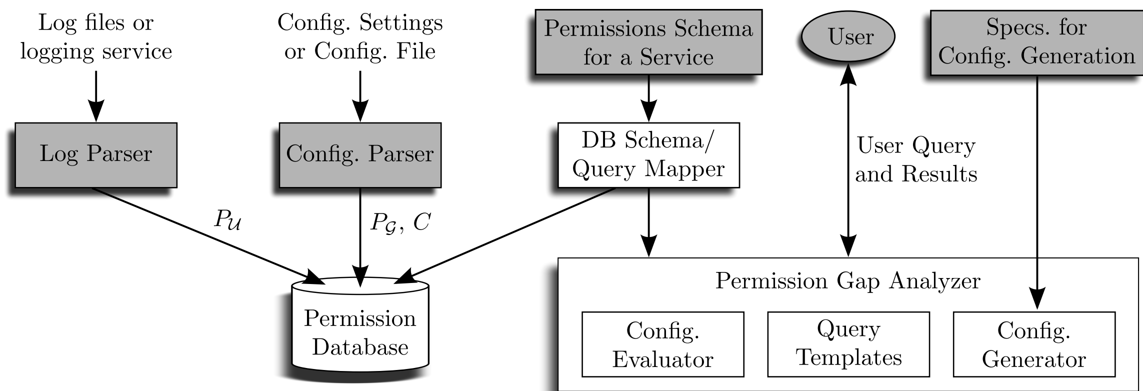


Figure 4.1: Conceptual model for DeGap. Arrows indicate dataflow. Shaded components are specific to the service being analyzed.

gap, one simply needs to simulate removing each user one by one and, for each $P'_{\mathcal{G}}$, determine if $P_{\mathcal{U}} \setminus P'_{\mathcal{G}} = \emptyset$; if yes, the user's authorization was not used for login and the user is a candidate for removal from the `AllowUsers` field. For `auditd` logs, we can apply a similar technique to periodically analyze membership lists in `/etc/group` for key groups for potential pruning.

4.5 System Architecture

4.5.1 Overview

Figure 4.1 shows the architecture of DeGap. Shaded boxes in the figure are specific to each service being analyzed (e.g., `SSHD`, file system permissions). Bulk of the gap analysis system and the database is automatically generated from schemata that describe the attributes of subjects, objects, rights, and format of the service configuration files.

A Log Parser extracts permissions used during accesses from either log files or directly from a service logging facility and system activities and puts it in a SQLite permissions database. For services, writing logs to a file is usually more efficient than writing to the database on the fly [46] and most services can log to files out of the

box. Thus, in DeGap's implementation, we adopt a hybrid approach where we use log files to record object accesses but post-process the logs to store required permissions, $P_{\mathcal{U}}$, to a database so as to permit a general-purpose query engine to help analyze permission gaps.

The Config. Parser is a service-specific component that generates the granted set of permissions $P_{\mathcal{G}}$. For services where configuration files are used to define granted permissions (e.g., `SSHD`), it reads in configuration files and generates $P_{\mathcal{G}}$. For others, like file system permissions on our Linux servers, it extracts data from the file permission mode bits. For configuration files, it also extracts a sequence of configuration settings, C, that are used by another component, Config. Generator, to help discover settings that could be changed to reduce permission gaps.

The DB Schema/Query Mapper is similar in spirit to tools like Django [10] in that it takes as input a schema and generates the initial tables for the Permissions Database. We also use it to generate SQL queries from *query templates*. Query templates provide simple ways of querying the database using select-project-join (SPJ) queries [38] on entities and attributes in the configuration schema. SPJ queries are highly expressive though not full SQL; full SQL is also available. Users who wish to extend DeGap or make custom queries can either use templates or full SQL. Other key components of the gap analysis system, Config. Generator and Config. Evaluator, internally make use of templates as well as direct queries on the database, wrapped in Python code.

Config. Evaluator takes as input candidate configuration file (or settings) that a user would like to evaluate against $P_{\mathcal{U}}$, the requests in the log files. It reports back whether the candidate configuration leads to a system that has a narrower gap than $P_{\mathcal{G}}$ without over-tightening.

Config. Generator uses specifications for configuration parameters (described in Section 4.5.4 and in Figure 4.5) as input and generates alternatives for configuration

settings that are tighter than the current settings. It then evaluates those modified settings using the Config. Evaluator to determine whether they tighten the permissions without over-tightening them. The subset of alternatives that are acceptable are presented to the user. Config. Generator can also apply a greedy algorithm (presented in Section 4.5.4) to generate a sequence of configuration setting changes that is *maximal*. In other words, tightening any remaining setting further leads to an over-tightened system.

4.5.2 Principles

The architecture and its mapping to an implementation reflects certain design principles:

- *Build once, reuse many times:*

  The DB Schema/Query Mapper and the Config. Generator segregate the logic of DeGap from the semantics of the logs and configurations for the service being analyzed. These components are generic and can be used across different services being analyzed.

- *Minimalistic Design:* To amortize the cost of gathering and writing the data, logs tend to contain as much of the data as possible. However, much of the data is usually unneeded for the computation of permission gaps . Moreover, many accesses are essentially identical to earlier accesses, except for their timestamps. We aim to keep the database model simple, the size of database small, and to provide a simple language for making queries related to gap analysis.

- *Handle scale by appropriate optimizations:* The set $P_{\mathcal{U}}$ can be very large in practice. We use optimizing for doing gap analysis on large sets so that $P'_{\mathcal{G}}$ usually does not have to be compared with all entries in $P_{\mathcal{U}}$.

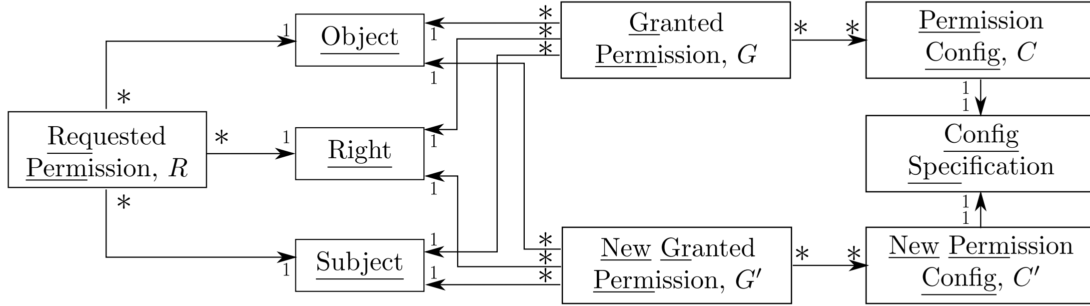


Figure 4.2: Database model for DeGap. Underlined substrings are used as table names.

4.5.3 Database Model

Figure 4.2 summarizes the database model used for DeGap. The model follows from our definitions in Section 4.4. A object is accessed by a subject using an access right. An subject may access a object using multiple access rights while a object may be accessed by multiple subjects using different access rights. Each granted permission is associated with a subject, an access right, and a object. The "Granted Permission" and "Used Permission" tables correspond to $P_{\mathcal{G}}$ and $P_{\mathcal{U}}$ respectively. The "New Granted Permission" table has the same attributes as the "Granted Permission" table, but stores the set of granted permissions $P'_{\mathcal{G}}$ for a modified set of configurations C', which we will elaborate shortly in Section 4.5.4. The reason for storing both $P_{\mathcal{G}}$ and $P'_{\mathcal{G}}$ is to allow the user to evaluate the potential permission gaps as a consequence of a new set of configurations.

The "Permission Config" and "New Permission Config" tables are used for storing the configurations used for determining $P_{\mathcal{G}}$ and $P'_{\mathcal{G}}$. More than one configuration may affect a certain permission, and each configuration may affect multiple permissions.

The "Config Specification" table describes the type, possible values, and default value for each configuration. They are used for generating a new set of configurations C' and will be discussed further in Section 4.5.4.

Figure 4.2 does not show the attributes within each table (e.g., Object, Right, and

Subject). Those are service-specific. For example, `SSHD` will have different attributes than a filesystem. These tables are generated by the DB Schema/Query Mapper from the schema for the service's permissions.

4.5.4 Permission Gap Analyzer

For DeGap to be useful, users should be able to evaluate the effect on permissions when a configuration setting is changed. Towards this end, the Permission Gap Analyzer (PGA) provides the following capabilities:

- *Evaluate a proposed configuration setting change:* The gap analyzer uses Over-Tightening Rule and Under-Tightening Rule (see Section 4.4) to help evaluate a proposed configuration setting with respect to $P_{\mathcal{U}}$. Over-tightening of permissions could be a concern since there could be potential denial-of-service to legitimate users.

- *Generate a list of candidate configuration changes:* The gap analyzer uses the specs for configuration generation to automatically iterate through possible one-step changes to configuration settings and identify the settings that can be tightened to reduce the gap.

- *Generate a sequence of configuration changes:* The gap analyzer can identify a full *sequence* of changes to a configuration file that provides a maximal solution to gap reduction. The solution is maximal and not necessarily optimal, in the sense that any additional tightening of settings in the configuration file will lead to over-tightening. But, there could be other sequence of changes that are longer in length or result in lower gap with respect to some objects. In general, finding an optimal sequence is likely to require exponential time as given n possible individual tightening steps, there can be 2^n combinations of tightening steps. We therefore focus on finding a maximal sequence rather than

Require: Configurations C'
Require: Granted permissions $P_{\mathcal{G}}$
Require: Used permissions $P_{\mathcal{U}}$
  $P'_{\mathcal{G}}$ = Permissions granted by configuration C'
  if $P'_{\mathcal{G}} \subseteq P_{\mathcal{G}}$ and $P_{\mathcal{U}} \subseteq P'_{\mathcal{G}}$ **then**
    $SaveResults(C', P'_{\mathcal{G}})$
    if $(P'_{\mathcal{G}} \setminus P_{\mathcal{U}}) \equiv \emptyset$ and $(P_{\mathcal{U}} \setminus P'_{\mathcal{G}}) \equiv \emptyset$ **then**
      return *tightest* {Can't tighten further}
    end if
    return *tighter*
  end if
  return *bad*

Figure 4.3: Algorithm for Config. Evaluator, E.

an optimal solution. Given that we are working with estimates of permission gaps, we believe this is a reasonable and practical strategy.

In the algorithms that follow, we use $P_{\mathcal{G}}$ to refer to granted permissions, $P_{\mathcal{U}}$ to refer to required permissions (these are derived from logs), and $P'_{\mathcal{G}}$ to refer to a candidate for granted permissions. C refers to the combination of configuration settings that result in $P_{\mathcal{G}}$. C' refers to the combination of configuration settings that result in $P'_{\mathcal{G}}$.

We discuss each of the above features of permission gap analyzer below. Gap analyzer also permits users to make direct queries on the database via SQL of via query templates. We will discuss those in Section 4.5.5 briefly. They are a convenience feature of the system to support specific queries on the permissions database and to allow extensibility of the tool.

4.5.4.1 Config. Evaluator

The Config. Evaluator E takes in C', $P_{\mathcal{G}}$, and $P_{\mathcal{U}}$, then computes the permissions $P'_{\mathcal{G}}$ that would be granted by configuration C'. It compares $P'_{\mathcal{G}}$ with $P_{\mathcal{G}}$ and $P_{\mathcal{U}}$, and returns three possible results: `tighter`, `tightest`, and `bad`. C' is `bad` if $P'_{\mathcal{G}}$ is not a subset of $P_{\mathcal{G}}$, or at least one permission in $P_{\mathcal{U}}$ is denied by $P'_{\mathcal{G}}$. Otherwise, $P'_{\mathcal{G}}$ is tighter than $P_{\mathcal{G}}$. To check if $P'_{\mathcal{G}}$ is the tightest possible set of permissions, in addition

Require: Configurations C
Require: Granted permissions $P_{\mathcal{G}}$
Require: Used permissions $P_{\mathcal{U}}$
  for all $c_j \in C$ **do**
    ret = ok
    while $((c'_j = restrict_setting(c_j))$ is valid)
      and $(ret \neq tightest)$ **do**
      $C' = (C - \{c_j\}) \cup \{c'_j\}$ {Replace c_j with c'_j}
      $ret = ConfigEvaluator(C', P_{\mathcal{G}}, P_{\mathcal{U}})$
      {Also saves good results to database}
      if $ret \neq bad$ **then**
        $C = C'$
      end if
      $c_j = c'_j$
    end while
  end for

Figure 4.4: Greedy Algorithm for Discovering a Maximal Patch.

to the two conditions, we check that $P'_{\mathcal{G}}$ is not being over-tightened and not being under-tightened, i.e., $P'_{\mathcal{G}} \setminus P_{\mathcal{U}} \equiv \emptyset$ and $P_{\mathcal{U}} \setminus P'_{\mathcal{G}} \equiv \emptyset$. The algorithm for E is shown in Figure 4.3.

4.5.4.2 Automatically Generating Alternative Configurations

To assist PGA in generating alternative configurations, a user specifies the format of a configuration file, along with choices for each field, as shown in Figure 4.5. The current version of the system views configuration files as a sequence of fields, where each field can either have (i) a single value from a domain or a set of values, or (ii) a sequence of values from a domain. This can obviously be generalized to allow nested fields, but is adequate as a proof-of-concept.

Each specification has three parts: `type`, `values`, and `default`. Type is specified as either `oneof` or `setof`. We found these two types sufficient for analyzing permissions for `SSHD`, `auditd`, and user groups, but more types can be easily added. If `type` is `oneof`, the `values` part is mandatory. It specifies the possible values for the configuration in increasing order of tightness and is delimited by '—'. In Figure 4.5,

```
field = {type = <type>,
         values = <value 1> | <value 2> | ... | <value n>,
         default = <default>}

PermitRootLogin = {type = oneof,
                   values = yes | without-password |
                            forced-commands-only | no,
                   default = yes}

AllowUsers = {type = setof(subject),
              default = *}
```

Figure 4.5: Configuration specification format and examples for `PermitRootLogin` and `AllowUsers` for `SSHD`.

`PermitRootLogin` is specified as having `type oneof` and can take four `values`, `yes`, `without-password`, `forced-commands-only`, and `no` in order of increasing strictness. It is specified to have a default value of `yes`, thus, if `PermitRootLogin` is not present in the `SSHD` configuration file, the value `yes` will be used.

If the `type` is `setof`, the configuration takes a set of values, such as `AllowUsers` for `SSHD`. These configurations typically specify a range of values for two categories, subjects, or objects. Specifying a range of values for access rights is rare but possible. If a configuration affects only a subset of a certain category, computing permission gaps for permissions excluding those affected by the category is redundant. Towards optimization, we allow the user to augment the `type` with either `subject`, `object`, or `right`. Using this annotation, PGA pre-filters the permissions for the specified category for computing operations such as $P_{\mathcal{G}} \setminus P'_{\mathcal{G}}$. We will elaborate on `type setof` in Section 4.5.4.2.

The Config. Generator is used for tightening a single configuration setting from a given state. It provides a Python method $restrict_setting(c)$ that generates a more restricted value for a field c. Repeated calls to the function return subsequent choices for that field (when choices are exhausted, the function returns None, which is equivalent to False in conditionals in Python). This function is used to automatically find

the list of all possible configurations that only tighten a setting as well as a maximal solution of a sequence of configuration tightening steps using a greedy algorithm. The function *restrict_setting*(c) works as follows for the two types of fields that are currently supported:

Type `oneof` fields Using the Configuration Generation Specification, *restrict_setting*(c) returns the next (tighter) value for a field c in a configuration. If a configuration `type` is `oneof`, the generator returns the value that follows the current one. For example, if the current value for `PermitRootLogin` is `without-password`, then the function on the first call returns `forced-commands-only`, on the second call returns `no`, and finally returns Python's `None`.

Type `setof` fields If the configuration `type` is `setof`, each time *restrict_setting*(c) is called to restrict a set, it simply removes one element in the set that has not been previously removed. For example, if `AllowUsers` were "user1, user2, user3", it would generate the following sequences: "user2, user3", "user1, user3", "user1, user2" and Python's None. Note that it only removes one value from the current list and does not generate all subsets. So, the procedure is linear in the size of the set.

In practice, sets can have special values such as `*`, which denote all possible values from a domain that are difficult to enumerate and apply the above strategy. For example, in `sshd_config`, if `AllowUsers` is missing, the default value for that can be considered to be `*`. The question then is how we represent $P_{\mathcal{G}}$ and generate alternatives for tighter configurations when the values of an attribute cannot be feasibly enumerated? One approach that we considered using is suggested in the SPAN system [49] where internally security policies are using binary decision diagrams and a SQL front-end is provided to the users for querying. A limitation of SPAN is that it requires all attributes to converted to integer ranges and there were concerns from a scalability perspective.

To address the problem of handling field values that are difficult to enumerate, we use a projection on $P_{\mathcal{U}}$ on the appropriate domain as the initial recommendation for the set of all granted permissions. There is no loss of generality since DeGap is concerned with reducing $P_{\mathcal{G}}$ towards $P_{\mathcal{U}}$. For example, if `AllowUsers` is missing (equivalent to `*`) and two subjects `user1` and `user2` are the only ones who successfully logged in, the tool will recommend narrowing down `*` to the set $\{user1, user2\}$.

4.5.4.3 Greedy Algorithm for Discovering a Maximal Patch

The Config. Generator includes a greedy algorithm to compute a maximal patch to a configuration automatically. This is achieved by iteratively restricting each configuration and testing if the new set of granted permissions is a subset of the original one without rejecting any required permission. The algorithm is shown in Figure 4.4.

4.5.4.4 Improving Performance of Algorithms for Large Logs

In practice, the set $P_{\mathcal{U}}$ can be large. It can be important to efficiently compute $P_{\mathcal{U}} \setminus P'_{\mathcal{G}}$ without having to go through every required permission in the database. Our implementation uses the following intuition to improve the performance. We first attempt to identify $P_{\mathcal{G}} \setminus P'_{\mathcal{G}}$, the set of permissions that were revoked in going from $P_{\mathcal{G}}$ to $P'_{\mathcal{G}}$. We then do a database projection of this difference on both objects and subjects. This gives us the objects and subjects that can be impacted by the change. We then only compute $P_{\mathcal{U}} \setminus P'_{\mathcal{G}}$ on the rows in $P_{\mathcal{U}}$ for which the subject and objects are in the projection of $P_{\mathcal{G}} \setminus P'_{\mathcal{G}}$.

As an example, if the change in going from $P_{\mathcal{G}}$ to $P'_{\mathcal{G}}$ is to remove `user1` from `AllowUsers` field, then the projection of $P_{\mathcal{G}} \setminus P'_{\mathcal{G}}$ on subjects will result in the set `user1`. In this case, the only permissions from $P_{\mathcal{U}}$ that are going to be relevant for computing $P_{\mathcal{U}} \setminus P'_{\mathcal{G}}$ are those for which the subject is `user1`.

```
Object.path = ?
Object.type = (LIKE,"%private key%")
ReqPerm = ?
Access.label = (=,"other-read")
```

Figure 4.6: An example of a query for `auditd` for the paths of files having types that match SQL pattern "%private key%" and have been read by others as indicated by "other-read".

```
General Form
distinct = [yes|no]
count = [?|(<cmp>,i)]
<table 1>.<attrib 1> = [?|<constraint expression>]
     .
     .
<table m>.<attrib n> = [?|<constraint expression>]
```

Figure 4.7: General form of a query. The comparison operator `<cmp>` can be either $<$, $<=$, $>$, $>=$, $<>$, $!=$, or $==$.

4.5.5 DB Schema and Query Mapper

The DB Schema and Query Mapper loads a user query, then extracts tables and their attributes from the database, and dynamically generates an SQL query before submitting it to the database. A significant advantage of the Query Mapper is that the user does not have to manually craft SQL queries for basic uses. For our experiments, we found that the Query Mapper suffices for investigating permission gaps; however an advanced user may choose to query the database directly if it is required.

A user can make three kinds of queries. The first kind of queries are SPJ queries and are made by the Query Mapper on the SQL database directly. The second kind of queries are gap analysis queries. The third kind of queries are configuration change queries.

SPJ queries allow users to formulate their own queries, thus they can leverage their domain knowledge to search for permissions granted to certain objects. The user simply specifies constraints using pairs having the format `(<operator>, <constraint`

$$
\begin{aligned}
\langle atom \rangle & ::= & (\langle operator \rangle, \langle operand \rangle) | (\langle expr \rangle) \\
\langle expr \rangle & ::= & \langle atom \rangle \langle operator \rangle \langle expr \rangle | \langle atom \rangle \\
\mathcal{C} & ::= & \langle expr \rangle
\end{aligned}
$$

Figure 4.8: BNF for constraint expression $\mathcal{C}$. The $\langle operator \rangle$ refers to one of the SQL operators, while the $\langle operand \rangle$ refers to a user-specified value, typically an integer or a quoted-string.

value>), where <operator> is one of SQL's comparison operator. For example, if a user knows that files having types containing keywords "private key" may be possible targets for attackers, she may make a query using the query template shown in Figure 4.6 to list all the other-read requests on files matching "%private key%" for further analysis. Figure 4.7 shows the general form of such queries. A query comprises key-value pairs, where the key is either one of the keywords or an attribute in a table. The keywords are count and distinct. The count keyword can be either '?' that indicates a query for the number of results, or a pair (<cmp>, i) where <cmp> is a comparison operator used for evaluating the number of results against an integer i. The distinct keyword removes duplicates in the results. An attribute is specified with its name preceded by its table name. Its value can be either '?' or a constraint expression using the format specified in Figure 4.8. An advantage of using SQL's operator is that rich SQL features such as regular expressions are allowed in the queries. For example, the user may feel that all files with filename containing "passwd" pose great security risks if their permissions are weak; using a prepared query, it is straightforward for the user to restrict their search for permission gaps to such files.

Gap analysis queries pertain to asking questions regarding permission gaps and how they may be reduced. There are two kinds of gap analysis queries that we have found useful. Firstly, given a specific change to a configuration setting, a user can ask what are the permission gap changes with respect to $P_{\mathcal{U}}$. This is helpful for a user

who is deciding if a change should be made. The Query Mapper may inform the user that the permission gap is (i) under-tightened, i.e., there are subjects who are granted permissions and did not previously access the object, (ii) over-tightened, i.e., there are subjects who were previously granted access but were denied given the change, or (iii) no permission gap, i.e., all subjects who were previously granted permissions and accessed the object still have the permissions, and all subjects who did not access the object are now denied.

The second gap analysis query we found useful is that the user can request for a list of possible one-step changes to the configuration settings that lead to reduction in the permission gap without over-tightening. There may be different ways to reduce permission gaps. For example, for `SSHD`, to disallow root login, the user can either set `PermitRootLogin` to “no”, exclude root from `AllowUsers`, or both. The user may use the result to selectively modify the configuration file. One can think of this type of one-step analysis to have the same semantics as a breadth-first-search for a set of tightened configurations.

The last kind of queries, the configuration change queries, return a set of configuration settings that can help reduce the permission gaps without over-tightening. This kind of queries provide a list of sequence of changes to the configuration settings that help reduce permission gaps without over-tightening. This uses the results from the Greedy Configuration Speculator discussed in Section 4.5.4.

The objective of dynamically creating an SQL query is to use the minimal tables needed to fulfill the query. A naïve approach of using all tables will erroneously introduce constraints on tables that are not intended to be queried. The Query Mapper creates a query dynamically by first generating a tree of the tables rooted at the “Object” table, where each node corresponds to a table and each edge corresponds to one or more relationships between two tables. A breadth-first search is used for this purpose. To prevent cycles, the “Right” and “Subject” tables will always be leaf

nodes. The Query Mapper then finds the paths to all the tables that will be queried. All tables along the paths are then included in the query.

4.6 Evaluation

We implemented DeGap in Python and used SQLite for the database for detecting permission gaps in `SSHD`, `auditd`, and user groups. The three scenarios have vastly different ways for specifying permissions. File permissions are specified in the file system's inode structure and retrieved using system calls. On the other hand, `SSHD` configurations and user groups are stored in configuration files. Also, the types of configuration values differ, i.e., binary versus ranges. Additionally, for `SSHD`, certain configuration settings, such as PermitRootLogin and AllowUsers, interact to determine if a certain permission should be granted.

Approximately 382 lines of code were required for the modules used for for analyzing both `SSHD` and `auditd`. Additionally, `SSHD` required 440 lines while `auditd` required 414 lines. In other words, 46.5% of `SSHD`'s code and 50.0% of `auditd`'s code were re-used.

4.6.1 Case Study: `SSHD`

In this section, we describe our experiences with analyzing `SSHD` logs using DeGap. We considered `SSHD` as a single object and subjects to be users attempting to connect with the service. Access rights had a `method` attribute that specified if either password-authentication or public-key authentication was used. The permissions were set in a configuration file found at `/etc/ssh/sshd_config`. For ease of discussion, we will focus on the fields `PermitRootLogin`, `PubkeyAuthentication`, `PasswordAuthentication`, and `AllowUsers`. Their use was inferable from our `SSHD` logs.

We evaluated the `SSHD` logs from two machines in our department. The servers

```
PermitRootLogin = {type = oneof,
                   values = yes | without-password |
                            forced-commands-only | no,
                   default = yes}

PubkeyAuthentication = {type = oneof,
                        values = yes | no,
                        default = yes}

PasswordAuthentication = {type = oneof,
                          values = yes | no,
                          default = yes}

AllowUsers = {type = setof(subject),
              default = *}
```

Figure 4.9: Configuration generation rules used as input to DeGap.

were used as a source code repository and a web-server for hosting student projects for a class. Since accesses to `SSHD` are logged by default, we did not have to make any changes to the system.

4.6.1.1 Analyzing Permission Gaps

We now examine the application of DeGap towards analyzing the permission gaps.

For both case studies, we used the configuration generation rules specified in Figure 4.9 for hinting to the Config. Generator how the values for each field should be tightened. `PubkeyAuthentication` and `PasswordAuthentication` could be either `yes` or `no`. If `without-password` was used, root would not be able to login using a password. If `forced-commands-only` was used, only public-key authentication would be allowed for root. `PermitRootLogin` and `AllowUsers` were given as examples in Section 4.5.4.2 and will not be discussed here.

Server 1 Figure 4.10 shows the configurations used by `SSHD` on the first server. Four users, including root, were allowed to access the systems (Note: userids are

```
PermitRootLogin        yes
AllowUsers             user1 user2 root user3
PubkeyAuthentication   yes
PasswordAuthentication yes
```

Figure 4.10: Partial configurations used by `SSHD` for Server 1.

```
PermitRootLogin        without-password
AllowUsers             user1 root
PubkeyAuthentication   yes
PasswordAuthentication no
```

Figure 4.11: Tightened partial configurations for Server 1 as suggested by DeGap after running it against the logs and configuration files.

anonymized). Both the `publickey` and `password` authentication methods were allowed. As specified by `AllowUsers`, four users, user1, user2, user3, and root were allowed to connect to the two systems via `SSHD`.

We instructed DeGap to automatically compute possible tightest configurations. The set of configurations suggested by DeGap is shown in Figure 4.11. We manually verified the results using traditional tools such as `grep` to search for keywords such as "Accepted" in the log files. Only user1 and root did access the server, suggesting that permission creep had occurred for two users. Upon checking with the owner of the server, we confirmed that they were student instructors for a course more than a year ago and had been granted the permission to access the server. However, after the end of the semester, these students were not removed from `AllowUsers`. All users only used public-key authentication. DeGap correctly suggested that three configurations should be tightened: `PermitRootLogin` should be set to `without-password` instead of `yes`, `AllowUsers` should include only user1 and root, and `PasswordAuthentication` should be disabled by setting its value to `no`.

Server 2 Figure 4.12 shows the `SSHD` configurations used by Server 2. All three options shown in Figure 4.12 were found to be `yes`. Also, the `AllowUsers` field

```
PermitRootLogin        yes
PubkeyAuthentication   yes
PasswordAuthentication yes
```

Figure 4.12: Partial configurations used by `SSHD` for Server 2.

```
PermitRootLogin        no
AllowUsers             user1 user2
PasswordAuthentication yes
PubkeyAuthentication   yes
```

Figure 4.13: Tightened partial configurations for Server 2 as suggested by DeGap after running it against the logs and configuration files.

was not specified. This implied that anyone could connect to Server 2 using either password or public-key authentication over ssh.

DeGap suggested the configurations for Server 2 shown in Figure 4.13. We can observe that the `AllowUsers` field was recommended to be specified with user1 and user2. The `PasswordAuthentication` and `PubkeyAuthentication` field were shown to retain their original values of "yes". Again, we were able to manually verify that user1 and user2 did access Server 2, with user1 using the public-key authentication method, while user2 used the password authentication method.

In summary, we observe that DeGap correctly discovered the changes to configuration setings for each server to reduce their respective permission gaps. For Server 1, an unused authentication method was proposed to be disabled. For Server 2, a much stricter setting for `AllowUsers` having only two users were suggested.

4.6.1.2 Log Size vs. Permission Database Size

Table 4.1 compares the number of lines in the `SSHD` logs and the corresponding number of tuples for the Subject, Right, and Object table. Comparing the large number of log entries and relatively smaller number of tuples in the Right table, there was a large number of duplicated accesses, many due to password brute-force

Table 4.1: Comparison between number of entries in log file and number of tuples in database for `SSHD`.

Server	# of Log Entries	# of tuples in DB Table			
		Subjects	Rights		Objects
			Success	Failure	
sshd-server1	579,788	74	32	73,395	1
sshd-server2	262,223	80	44	49,213	1

attacks. For example, for sshd-server1, 73,395 accesses that failed using the password authentication method originated from 66 subjects. Except for one, the remaining 65 IP addresses do not belong to our university's network. The savings in space was reflected in the log sizes, where the ratio of the log sizes to the database sizes were 8.31 and 6.47 for `sshd-server1` and `sshd-server2` respectively.

4.6.2 Case Study: `auditd`

In this section, we describe the application of DeGap towards finding file permission gaps using logs generated by `auditd`. We examined the `auditd` logs, which were collected over seven days on 17 departmental servers at our school. The servers were installed with Fedora 15 with patches regularly applied. Over the period, for each server on the average, we collected over 12 million file open and execute events for over 345,000 files by over 200 users. `Auditd` is the primary means used by administrators to collect accesses to system objects for Linux kernel 2.6. Logs generated by `auditd` are often scrutinized during security audits for unauthorized accesses. For this data collection, we added a rule for recording `execve` and `open` syscalls on files, "`-a exit,always -F arch=b64 -S execve -S open`", to `auditd`'s rule file. The analysis was performed on two identical machines, both having two Dual-Core AMD Opteron 2218 Processors and 16GB RAM. We present the results below.

We examined two syscalls, `open` and `execve`. A successful call to `open` indicates that the calling process has acquired the `read` and `write` access rights specified by

the open flags (O_RDONLY, O_WRONLY, O_RDWR). While one could log other syscalls such as read and write to ascertain the actual accesses carried out on a object, the open flags already circumscribe these access rights. Moreover, logging these syscalls leads to expensive disk operations.

Leveraging Query Language Regex We performed a query that leverages the query language's regular expression matching capabilities. We searched for files with types matching the SQL pattern "%private key%" using the query shown in 4.6. On all the machines, we found that the RSA private keys for Dovecot, a popular IMAP and POP3 server, on all analyzed servers were world-readable. We verified this by checking Dovecot's SSL configuration file, which is also world-readable. If an attacker obtains the key, she can generate a fake certificate and launch a man-in-the-middle attack. In fact, it is highlighted on Dovecot's SSL configuration page that no users, except root, require access to the key file [119]. The administrator agreed that the other read permission should be removed.

Dealing with File Semantics Analyzing all file accesses may result in DeGap suggesting a large number of files with permission gaps, and consequently unnecessarily complex results. To keep the list of files manageable, DeGap only considered successful syscalls. Also, we ignored files in /dev and /proc. The /dev directory stores device files for all devices, while the /proc directory contains virtual files, which are constantly updated with the system's information. In addition, we ignored symbolic links, since the access granted to the target of a symbolic link ultimately depends on the permissions set on the target itself.

Deriving Used File Permissions for Unix Semantics Each entry for an access on a file in an auditd log must be mapped to a form so that it is amenable for comparison with granted permissions, as defined by the Unix permission mode bits.

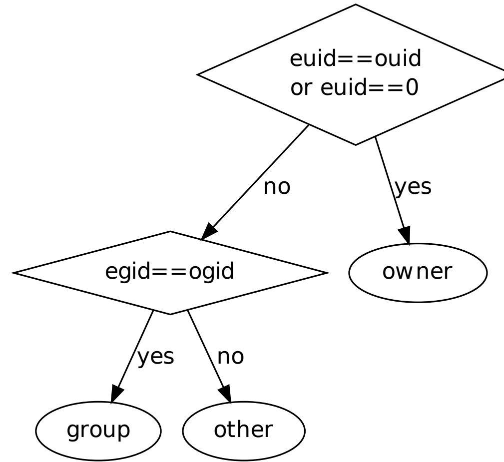


Figure 4.14: Decision trees for determining file role type, i.e. owner, group, or other.

This is tricky to do since an entry contains information about the userid and the type of access, but not how that relates to a permission mode bit.

Each access on a file was analyzed by first determining the role types, `owner`, `group`, or `other`, using the decision trees in Figure 4.14. The required permission bit for either `read` or `write` access can be computed by checking the last two bits of the `open` syscall's flags against the access modes O_RDONLY, O_WRONLY, and O_RDWR. For executables, the `execute` permission bit is required if the file is accessed using the `execve` syscall.

It is also necessary to determine if the `suid` or `sgid` bit is required. If the `ouid/ogid` is the same as the `euid/egid` but the `euid/egid` differs from the syscall's `uid/gid`, then the `suid/sgid` bit is required. This also implies that the `suid/sgid` bit in the file's mode must be set. Otherwise, it is the scenario where the process calls the `setuid()/setgid()` function to lower its own privileges. This typically occurs during system boot and the process relinquishes its root privileges as a security measure against an exploitable vulnerability being used by an attacker for privilege escalation.

DeGap does not need to analyze an owner's used permissions, since a permission gap with respect to an owner is somewhat meaningless with Unix's discretionary

```
distinct = yes
count = ?
Right.description = "other-write"
GrPerm.has_gap = yes
```

Figure 4.15: Query used for finding the number of files that have permission gaps for other-write permissions. The query for other permissions can be achieved by changing the operand for `Op.description`.

access control model; an owner can always acquire additional rights to their objects.

Aggregating Used Permissions for Related Files We sometimes found a need to treat a collection of files as a single object, rather than as individual objects, for the purpose of gap analysis. As one example, In the case of Apache web server, there is usually a directory `cgi-bin` that contains executable web content. Normally, all files in that folder are intended to be viewed as a collection with files having identical permissions. We leverage SQL's regular expression to allow a user to treat a collection of files as one object by by specifing a regular expression pattern on full file pathnames; all matching files are then treated as one object. Typically, the aggregation is over files of the same type (e.g., *.cgi) within the same directory (e.g., /var/www/cgi-bin). A log entry for one file is attributed to all the files covered by the aggregation. The net result is that gap analysis will show a common gap for all the files that are part of an aggregation.

File Permission Gaps In this part of the analysis, we take a macro view on the permission gaps. Using the query shown in Figure 4.15, we were able to identify a large number of files with permission gaps.

Figure 4.16 shows the number of files and directories that have the original read, write, or execute permission set with the number of files whose corresponding permissions were actually used. For files, respectively, only 0.592%, 0.00432%, and 2.59% of the group read, write, and execute permissions were actually used. More importantly,

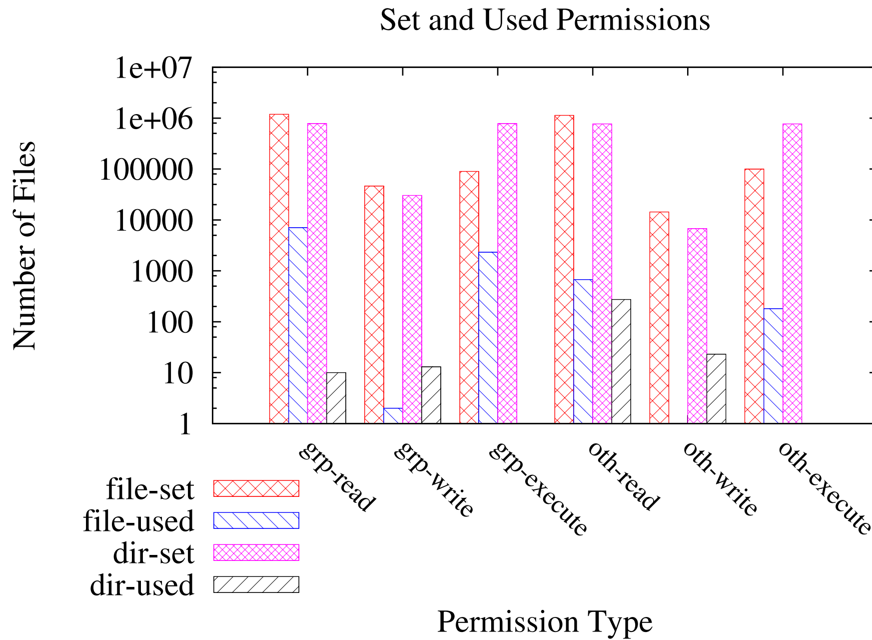


Figure 4.16: Number of files and directories with permissions set and actually used. Note the log-scale used for the number of files and directories.

only 0.0592% and 0.181% of the other read and execute permissions were used. No writes by other users were observed.

As for directories, the percentages for group read (list), group write (modify), other read, and other write were 0.0128%, 0.0429%, 0.0354%, and 0.341% respectively.

World-writable files presented a huge security risk since an adversary could easily modify the files while violating the non-repudiation principle. While there were 14,348 world-writable files, fortunately they belonged to only 17 users. These files include research data, papers, consultation reports, honor code violation report, class teaching material, exams, research proposals, and correspondent information.

The risks of world-readable files are not necessarily less than those of world-writable files. In total, we observed 1,133,894 world-readable files. Based on filenames in the auditd log, some of the user files potentially contained sensitive information such as visa applications, passwords (both hashed and plaintext), and emails. For one user, while the inbox was not world-readable, the outbox was.

We applied DeGap towards generating configuration settings for reducing the permission gaps for files. DeGap was able to propose the correct suggestions for reducing the permission gaps.

While it may be overly idealistic to remove every unused permission, the huge discrepancies in the number of files whose permissions were set and actually used illustrate the enormous potential for some of these redundant permissions to be removed. The large number of files may lead to manageability issues. The list can be fed into another system capable of extracting semantic information about the files to rank files according to potential criticality from access control perspective.

4.6.3 Case Study: Tightening `/etc/group`

The `passwd` and `group` files in the `/etc` directory are used to manage users on Unix-based OSes. We now discuss how DeGap can be used to identify dormant groups, which can possibly be removed from `group`, during the monitoring period. DeGap can also be used to tighten `passwd`. However, this is less interesting and can be achieved easily using other means. Thus, we will not discuss DeGap's usage for the purpose of tightening `passwd`.

We considered a group to be dormant if it was not used to access files. We used the same logs in Section 4.6.2 as inputs[1]. However, we re-defined the object to be a collection of all the files (instead of each file being a single object). Also, there is only a single access (instead of read, write, and execute accesses).

The `group` files for all 17 servers were the same. This implied that a truly dormant group must be dormant across all servers. Thus, after finding the dormant groups for each server, we computed the intersection of these groups to determine a set of dormant groups for all servers. From this set, we removed groups having system users as members, since such groups are less likely to be susceptible to permission creeps.

[1] While other logs, e.g., `SSHD` logs, may allow us to estimate active users, they may not provide sufficient information for determining whether a certain group is dormant.

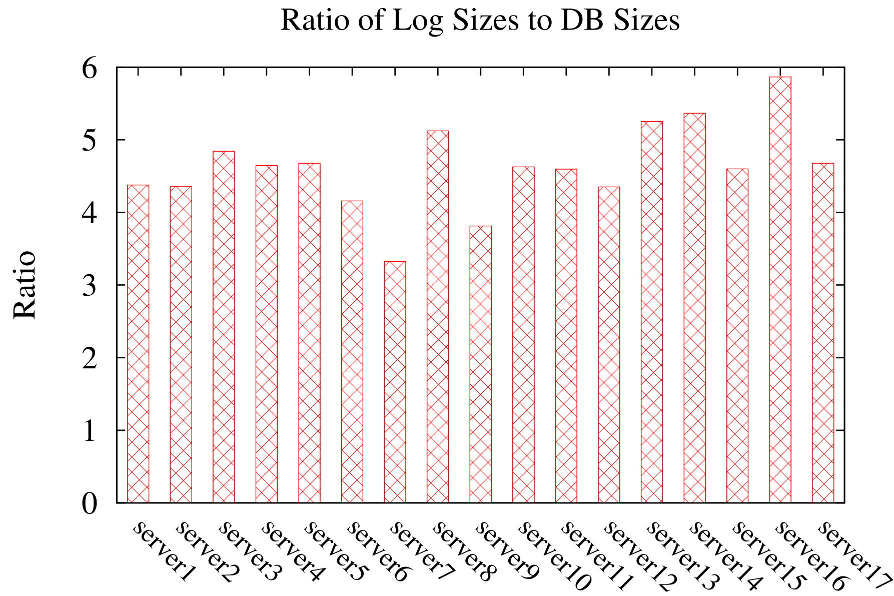


Figure 4.17: Ratio of log sizes to database sizes for `auditd`.

Table 4.2: Comparison between average number of entries in log files and average number of tuples in databases for `auditd`.

# of Log Entries	# of tuples in DB Table			
	Subjects	Rights		Objects
		Success	Failure	
12,607,326	215	5,454,845	136	345,669

We identified system users as those whose home directories are not `/`, `etc`, `bin`, `sbin`, or `var`. 526 out of 565 groups were found to be dormant during monitoring. While it may be possible that there are false positives, as we acknowledged earlier to be a limitation of log-based approaches, it provides a starting point for administrators to tighten the `group` file.

4.7 Improving Log Parser Performance

Making queries on the permissions database and running gap analysis was relatively fast. Most queries were interactive, with slowest ones taking up to a minute.

Table 4.3: Timings for various benchmarks when `auditd` is activated (`auditd-def`), and when the modified kernel for improving `auditd` is used (`auditd-degap`).

	`auditd-def`	`auditd-degap`
blogbench (writes, more is better)	577.5	574
blogbench (reads, more is better)	6956	7401.66
apache-1.4.0 (requests/sec, more is better)	14623.58	25517.87
postmark (total time in secs, less is better)	124.67	124.83
postmark (transaction time in secs, less is better)	56.33	55.83
postmark (read Mb/sec, more is better)	50.84	50.77
postmark (write Mb/sec, more is better)	146.38	146.19

In contrast, parsing the log files and populating the database for `auditd` took almost an hour. The main reason is that log files generated by `auditd` were large, even when restricted to only file accesses of interest (`open` and `execute`). To determine if there were opportunities to speed up the parsing, we examined the possibility of modifying `auditd`'s kernel code to generate more efficient log files and report the findings here.

We observed that many applications tended to make the same accesses to the same objects repeatedly. For example, some log entries differed only in their timestamps. While having such fine-grained information may be useful under some situations, such as analyzing the sequence of events that leads up to an intrusion, it is not useful for discovering permission gaps.

We modified the Linux kernel's `auditd` code to check and eliminate most duplicate entries at event capture time. We computed the SHA256 of an entry's fields that were not used for computing permission gaps and checked that the entry had not been previously logged against a bounded buffer that recorded previous unique events. If the entry's hash was found to be previously logged in the bounded-buffer, the entry was discarded and not written out to the logs. We used a two-level prefix tree for storing the hashes. The first two bytes were used for determining the branch to be taken towards the leaf that stores the remaining hash data. The clock algorithm [35] was adapted to evict old entries from the bounded-buffer.

We compared the average number of entries in the log files to the average number of tuples for Subject, Right, and Object tables as shown in Table 4.2, using a bounded-buffer for 327,680 hashes. The ratios of the log sizes to the database sizes are shown in Figure 4.17. On average, the logs were found to be approximately five times larger than the databases for our datasets. If logs from a modified `auditd` were used, the cost of parsing the logs could potentially be cut by approximately 80%, given the smaller size of the logs.

To examine the performance impact on `auditd` with additional checking, we ran three filesystem benchmarks, `blogbench` [37], `apache benchmark` [8], and `postmark` [67], on a system with the default `auditd` (`auditd-def`), and duplicates-eliminating `auditd` (`auditd-degap`). Table 4.3 shows the results.

We observe that the performance impact of adding a feature to `auditd` to remove duplicates (except for timestamp) is minimal. In fact, for `apache benchmark`, `auditd-degap` handled about 1.7 times more requests per second than `auditd-def`. `Apache benchmark` probably showed a significant improvement in auditing with duplicate removal because large fraction of access requests in the benchmark were found to be duplicates (over 99%). As a result, the need for `auditd` to log the events, and thus file I/O, is significantly reduced. On the other hand, in case of `blogbench`, duplicate entries were negligible (approximately 2%). From these results, we conclude that for workloads that are repetitive in the files they access, removing duplicates during auditing is potentially advantageous, while not sacrificing performance significantly during log collection for workloads with few duplicates.

4.8 Conclusions

Permission gaps can expose a system to unnecessary risks. This can be worsened by permission creep that can be hard to eliminate in practice [96]. Without information about whether permissions are actually used, identifying and thus removing

permission gaps is a challenge. Towards a common framework for extracting this information, we propose DeGap. We described the framework, and demonstrated the applicability of DeGap for analyzing file and `SSHD` permissions from `auditd` and `SSHD` logs respectively, while highlighting the nuances.

From `SSHD` logs, we found two users being granted access to a server when they should have been disallowed, suggesting permission creep. Additionally, we found that legitimate users only accessed the server using public key authentication. Despite this, the server had allowed password authentication while being subjected to password brute-force attacks. In addition to identifying the permission gaps, DeGap correctly proposed the changes needed for the configuration settings to eliminate these gaps without over-tightening the permissions.

From `auditd` logs, leveraging the user's domain knowledge on potential security threats, DeGap found that Dovecot's private key was world-readable on all the servers we tested, contradicting the recommendation on Dovecot's site [119]. DeGap also found a large number of user files with permission gaps for world-read or world-write. Some of these files appeared to contain sensitive information including passwords. This suggests that the tool could be useful to both administrators and typical users for detecting permission gaps. DeGap also uncovered dormant user groups as candidates for removal from `/etc/group`.

CHAPTER V

Discovering Potential Binary Code Re-Use with Exposé

5.1 Introduction

For a piece of code to be re-used, physical ownership of the code (or a copy of it) is sufficient. However, this does not imply that permission has been explicitly and legitimately granted to a subject to use the code. Unfortunately, sometimes it is infeasible to have an access control mechanism that can determine and enforce granting of access rights (besides the inclusion of a usage license with the software package), thus leading to security implications. Without the ability to mediate accesses, we have to fall back to binary code re-use detection techniques, which we discuss in this Chapter.

5.1.1 Security Implications of Binary Code Re-Use

Developers who do not adhere to the conditions specified in the license agreement for a piece of code should not be allowed to use it. Otherwise, software license violations and intellectual property theft can occur [85]. Organizations go to great lengths to prevent leakage of their software intellectual property. But, significant risks exist due to channels that are difficult to monitor and control. One channel occurs

when a developer within an organization inadvertently uses a library with a license that would later require the organization to openly release the software that makes use of the library. In an anecdotal example, a researcher ran into this issue when using Berkeley DB, thinking that it was under Berkeley license and thus acceptable to use on a project with a company without a fee. It turned out that while the early versions of Berkeley DB were indeed under the Berkeley license, more recent versions were significantly more restrictive, and could have put the company's software at risk with the free version of the license. In another case, researchers found an Internet filtering software using libraries for image recognition from OpenCVS without including a copy of its BSD license text as stipulated by their authors [134].

5.1.2 Other Applications of Detecting Code Re-Use

Besides detecting illegitimate accesses, another application for detecting code re-use is to identify software that are statically linked to an unpatched version of a library. While the library's author may prefer a patched version of the library to be used over an unpatched one, this is not always possible. This can lead to situations where bugs and vulnerabilities in the libraries can be inherited by the applications [92, 58]. Often, the associations between a library and statically-linked applications are not tracked (developers may know it, but users usually do not). Examples of such binaries are (i) customized in-house applications that are not actively maintained, (ii) third-party software, and (iii) pre-installed binaries that come with the operating systems. Over time, bugs and vulnerabilities may be discovered and patched in the libraries. However, they can continue to remain in such applications for a while. Ng et al. termed this phenomenon as latent vulnerabilities [92]. The problem is also studied by Jang et al. in their work on ReDeBug [58], albeit using source code (our focus is on binary code), for uncovering such vulnerabilities.

5.1.3 Possible Approaches

One possible approach to the problem of controlling code re-use may be to leverage mechanisms from digital rights management (DRM). Thus, the compiler will be analogous to the media player while the source code or library will be analogous to the digital media. In other words, a more robust access control mechanism is needed beyond the current practice of using software licenses. However, this approach will not be backward compatible with existing software development paradigms. Besides, scalability may be an issue when determining who may use the code.

Another approach that we will examine in this Chapter is to detect the presence of illegitimate code re-use. The problem of identifying malign code re-use has been studied extensively for the case of source code. It includes works such as ReDeBug [58], DECKARD [62], CCFinder [65], and CP-Miner [73]. However, only few works, such as BinHunt [45], BitShred [60], and SMIT [52], have been proposed for the case of detecting binary code re-use. Reps et al. advocate analyzing executable binary code as it is the authoritative source of information for an application, and note that there are at least three varieties of binary code analysis problems: (i) source code and binary are available, (ii) binary with symbol/debugging information is available, and (iii) binary without symbol/debugging information (stripped binary) [104]. According to Reps et al., the third type of problem is the most challenging, and is the common situation when one uses off-the-shelf applications. We examine the third type of problem for code re-use, and aim to address the following challenges in a scalable manner.

1. **Function inlining** Compilers often perform inlining on small functions to eliminate function call overheads.

2. **Lack of Symbol Information** Symbols, which can be helpful in matching functions, are commonly absent. Our analysis shows that at least 80% of the

applications lack symbols.

3. **Code mutation due to compiler options** Different compiler options can lead to syntactically different but semantically equivalent code being emitted.

Towards identifying binary code re-use, we propose Exposé for identifying potential code re-use between a library and a set of applications. This does not necessarily mean that identified applications derive the re-used code from the library directly. Besides statically linking with the library, a developer could have compiled the application containing source code for a library. Abstractly, in identifying library re-use, we want to identify the set of functions from the library that are used within an application. Unfortunately, this can be a tedious task, even with manual analysis using disassemblers and analysis tools like IDA Pro. If an organization wishes to do this for a large number of applications, the task can be overwhelming.

The main contributions of this Chapter include the following.

- We identify function inlining, lack of symbol information, and code mutation due to varying compiler options as significant challenges for detecting code re-use in binaries.

- To overcome these challenges while ensuring scalability, we propose a multi-phase technique, with the first phase being a fast pre-filtering step to identify a small set of relevant functions in the application that are good candidates for doing a match with the library functions. Subsequent phases do a more thorough matching to ensure a high-quality match. For the first phase (fast), we identified four attributes – number of input parameters, out-degree, function size, and cyclomatic complexity – that determine a set of candidate function pairs (between a library and an application) for a subsequent phase (slower) of semantic matching using a theorem prover. We also use syntactic techniques for matching functions that are not amenable to semantic matching. We propose a method to compute the distance scores used for ranking applications in the order of likelihood that they are using the library.

- We implemented Exposé to evaluate our techniques. Exposé successfully ranked an application known to use a library above a set of 128 other applications known not to use the same library. Using another test library, Exposé ranked 10 out of 2,927 applications amongst the top 11 positions. We verified the correctness with a signature scanner. On manual investigation, the single application, which was not detected by the scanner to use the library, was identified as most likely using some functions from the library.

- Exposé was able to distinguish applications using the different versions of a library, as well as variants compiled using different compiler options.

- Exposé analyzed 97.68% and 99.48% of the applications within five minutes and 10 minutes respectively.

In Section 5.2, we discuss the scope and limitations of the work. In Section 5.3, we detail our approach. In Section 5.4, we provide experimental results and evaluation of our technique. And finally, we conclude in Section 5.5.

5.2 Assumptions and Scope

When we say that an application "uses" a library, we mean that the application is either statically linked to the library, or has been compiled together with the library's source code. Our work focuses on examining these more challenging cases rather than dynamically linked libraries, where the names of the shared libraries typically provide enough information to correctly identify them; moreover, the names of exported functions can be easily extracted and leveraged for identification purpose. Also, updating a dynamically linked library will automatically update the applications that dynamically link to it. However, static linkage of code from libraries continues to be common. An advantage of static linkage is that the library does not have to be distributed with the code and fewer assumptions need to be made about the availability

Table 5.1: Percentage of binaries without symbols in various Linux distributions.

Distributions	Total # of Files	# without Symbols	
		(#)	(%)
Ubuntu 10.04	4268	4254	99.672
Fedora 13	4355	4080	93.685
DVL 1.5	13325	10760	80.750
Debian 5.0.4	4077	4039	99.068
Mandriva Free 2010	5610	5603	99.875

of the library.

We acknowledge the challenge in providing an efficient and effective solution for the code re-use problem. Thus, we focus our efforts in finding a solution that is *practical*. By practical, we mean that the technique should rank candidate applications with true positives in the highest ranks within a reasonable amount of time. The sorted binaries can then be prioritized for further analysis for the presence of specific functions of interests using a combination of manual and more detailed techniques that are typically less scalable.

To further complicate the problem of identifying library re-use, many applications are optimized so that they lack symbol data. Symbol data could have been used to identify names of functions that are used within an application, which could be checked against the names of functions in a library. We ran a script that checks for presence of symbols against the binaries from several Linux distributions. As shown in Table 5.1, at least 80% of binaries on the systems do not have symbols. This is not surprising as symbols consume unnecessary disk space on production systems. We therefore assume the absence of symbols in this work. Of course, symbols can be trivially leveraged to improve our results in practice. Implicitly, tools leveraging symbols, such as `objdump`, will not suffice for most cases.

We assume a benign library usage environment in which functions from the libraries or an entire library is used to build applications. A developer is assumed not

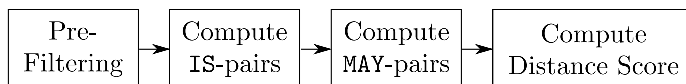


Figure 5.1: Exposé overview.

to be attempting to obfuscate the use of the functions in the library. Existing techniques to de-obfuscate the binary, such as [78] and [79], can be applied if necessary. Moreover, the problem is significant enough even without obfuscated library re-use.

Lastly, we would like the matching to be relatively insensitive to different compiler options. For example, we do not make any assumptions about compiler optimizations that result in basic block re-ordering or function inlining.

5.3 Approach

The inputs to Exposé are two sets of disassembled functions, F and G, that are respectively derived from a library and an application. The goal of the technique is to find the set of functions in F that are likely to be in G. From the results of the match, Exposé attempts to compute a matching score that can be used to rank the results from a large number of applications for a given library. A smaller score corresponds to more similarities.

The library functions in the set F are extracted from a single object file generated by linking all the object files. The functions in the set G are simply extracted by analyzing the application binary. In both cases, extracting the function call graphs is trivial since the linker resolves all the symbols.

Figure 5.1 shows an overview of the different phases for our approach. The pre-filtering phase prepares a set of candidate function pairs for computing semantic equivalence by removing improbable function pairs and functions of no interest to us. Next, we compute the `IS`-pairs to identify equivalent functions. For the remaining functions, we compute the `MAY`-pairs using syntactic techniques. Exposé then com-

putes a distance score that summarizes the pairings by considering the caller/callee relationships.

Algorithm 1 provides an algorithmic overview of the various phases for our technique. In the following sub-sections, we discuss the details of each phase.

5.3.1 Pre-Filtering

Symbolic execution has the advantage of being sound under typical variations in instruction opcodes and optimizations that may be introduced by compilers. But, comparing two functions via symbolic execution can have a scalability problem, usually as a consequence of having many possible paths through the functions. The pre-filtering phase selects candidate function pairs by excluding loader support functions and improbable function pairs to avoid unnecessary symbolic executions, which are computationally intensive.

Loops also require special handling to limit the search space. These problems are widely acknowledged and are currently an area of intense research [30, 50, 135, 136]. To simplify the problem, we only consider the first iteration of each loop.

5.3.1.1 Excluding Loader Support Functions

Given a binary, identifying the functions is basically straightforward, except for a few nuances. During compilation, compilers insert loader support functions to provide support for loading, executing, and terminating the application. However, varying functions are inserted by the compiler depending on the compilation options. For example, stubs such as `__i686.get_pc_thunk.bx` or `__i686.get_pc_thunk.cx` are inserted when the `fPIC` option is specified. To avoid misleading matches with these functions, we exclude them from our pairing algorithm. We list these functions in Table 5.2 for the `GCC` compiler. Table 5.2 shows the common functions that we have observed and may not be exhaustive. Procedure 1 assumes that these functions are

`__do_global_ctors`	`_fini_array`
`__do_global_dtors`	`__fini_array_start`
`__do_global_ctors_aux`	`__fini_array_end`
`__do_global_dtors_aux`	`frame_dummy`
`__i686.get_pc_thunk.bx`	`start`
`__i686.get_pc_thunk.cx`	`_start`
`_init_array`	`__libc_csu_fini`
`__init_array_start`	`__libc_csu_init`
`__init_array_end`	

Table 5.2: List of common functions excluded.

already excluded.

5.3.1.2 Excluding Improbable Function Pairs

Ideally, we would like to test all functions in F with functions in G for semantic equivalence using symbolic execution. But, this would not be a scalable approach. Towards achieving a balance between scalability and correctness, Exposé uses the following criteria based on attributes that are easy to compute for quickly filtering out non-probable function pairs and selecting suitable candidate pairs to test for semantic equivalence.

- Same number of input arguments.
- Same out-degrees.
- Cyclomatic complexity of less than 15.
- Function size of less than 300 bytes.

Two functions are more likely to be equivalent if they have the same number of input arguments and out-degrees. Cyclomatic complexity reflects the number of independent paths [82]. Symbolically executing a function with high cyclomatic complexity quickly degenerates into the path explosion problem. Also, large functions, which usually imply more execution paths, generally lead to disproportionate slowdowns when computing for semantic equivalences. Figure 5.2 and 5.3 show the cumulative

distribution functions for the function size and cyclomatic complexity of 582,959 functions respectively. We chose to perform symbolic execution on function pairs with cyclomatic complexity less than 15 and function sizes less than 300 bytes. 455,078 (78.06%) of the functions satisfy these criteria. In Procedure 1, the sub-procedure `MeetCriteria` is responsible for rejecting improbable function pairs.

5.3.2 Computing semantic matches (`IS`-pairs)

If they are semantically equivalent, Exposé adds the pairing to the `IS`-pairs set. We define functions f_i and g_j to form an `IS`-pair if f_i is semantically equivalent to g_j. For determining semantic equivalence, we use the method proposed by King in which a sequence of instructions are executed symbolically by substituting the input variables with symbolic formulas [68].

To perform symbolic execution, each instruction is translated into a set of constraints on the symbolic formulas. Additionally, constraints on the path conditions and input variables are created. We assert that the input variables for both functions are equivalent. We then extract a set of outputs as a consequence of the function executing along each path. A theorem prover is then used to query for the satisfiability of the output equivalences.

To compare paths in the two functions, Exposé uses the STP constraint solver proposed by Ganesh et al. [44]. We modified the translator from the Binary Analysis Platform by Brumley et al. [29], to process x86 instructions and interface directly with STP. We assume that functions use `GCC`'s default calling convention, `cdecl` [98]. In `cdecl`, the calling function pushes function arguments onto the stack from right to left, and the return value is saved in the `EAX` register. We did not include support for other calling conventions because there is currently no efficient method for distinguishing all calling conventions.

We consider two functions to be a matched `IS`-pair if we can find satisfiable output

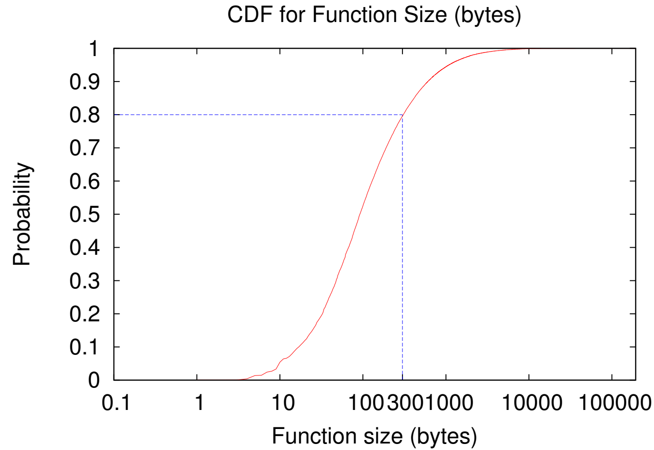


Figure 5.2: Cumulative distribution of function sizes.

equivalences between the paths in the two functions. On the other hand, if the output equivalences cannot be satisfied, we cannot assert that the two functions are not equivalent. This is because of the simplifications in the matching process to achieve scalability. For example, we assumed that functions called within the functions being matched to have a null behavior to reduce matching cost and avoid inter-procedural symbolic execution. As a result, inlining could cause two matching functions to diverge. For example, suppose f_x calls f_y in the library and the corresponding code for f_x in the application is g_x. We may fail to match f_x with g_x if f_y is inlined with f_x to produce g_x during compilation.

We note that when we find two functions f and g to form an `IS`-pair, they are equivalent only under some abstractions. For example, one abstraction we make is that the behaviors of any functions called by f and g are ignored. We also ignore possible differences in the values of CPU registers at the end of the execution since compiler optimizations can introduce differences in the use of registers. Our hope is that these approximations will turn out to be acceptable on real data sets since the end goal is to narrow down the candidate set of applications to a small number that can be manually inspected or analyzed.

It turns out that the lack of a semantic match does not necessarily mean that

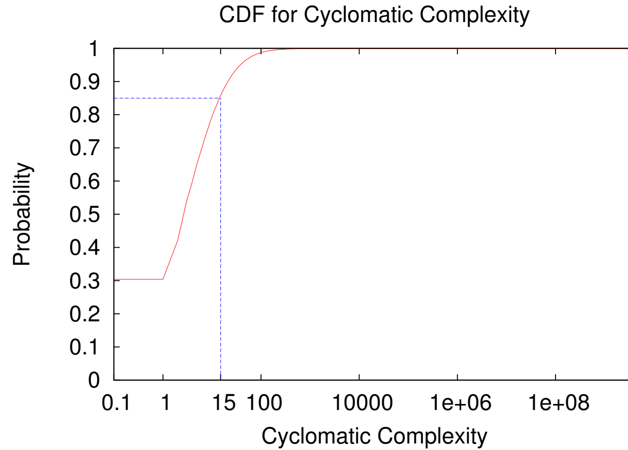


Figure 5.3: Cumulative distribution of function cyclomatic complexities.

the functions are semantically different. If a match is found, then the functions are equivalent. But the absence of a match does not prove that the functions are different. For example, semantic matching is sensitive to order of parameters and return values. If we make a wrong assumption about the calling conventions, semantic matching could fail to return a match.

Semantic matching can also return false positives if some outputs or side-effects of the functions, e.g., register values, are important, but not modeled as inputs or outputs. We found instances of those to be rare on practical data sets. We found one instance in our data sets and it was due to a function stub in both the library and the application.

5.3.3 Syntactic function matching (`MAY`-pairs)

For the set of library functions for which Exposé is unable to find `IS`-pairs, we adopt a heuristic approach for generating matched function pairs, called `MAY`-pairs based on a similarity measure. False positives can be a problem with such approaches. To help reduce false positives significantly, we factored in both function-level similarity and caller-callee relationships. A `MAY`-pair is a pairing between two functions f_i and g_j such that the following two properties are satisfied.

1. f_i makes a function call to itself if, and only if, g_j makes a function call to itself. That is, f_i is recursive if, and only if, g_j is recursive.

2. The biased cosine distance value for the f_i and g_j pair is amongst the r-th smallest biased cosine distance values for pairings between f_i and all functions in G.

Implementation-wise, we chose r to be five. Thus, for each function in the library, we first identified up to five similar functions in an application using a biased cosine distance.

Identifying function recursion is trivial. Therefore we will focus our discussion on the second property. We compute the cosine distance of the n-gram word frequencies between two functions f_i and g_j. The biggest challenge in computing syntactic distances is in handling syntax variations. Ensuring exactness will result in intolerance of any variation in the syntax. On the other extreme, correctness is compromised if all forms of syntax are tolerated. Thus Exposé uses n-gram in a bid to smoothen slight variations in the syntax. An advantage of using the n-gram based approach is its robustness against code relocations made by the compilers, primarily for the purpose of optimization. Code relocations occur during block re-ordering and function inlining.

It is possible that compilers use instructions that are semantically equivalent but syntactically different. For example, "`mov eax, 0`" is semantically equivalent to "`xor eax, eax`", but syntactically different. One possible solution to mitigate such differences is to normalize the instructions such that all instructions that are members of a set of semantically equivalent instructions are replaced with the same instruction. While this is a possible improvement to Exposé, we find that the effects of semantically equivalent but syntactically different instructions are usually smoothened out by syntactically equivalent instructions.

Listing 1 shows the instructions in a basic block of the shared library `libz.so` and Listing 2 shows the instructions for a semantically equivalent basic block. Three of the opcodes differ between the two basic blocks despite having similar mnemonics. Since sequences of instructions having the same mnemonics are likely to be semantically similar, we assign intermediate representation values to similar mnemonics, as shown in the first column of the two listings. Additionally, we do not consider the operands when computing the trigrams since, intuitively, they are highly variable.

A larger value of n would require the binaries to have longer common subsequences of opcodes for a good match. This implies that the number of true positives is also reduced for large n, and the algorithm is less tolerant of opcode variations. On the other hand, if n is too low, the number of false positives will be high as short common subsequences are sufficient for the binaries to match. Empirically, we find three to be a good value for n. Thus, Exposé computes trigrams for the opcodes. Using the intermediate representation, the first trigram word in Listing 1 would therefore be (0x7A, 0x7A, 0xD2). Computation of n-grams can be expensive in both space and time. Nagao et al. proposed an efficient technique that does not require a separate table for storing the n-gram words during computation [90]. We implemented their algorithm for computing the trigrams.

5.3.3.1 Eliminating Function Prologues and Epilogues

Function *prologues* and *epilogues* do not contribute any additional information on whether two functions are equivalent. A function prologue is responsible for setting up the stack frame. A function epilogue destroys the stack frame and restores the original value of the base pointer. Table 5.3 shows the typical function prologue and two equivalent epilogues on 32-bit machines. We exclude prologues and epilogues from the generation of our n-gram words.

Prologue	Epilogues	
`push ebp` `mov ebp,esp`	pop ebp retn	leave retn

Table 5.3: Typical x86 function prologue and two possible epilogues.

5.3.3.2 Computing Cosine Distance from n-grams

Suppose we let N_x be the vector of trigram word frequencies in x. Then we can define the cosine distance between $f_i \in F$ and $g_j \in G$ as in Equation 5.1.

$$c_{i,j} = 1 - \frac{N_i \cdot N_j}{|N_i| \cdot |N_j|} \tag{5.1}$$

Cosine distance computes the cosine of the angle between two vectors of the same dimension. A value of 1 indicates that the two vectors are fully independent, while a value of 0 implies they are exactly the same.

It is possible for two distinct function pairs to have the same cosine distance value. To encourage function pairs that are more likely to be similar and discourage function pairs that are unlikely to be similar, we bias the cosine distance value, $c_{i,j}$, using three rules. Firstly, if f_i and g_j have different out-degrees, increase $c_{i,j}$ by 0.1. Functions with different out-degrees are less likely to be similar. However, it is still possible for such functions to be similar, such as when inlining occurs. Next, if f_i and g_j have the same out-degrees, decrease $c_{i,j}$ by 0.05. Lastly, if f_i and g_j have the same number of non-zero input parameters, decrease $c_{i,j}$ by 0.2. The values used for biasing are derived empirically and we have found them work well in our experiments.

5.3.4 Distance Score

Given the results of matches for each function in the library to a corresponding function in the application, Exposé computes a summary score to help rank the results. A match could be generated by semantic equivalence (which is a Boolean

value) or by the heuristic approach (which is a value between 0 and 1, with lower-values indicating higher similarity). We now explain how the distance score for the two sets of functions F and G is computed. The procedure comprises three main steps. The first step involves computing the local score by observing the neighbors of the two functions f_i and g_j whose pairing can be found in `MAY`-pairs. Based on the local scores, the second step identifies candidate function pairs whose functions have corresponding lowest local scores. The third step attempts to group functions in the library that match functions in the application with the same caller/callee relationships.

5.3.4.1 Computing Local Scores

Given the set of `MAY`-pairs, we want to find the best possible match. Recall that for each function in the library, Exposé has five potential `MAY`-pairs to functions in an application based on best cosine scores. It seems that we could just run a mincost bipartite matching algorithm, but we found that it can give undesirable results, which eventually cause too many false positives. The problem is that it does not take into account caller-callee relationships and if f erroneously matches with g, the error can propagate because it prevents g from being correctly matched.

To get more robust matching, Exposé computes a local score, $l_{i,j}$, for every `MAY`-pair (f_i, g_j) as shown in Algorithm 2, factoring in the callers and callees of f_i and g_i. This is a recursive procedure that will lead to better similarity scores for functions with similar callers or similar callees. To account for function inlining, which may occur when a library is linked into an application, the procedure allows pairing between f_i and a caller function of g_j when computing the minimum cost assignment. For the same reason, we also allow pairings between a callee function of f_i with g_j. Hungarian algorithm (also called Munkres algorithm) [69] is used to find the best localized mapping between the caller/callee functions of f_i and g_j.

5.3.4.2 Inconsistent `MAY`-pairs Elimination

To help find a high-quality bipartite matching from the pairs, Exposé identifies and removes *inconsistent* `MAY`-pairs after computing the local scores for all `MAY`-pairs. We consider the pairing between two functions f_i and g_j to be consistent only if the minimum local score for f_i corresponds to g_j and if the minimum local score for g_j corresponds to f_i. Otherwise, the pairing is inconsistent and Exposé eliminates them from the set of `MAY`-pairs.

After eliminating inconsistent `MAY`-pairs, we normally end up with a bipartite matching (since the minimum scores in rows and columns are usually unique). If not, we can run a standard bipartite matching algorithm to generate the final pairs.

5.3.4.3 Grouping Related Matching Functions

We prefer matches in which both callers and callees are matched to isolated matches. Based on this intuition, given a set M of final matched pairs (which includes both `IS`-pairs and the consistent `MAY`-pairs), Exposé groups the library functions as follows. Given a matched pair (f_1, g_2) in M, if there is another pair (f_2, g_2) in M such that f_1 calls f_2 and g_1 calls g_2, Exposé puts f_1 and f_2 in the same group. This is achieved programmatically by finding undirected cycles of length four where edges are formed by function calls or pairings. We show a trivial example in Figure 5.4. In this example, two groups exist, with the first group consisting of f_1 and f_2 and the second group consisting of f_3.

To give preference to matched groups over isolated matches, Exposé scales the cosine distance of each matched function f_i by dividing by the size of its group (recall that lower scales indicate higher similarity).

To compute an overall composite matching score between a library and an application, Exposé uses a simple method for ranking the matches: the final distance score is obtained by taking the average of the scaled cosine distances of the pairs in

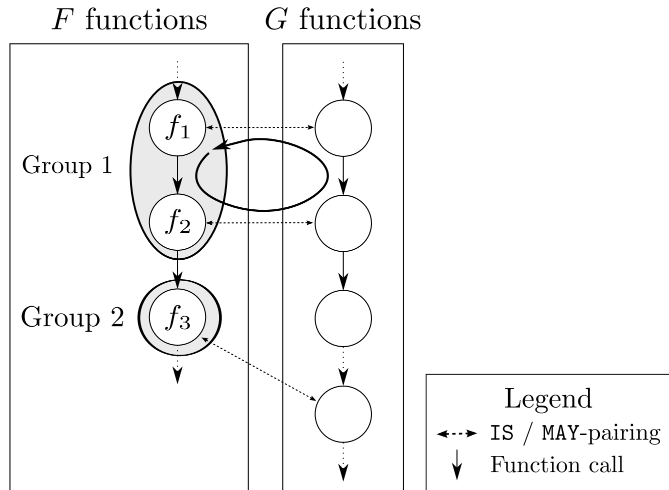


Figure 5.4: Function grouping, given a set of matching pairs.

the final matched set, M. This simple method worked quite well for the purpose of ranking, as discussed in the next section.

5.4 Results and Evaluation

The experimental objectives are to understand how well Exposé performs qualitatively (in terms of ranking matching applications to a given library) and quantitatively (in terms of timing performance). Our experiments were conducted on machines with Intel Xeon processor at 2.50 GHz running Redhat Enterprise Linux release 5.4.

5.4.1 Quality of Ranking of Applications

For this experiment, we wanted to evaluate the quality of Exposé's rankings. One challenge we encountered was the lack of ground truth, since it would not have been possible for us to manually analyze thousands of binaries. To address the challenge, we conducted the experiment in two parts. In the first part (controlled), Exposé generated rankings for a given library against a small set of applications that we knew to be using the library. In the second part (uncontrolled), Exposé computed rankings for another library against a large set of applications for which we had little

Table 5.4: 10 smallest distance scores using `libpng` as test library. Smaller scores indicate better matches between the library and the applications.

File	Score	Elapsed (min)	Size (kb)
feh-static	0.273	0.235	482.10
feh-shared	0.288	0.181	435.50
sash	0.290	1.599	595.04
bash	0.306	3.754	663.69
md5sum	0.312	0.210	18.93
sln	0.314	0.885	449.10
gawk-3.1.5	0.330	1.188	293.05
tar	0.342	0.776	188.20
sed	0.353	0.739	93.18
cpio	0.355	0.821	51.53

information about their library usages. We used different libraries for the two parts to study Exposé's ability to work for different libraries.

5.4.1.1 Controlled

We used `libpng v1.2.43` as the test library. The library is commonly used by applications for processing PNG images. The version of the library we used contained a buffer overflow vulnerability CVE-2010-1205 resulting from insufficient checks on buffer space in the function `png_push_process_row`. For the test applications, we compiled two versions of a simple image viewer application called `feh`. The first version, `feh-static`, was statically linked with the test library, while the second version, `feh-shared`, used dynamic linking and was used as a control. In addition, we included 128 executable binaries from the `/bin` directory of one of the RedHat machines. These applications were unlikely to use `libpng`. We ran Exposé for finding code re-use of the `libpng` test library in the set of test applications. The results for the 10 smallest scores are shown in Table 5.4. From the results, we observe that `feh-static` correctly had the nearest distance score.

Table 5.5: 15 smallest distance scores using `zlib` as test library.

File	Score	Elapsed (min)	Size (kb)
dpkg-deb	0.07	1.352	174.40
rsync	0.09	5.762	262.93
sash	0.10	2.159	595.04
insmod.static.old	0.11	2.827	692.80
modinfo.old	0.14	1.283	107.64
depmod.old	0.14	1.317	131.54
depmod	0.14	1.256	88.28
modinfo	0.14	1.175	64.12
modprobe	0.14	1.258	77.17
rateup	0.14	1.416	337.43
insmod.old	0.14	1.369	174.46
ddd	0.19	6.755	2707.53
psp	0.21	0.044	22.38
snort	0.22	2.164	676.97
jpegtran	0.22	0.379	77.69

5.4.1.2 Uncontrolled

To test Exposé's ability to correctly rank a large set of applications, we used 2,927 unique application binaries (determined by their SHA-1 hash) from DVL v1.5 [128], a Linux distribution based on Slackware containing a large collection of applications for practicing purpose by penetration testers. We used `zlib v1.2.3` as the test library. The choice of the library was intentional because it contained identifiable signatures. Our techniques did not make use of the signature; the signature was only used to estimate the ground truth.

To verify the correctness of the results, we used `Clamscan` to analyze the same set of applications for the known signature. `Clamscan` is an open-source tool for scanning applications using known signatures, and it is more commonly used as part of `ClamAV`, an anti-virus engine [5]. The library contained 88 functions. Using `Clamscan`, we identified 10 application binaries that made use of `zlib`, all using version 1.2.3: `depmod`, `modinfo`, `modprobe`, `rateup`, `sash`, `rsync`, `depmod.old`, `insmod.old`, `insmod.static.old`, and `modinfo.old`.

Table 5.5 ranks 15 application binaries with the nearest distance scores. We

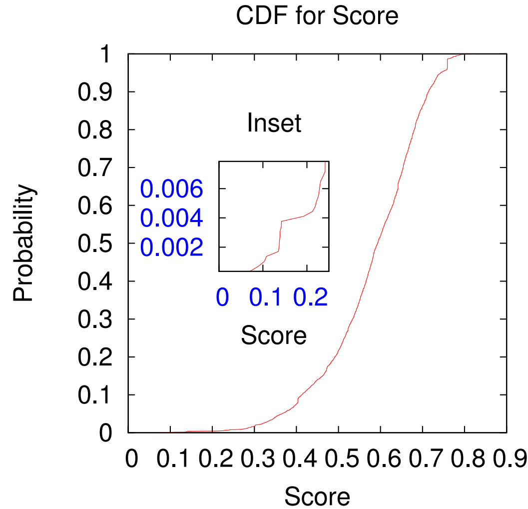


Figure 5.5: Cumulative distribution of distance scores. The inset shows the probabilities for scores between 0 and 0.25.

observe that the 10 application binaries detected by `Clamscan` were amongst the top 11 scores. There was one surprise. `dpkg-deb`, which was not detected by `Clamscan`, was ranked first.

We used `Clamscan` on `dpkg-deb` using the signature for `zlib v1.2.2` and a positive result was returned. Additionally, we examined dumping out the strings in the binary and found that it contained the string "inflate 1.2.2 Copyright 1995-2004 Mark Adler", indicating that it was possibly linked with `zlib v1.2.2` at some point. This was further confirmed by disassembling `dpkg-deb` with IDA Pro [51] and comparing the instructions. The small similarity score (indicating a good match) for version 1.2.3 was achieved because the set of `zlib` functions called by `dpkg-deb` was a subset of the functions that did not vary significantly between versions 1.2.2 and 1.2.3.

Figure 5.5 shows the cumulative distribution of the distance scores for all 2,927 binary applications. The distance scores for the true positives were ranked above that for all other applications except `dpkg-deb`. This indicates that Exposé can be helpful in significantly narrowing down the number of binaries that need to be examined for a closer match.

Table 5.6: Distance scores of applications compiled with different `zlib` versions compared with `zlib v1.2.3`.

App	`zlib` version	Score
app1	1.1.3	0.034
app2	1.1.4	0.034
app3	1.2.1	0.010
app4	1.2.2	0.010
app5	1.2.3	0.006
app6	1.2.4	0.020

5.4.2 Library Versions and Compiler Options

Next, we did experiments to study Exposé's ability to distinguish among library versions and its sensitivity to common compiler optimization options.

5.4.2.1 Distinguishing Library Versions

Our goal here was to determine if Exposé could be effective in identifying specific library version use without resorting to specialized methods such as computing differences among libraries or generating signatures for the differences. As an initial test case, we compiled a test application statically linked with different versions of `zlib` and ran Exposé against `zlib v1.2.3`. The functions called were the same and included modified functions across the different library versions. The results are shown in Table 5.6. We found that the distance score increases as the version is further away from the current version, indicating that Exposé can be effective in distinguishing variations in the binaries.

We next evaluated Exposé with different versions of `zlib` on the 11 applications (including dpkg-deb) that evidently made use of `zlib`. The results are shown in Table 5.7. We observe that the 10 applications that were statically linked with `zlib v1.2.3` have the smallest distance scores with `zlib v1.2.3`. However, `dpkg-deb` also has its smallest distance score with `zlib v1.2.3`. In addition, its distance score with `zlib v1.2.2` is also relatively small. We found four functions in both `v1.2.2`

Table 5.7: Distance scores between the 11 applications with smallest distance scores and different `zlib` versions. The smallest distance scores for each application are **bolded**. The correct version used by the application is marked with an asterisk (*).

Application	v1.1.4	v1.2.2	v1.2.3	v1.2.4
dpkg-deb	0.300	0.073*	**0.070**	0.285
rsync	0.270	0.092	**0.089***	0.303
sash	0.267	0.154	**0.103***	0.295
insmod.static.old	0.443	0.156	**0.108***	0.683
modinfo.old	0.441	0.205	**0.135***	0.426
depmod.old	0.407	0.209	**0.137***	0.431
depmod	0.431	0.214	**0.137***	0.441
modinfo	0.443	0.209	**0.139***	0.441
modprobe	0.436	0.211	**0.139***	0.448
rateup	0.432	0.211	**0.142***	0.454
insmod.old	0.443	0.220	**0.142***	0.431

and `v1.2.3` to have `IS`-pairs: `compressBound`, `deflatePrime`, `deflateBound` and `zcalloc`. These matches contributed the most to the small distance score. All these functions were found to be unchanged between the two versions.

To further analyze the relative contributions of `IS`-pairs and `MAY`-pairs to the final scores, Table 5.8 shows the number of `IS`-pairs, number of `MAY`-pairs and the final score that would have resulted if only `MAY`-pair scores were used for the 11 applications for the various library versions. Our findings indicate that the count of `MAY`-pairs or the score from `MAY`-pairs, while useful as a component for generating a ranking of applications, is less effective in distinguishing among versions, probably because n-gram matching is not likely to produce significant differences for versions of the same library. On the other hand, the count of `IS`-pairs, even if small, is a very good indicator for identifying the likely library version.

We note that when we attempted to use semantic matching based on symbolic execution to the entire set of library functions, the matching task took too long (it did not finish). Our results suggest that restricting the semantic-match approach to a promising set of candidate functions, based on an appropriate criteria (e.g.,

Table 5.8: Number of tuples (number of `IS`-pairs, number of `MAY`-pairs, average `MAY`-pair scores) for the top 11 binaries in the same order as Table 5.5.

File	v1.1.4	v1.2.2	v1.2.3	v1.2.4
dpkg-deb	0, 38, 0.34	4, 39, 0.35	4, 34, 0.33	1, 36, 0.36
rsync	1, 27, 0.30	3, 33, 0.33	3, 32, 0.34	1, 42, 0.35
sash	0, 31, 0.30	2, 39, 0.36	3, 42, 0.36	1, 46, 0.33
insmod.static.old	0, 34, 0.30	2, 41, 0.36	3, 46, 0.37	0, 21, 0.68
modinfo.old	0, 26, 0.48	2, 27, 0.47	3, 33, 0.47	1, 31, 0.47
depmod.old	0, 24, 0.44	2, 29, 0.48	3, 31, 0.47	1, 32, 0.47
depmod	0, 26, 0.46	2, 30, 0.47	3, 31, 0.47	1, 33, 0.47
modinfo	0, 23, 0.46	2, 26, 0.47	3, 29, 0.47	1, 28, 0.47
modprobe	0, 25, 0.45	2, 26, 0.47	3, 30, 0.47	1, 30, 0.48
rateup	0, 30, 0.48	2, 35, 0.47	3, 41, 0.47	1, 45, 0.48
insmod.old	0, 32, 0.47	2, 34, 0.49	3, 37, 0.48	1, 36, 0.47

same number of inputs/out-degrees or low cyclomatic complexity), can still produce good match results, including identifying the right library version, while significantly enhancing scalability.

5.4.2.2 Robustness under Varying Compiler Options

To study the robustness of Exposé under different compiler options, we compiled the test application using different compiler options. We verified that the compiled applications were different using `bsdiff`, a tool for generating patch files between binaries. We obtained the scores 0.031, 0.008, 0.006, and 0.054 for compiler options O1, O2, O3, and Os respectively. The distance score is the smallest when the application was compiled with the same compiler option as `zlib`, i.e., O3. Comparing with the results in Table 5.5, the distances obtained for applications using other compiler options are also relatively small.

5.4.3 Timing Performance

Figure 5.6 shows the cumulative distribution of the elapsed times. The elapsed times do not include the times taken for disassembling the binaries since that will

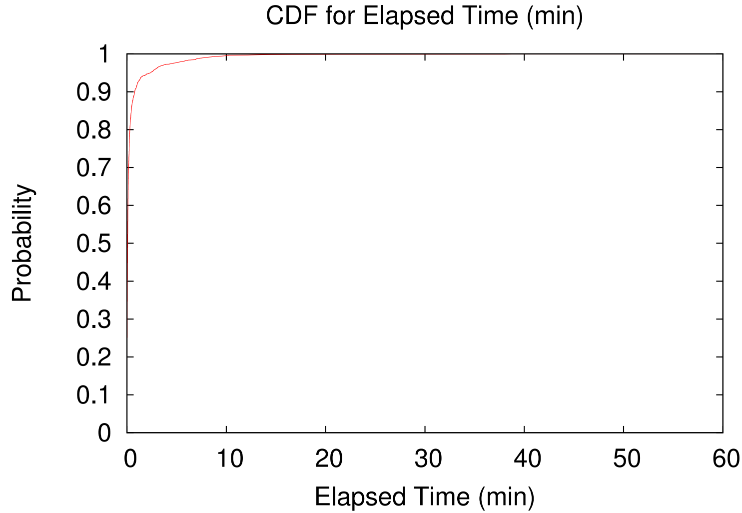


Figure 5.6: Cumulative distribution of elapsed times.

vary depending on the disassembler's performance. We used IDA Pro [51] as our disassembler. Exposé analyzed 97.68% and 99.48% of the binaries within five and 10 minutes respectively, with the longest analysis taking 57.62 minutes. Our results demonstrate the scalability of Exposé clearly.

5.5 Conclusion

Identifying code re-use has many important security applications such as detecting illegitimate software usage and vulnerable or buggy code re-use. Previous efforts towards solving the problem largely focused on detecting code re-use in source code. With Exposé, we aim to provide a practical solution for detecting potential binary code re-use, which is challenged by the lack of symbols, function inlining, and compiler-induced instruction variations.

Exposé determines candidate function pairs from a library and an application for semantic matching based on the number of input parameters, out-degree, function size, and cyclomatic complexity. We also use function level n-gram analysis to determine matching for pairs that are not amenable to symbolic execution. Thus, for each

function in the library, we have either one of the three results: a semantic match, a syntactic match with a distance measure, or no match to a function in an application. When applied to a large number of applications for the given library, these results are summarized into a score for ranking the match quality between the library and the application.

To evaluate the ranking quality, we used Exposé to identify an application statically linked with `libpng` and placed with a set of 128 applications not known to be statically linked with `libpng`. In this controlled experiment designed to overcome the lack of ground truth, Exposé ranked the application statically linked with `libpng` at the top. Using another test library, `zlib`, Exposé ranked 2,927 applications, with the top 10 out of 11 applications also found to use `zlib` by a signature scanner. Upon manual analysis, the top ranked application that was not detected by the signature scanner was found to be linked with an earlier version of the library that contained similar functions as the test library. When we varied the compiler options, and used different versions of a test library, Exposé generated the shortest distances (or very close to that value) between applications and the library variants they were using. Exposé analyzed 97.68% and 99.48% of the binaries within five and 10 minutes respectively.

Algorithm 1 Compute distance score between two sets of functions F and G.

Require: Disassembled functions $F = \{f_1, f_2, \ldots, f_m\}$
Require: Disassembled functions $G = \{g_1, g_2, \ldots, g_n\}$

```
IS-Pairs = ∅
MAY-Pairs = ∅

{STEP 1: Populate IS-pairs with function pairs that are semantically equivalent and MAY-pairs
with pairs that are either not tested for semantic equivalence or test does not return true.}
for i = 1 → m do
   A_i = ExtractAttributes( f_i )
   for j = 1 → n do
      B_i = ExtractAttributes( g_j )
      if MeetCriteria( f_i, A_i, g_j, B_j ) ∧
      IsSemanticallyEquivalent( f_i, g_j ) then
         {STEP 1a: Test for semantic equivalence between f_i and g_j.}
         IS-Pairs = IS-Pairs ∪ (f_i, g_j)
      else
         {STEP 1b: Add (f_i, g_j) to MAY-Pairs. The MAY-Pairs data structure keeps at most 5
         g_j's per f_i with the smallest cosine distance from f_i.}
         MAY-Pairs.add(f_i, g_j)
      end if
   end for
end for

{STEP 2: Compute localdist score for each MAY-pair.}
for all (f_i, g_j, c_{i,j}) ∈ MAY-Pairs do
   localdist[f_i][g_j] = CalcLocalScore( (f_i, g_j, c_{i,j}) )
end for

{STEP 3: Eliminate inconsistent MAY-pairs. }
C = ∅
for i = 1 → m do
   u = RowMin(localdist, i)
   for j = 1 → n do
      v = ColMin(localdist, j)
      if u = v then
         C = C ∪ {(f_i, g_j)}
      end if
   end for
end for

{STEP 4: Form function groups.}
H = FunctionGrouping( C )

{STEP 5: Find average distance of scaled localdist scores and divide by number of IS-pairs.}
s = 0
for all (f_i, g_j) ∈ C do
   s = s + localdist[f_i][g_j]/SizeOfGroup(H, f_i)
end for
s = s/SizeOf(C)
s = s/SizeOf(IS-Pairs)
return s
```

Listing 1 Basic block in `libz.so`.

```
IR   Opcode                  Mnemonic
7A   8B 6B 70                mov     ebp, [rbx+70h]
7A   48 8B BB 80 00 00 00    mov     rdi, [rbx+80h]
D2   85 ED                   test    ebp, ebp
EF   41 0F 49 EC             cmovns  ebp, r12d
D2   48 85 FF                test    rdi, rdi
55   74 05                   jz      short loc_2CC2
10   E8 96 F0 FF FF          call    _free
```

Listing 2 Semantically equivalent basic block in `modprobe`.

```
IR   Opcode                  Mnemonic
7A   44 8B 63 70             mov     r12d, [rbx+70h]
7A   48 8B BB 80 00 00 00    mov     rdi, [rbx+80h]
D2   45 85 E4                test    r12d, r12d
EF   44 0F 49 E5             cmovns  r12d, ebp
D2   48 85 FF                test    rdi, rdi
55   74 05                   jz      short loc_4054F1
10   E8 E7 C0 FF FF          call    _free
```

Algorithm 2 Compute local score between two functions f_i and g_j.

```
function CalcLocalScore (f_i, g_j, d)
if d >DEPTHLIMIT then
  return c_{i,j}
end if
c_p = CalcParentLocalScore(f_i, g_j, d)
c_c = CalcChildLocalScore(f_i, g_j, d)
l_{i,j} = (c_{i,j} + c_p + c_c)/3
return l_{i,j}
end function

function CalcParentLocalScore (f_i, g_j, d)
for all parent f_p of f_i do
  for all parent g_p of g_j do
    m[f_p][g_p] = CalcLocalScore( f_p, g_p, d + 1 )
  end for
  m[f_i][g_p] = CalcLocalScore (f_i, g_p, d + 1)
end for
l_{i,j} = CalcMunkres( m[f_i][g_p] )
return l_{i,j}
end function

function CalcChildLocalScore (f_i, g_j, d)
for all child g_c of g_j do
  for all child f_c of f_i do
    m[f_c][g_c] = CalcLocalScore( f_c, g_c, d + 1 )
  end for
  m[f_c][g_j] = CalcLocalScore (f_c, g_j, d + 1)
end for
l_{i,j} = CalcMunkres( m[f_i][g_p] )
return l_{i,j}
end function
```

CHAPTER VI

Conclusions and Future Work

6.1 Contributions

This thesis discusses the challenges of adhering to the least privilege principle, specifically in three common scenarios where traditional access control mechanisms are ineffective and accesses can be abused. For three scenarios where accesses are commonly abused, it discusses methods to detect such accesses, and where practically feasible, techniques to propose changes to access control policies to remove them.

- **SEAL** Uses semi-private email aliases to limit impact of email address leakages.
- **DeGap** Detects permission gaps using logs to estimate needed permissions, and proposes solutions to mitigate the gaps.
- **Exposé** Detects binary code re-use towards preventing illegitimate uses.

In the first scenario, this thesis focuses on the problem of email leakages, where the email address (object) is accessed by rogue senders (subjects). As characterized by the Spamhaus Project, spam is really an issue of consent rather than content [122]. In the current paradigm, email address owners do not have an effective means of protecting themselves against email address leakages. In addition, some businesses require users to disclose their official email addresses to validate their affiliations with certain organizations. This can over-disclose user information to those businesses. This thesis presents the concept of semi-private email aliases and discusses its imple-

mentation in SEAL, a system that allows users more control over their email aliases. SEAL also allows web services to validate a user's affiliation with an organization without having access to the user's official email address, which might have to be used for strictly official purposes according to company policies. SEAL distinguishes itself from other DEA systems by including a more advanced mechanism for managing finer-grained alias lifecycles, thus allowing more flexibility when retiring compromised email addresses. SEAL also integrates well with current email systems while at the same time not being overly restrictive. Our findings show that SEAL is compatible with existing email systems and can be effective in controlling unsolicited emails. For validating affiliations, SEAL proved useful when used in a real life scenario where an instructor of a freshman course required students to sign-up for an online forum. The objective was to prevent the students' university email IDs from being disclosed to the service. Over 80% of the students chose to use SEAL rather than their university email IDs.

In the second scenario, this thesis is concerned with systems whose permissions are guarded by access control mechanisms, and system objects are accessed by other processes or users (subjects). Specifically, this thesis examines permission gaps, which expose a system to needless risks. Existing work on access control mechanisms focuses on methods for describing access control policies. However, the lack of information about which permissions are actually used can make the task of identifying and removing permission gaps hard. This thesis describes DeGap, a framework that uses a common logic for computing permission gaps across different services. In our experiments for analyzing `SSHD` logs, DeGap found users and authentication methods that should be removed from `SSHD` configuration files. From `auditd` logs, DeGap discovered numerous files that were granted unneeded permissions, including Dovecot's private key and password files that were world-readable. DeGap also uncovered potentially dormant user groups that could be removed from `/etc/group`.

In the third scenario, towards mitigating illegitimate code re-use, this thesis elaborates on detecting accesses of binary code. To detect code re-use, which has many important security applications including intellectual property theft and vulnerable code inheritance, this thesis presents Exposé. Previous efforts towards detecting code re-use focuses on source code. Exposé aims to provide a practical solution for binaries. The task is challenged by the lack of symbols, function inlining, and compiler-induced instruction variations. Exposé combines syntactic techniques based on n-gram with semantic analysis. When applied to a large number of applications for a given library, these results are summarized into a score for ranking the match quality between the library and the application. In our experiments, to overcome the lack of ground truth when evaluating the ranking quality, we used Exposé to identify an application statically linked with `libpng` and placed with a set of 128 applications not known to be statically linked with `libpng`. Exposé ranked the application statically linked with `libpng` at the top. Using another test library, `zlib`, Exposé ranked 2,927 applications, with the top 10 out of 11 applications also found to use `zlib` by a signature scanner. Manual analysis showed that the top ranked application that was not detected by the signature scanner was linked with an earlier version of the library that contained similar functions as the test library. When we varied the compiler options, and used different test library versions, Exposé generated the shortest distances (or very close to that value) between applications and the library variants they were using. Exposé analyzed 97.68% and 99.48% of the binaries within five and 10 minutes respectively, demonstrating its practicality.

Having a single mechanism to solve the problem of detecting and mitigating unintended accesses for all three scenarios is a major challenge due to each of them having different characteristics. However, lessons from studying these characteristics can help us approach similar problems more systematically as shown in Figure 6.1. Before we elaborate on these characteristics, it is important to note that one

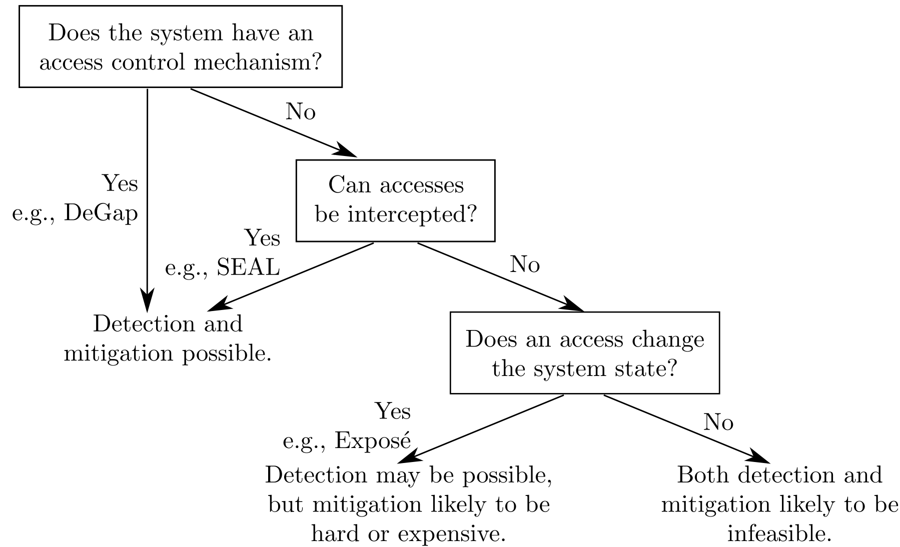


Figure 6.1: Summary of approaches towards detecting and mitigating unintended accesses.

must be able to define a permission, i.e., the subjects, access rights, and objects as a pre-requisite. For a given system, the ability to detect the subjects, access rights, and objects in the system determines the granularity for which unintended accesses can be detected and mitigated. For example, suppose there is an office (object) that people (subjects) can enter (access right). If one is able to identify the individual subjects accessing the office, then one may mitigate unintended accesses to the office at the granularity of individuals. Otherwise, if one can only detect people entering the office but not distinguish between them, one can only prevent all or none of the people from entering the room.

If a system has an access control mechanism, such as in DeGap, the problem of detecting and mitigating unintended accesses can be focused on reducing the permission gaps, since the ability to detect accesses is implied, while mitigation will involve removing extraneous permissions. In other words, the objective of the problem is to make the set of granted permissions be equivalent to the set of needed permissions.

In the absence of an access control mechanism, the ability to intercept and evaluate accesses is necessary for solving the problem. Interception would imply the ability to

detect accesses. It also allows one to develop a mechanism for denying unintended accesses, such as in the case of SEAL.

If an access control mechanism is absent and accesses cannot be intercepted, then one needs to depend on the observable states of a system with and without an access for detecting it. In other words, if a change in the system's state can be observed when an access occurs, then it may be possible to provide a mechanism for detecting the access, such as for Exposé. However, mitigation may be hard or expensive. One possible approach is to try to prevent some types of system changes. For example, for the case of code re-use, it is theoretically possibly for a compiler to verify that a library is being used legitimately by contacting a server. However, clearly there are practicality issues.

If the system state does not change when an access occurs, then detecting and mitigating unintended accesses is likely to be infeasible. In that case, perhaps penetration testing and auditing techniques can be applied to analyze a system.

6.2 Future Work

Looking forward, DeGap currently automatically proposes the set of permissions that excludes accesses not found in the logs. However, users may want more flexibility in specifying specific accesses to exclude. Additional research is needed to achieve this flexibility.

In addition, the number of objects with permission gaps may become prohibitively large, such as when there is a large number of files. Not all permission gaps may have the same level of threats. Thus, it may be possible to generate scores to rank the objects that reflect the different threat levels. Machine learning techniques could be used to extract object attributes, such as file types and access patterns, that can be used towards computing a score for ordering objects based on likelihood of access abuses.

When tightening permission gaps, there may be dependencies between services such that removing access for a particular subject to a certain service is sufficient to prevent it from accessing other services. This may be useful in scenarios where modifying the access control policies of certain services is not desirable. In a trivial example, to prevent a certain user from logging in to a system, changing the firewall rules may not be acceptable while denying `ssh` access may be sufficient. Techniques to identify such opportunities for tightening permission gaps may thus be useful.

In this thesis, DeGap generates solutions to tighten permission gaps based on models generated manually by users. However, it may be beneficial to automate the task. Research into this may involve taint tracking as well as system replay methods.

Challenges also remain for detecting malign binary code re-use. For example, with the growing popularity of mobile computing devices, such as tablets, detecting malign binary code re-use across different platforms may uncover common security issues.

Bibliography

[1] The Bastille Hardening program: increase security for your OS. http://bastille-linux.sourceforge.net/.

[2] Tiger analytical research assistant. http://www-arc.com/tara/, 2002.

[3] Security Threats: A Guide for Small and Medium Businesses. Whitepaper, GFI, 2009.

[4] Anonymous Email: Free disposable email service for receiving emails anonymously. Online, 2011.

[5] ClamAV, 2011.

[6] Mailinator. Online, 2011.

[7] Your Own Protection Mail. Online, 2011.

[8] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html, 2012.

[9] Conflicts Between Privileges and Permissions. http://technet.microsoft.com/en-us/library/cc962012.aspx, 2012.

[10] Django: The Web framework for perfectioniss with deadlines. http://www.djangoproject.com, 2012.

[11] Files and file system security. http://tldp.org/HOWTO/Security-HOWTO/file-security.html, June 2012.

[12] Open Source Host-based Intrusion Detection System. http://www.ossec.net/, 2012.

[13] Open Source Tripwire. http://sourceforge.net/projects/tripwire/, 2012.

[14] Martín Abadi, Andrew Birrell, Mike Burrows, Frank Dabek, and Ted Wobber. Bankable Postage for Network Services. In *In Proc. Asian Computing Science Conference*, pages 72–90, 2003.

[15] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, pages 41–42, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Alexa. Alexa The Web Information Company. Online, 2011.

[17] Khalid Alzarouni, David Clark, and Laurence Tratt. Semantic malware detection. Technical Report TR-10-03, Department of Computer Science, King's College London, February 2010.

[18] Ion Androutsopoulos, John Koutsias, Konstantinos Chandrinos, Georgios Paliouras, and Constantine D. Spyropoulos. An evaluation of Naive Bayesian anti-spam filtering. *CoRR*, cs.CL/0006013, 2000.

[19] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinos, and Constantine D. Spyropoulos. An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and*

development in information retrieval, SIGIR '00, pages 160–167, New York, NY, USA, 2000. ACM.

[20] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[21] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. *CoRR*, abs/1206.5829, 2012.

[22] Lujo Bauer, Jarred Ligatti, and David Walker. More Enforceable Security Policies, 2002.

[23] Mick Bauer. Paranoid penguin: an introduction to novell apparmor. *Linux J.*, 2006(148):13–, August 2006.

[24] Andreas Beckmann. Debian 'extplorer' Package Insecure File Permissions Vulnerability. `http://www.securityfocus.com/bid/54801/info`, August 2012.

[25] Andreas Beckmann. Debian 'logol' Package Insecure File Permissions Vulnerability. `http://www.securityfocus.com/bid/54802`, August 2012.

[26] Andreas Beckmann. Debian 'openvswitch-pki' Package Multiple Insecure File Permissions Vulnerabilities. `http://www.securityfocus.com/bid/54789`, August 2012.

[27] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, August 2001.

[28] P. Oscar Boykin and Vwani Roychowdhury. Personal Email Networks: An Effective Anti-Spam Tool. *IEEE COMPUTER*, 38:61, 2004.

[29] David Brumley and Ivan Jager. *The BAP Handbook*, May 2009.

[30] Stefan Bucur. Scalable Automated Testing Using Symbolic Execution, 2009.

[31] K. Buyens, B. De Win, and W. Joosen. Resolving least privilege violations in software architectures. In *Software Engineering for Secure Systems, 2009. SESS '09. ICSE Workshop on*, pages 9 –16, may 2009.

[32] Steven Chamberlain. syslog-ng wrong file permission vulnerability. `http://marc.info/?l=bugtraq&m=129597471724746&w=2`, January 2011.

[33] Seokwoo Choi, Heewan Park, Hyun-il Lim, and Taisook Han. A static birthmark of binary executables based on api call structure. In Iliano Cervesato, editor, *Advances in Computer Science ASIAN 2007. Computer and Network Security*, volume 4846 of *Lecture Notes in Computer Science*, pages 2–16. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-76929-3_2.

[34] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[35] F. J. Corbato. A paging experiment with the multics system.

[36] Eugene H. Spafford Dan Farmer. The cops security checker system. In *USENIX Summer Conference*, pages 165–170, June 1990.

[37] Frank Denis. Blogbench: A Filesystem Benchmark for Unix. `http://www.pureftpd.org/project/blogbench`, 2012.

[38] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. *Adaptive Query Processing*, volume 1. 2007.

[39] Dovecot. Dovecot. http://www.dovecot.org/, 2011.

[40] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

[41] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[42] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[43] Steven Furnell and Kerry-Lynn Thomson. Recognising and addressing "security fatigue". *Computer Fraud & Security*, 2009(11):7 – 11, 2009.

[44] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[45] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *ICICS '08: Proceedings of the 10th International Conference on Information and Communications Security*, pages 238–255, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] Rainer Gerhards. Performance Optimizing Syslog Server. http://www.monitorware.com/common/en/articles/performance-optimizing-syslog-server.php, January 2004.

[47] Jennifer Golbeck and James Hendler. Reputation Network Analysis for Email Filtering. In *In Proc. of the Conference on Email and Anti-Spam (CEAS), Mountain View*, 2004.

[48] Joshua Goodman and Robert Rounthwaite. Stopping Outgoing Spam, 2004.

[49] Swati Gupta, Kristen LeFevre LeFevre, and Atul Prakash. SPAN: A unified framework and toolkit for querying heterogeneous access policies. In *Proceedings of Fourth USENIX Workshop on Hot Topics in Security*, pages 29–36, August 2009.

[50] Trevor Hansen, Peter Schachte, and Harald Sndergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In Saddek Bensalem and Doron Peled, editors, *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 76–92. Springer Berlin / Heidelberg, 2009.

[51] Hex-Rays. The IDA Pro Disassembler and Debugger. Online.

[52] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 611–620. ACM, 2009.

[53] IETF. RFC 5322: Internet Message Format. Online, October 2008.

[54] Jaime Devesa Igor Santos, Yoseba K. Penya and Pablo G. Bringas. N-Grams-based FIle Signatures for Malware Detection, 2009.

[55] Hyun il Lim, Heewan Park, Seokwoo Choi, and Taisook Han. A method for detecting the theft of Java programs through analysis of the control flow information. *Inf. Softw. Technol.*, 51:1338–1350, September 2009.

[56] John Ioannidis. Fighting spam by encapsulating policy in email addresses.

[57] Jay Harrell James Cannady. A comparative analysis of current intrusion detection technologies. Technical report, Georgia Tech Research Institute, Atlanta, GA 30332-0800.

[58] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

[59] Jiyong Jang and David Brumley. BitShred: Fast, Scalable Code Reuse Detection in Binary Code. Technical report, Carnegie Mellon University, 2009.

[60] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[61] Min-A Jeong, Jung-Ja Kim, and Yonggwan Won. A flexible database security system using multiple access control policies. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 236 – 240, aug. 2003.

[62] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[63] Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou. Analysis of android applications' permissions. In *SERE (Companion)*, pages 45–46. IEEE, 2012.

[64] Brent Jones. Personalized spam rising sharply, study finds. `http://abcnews.`

go.com/Technology/story?id=6502392&page=1\#.UWW01WJst7k, December 2008.

[65] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28:654–670, 2002.

[66] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 3–14, New York, NY, USA, 2008. ACM.

[67] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997.

[68] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[69] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[70] Butler W. Lampson. Protection. In *Princeton University*, pages 437–443, 1971.

[71] Bo Lang, You Lu, Xin Zhang, and Weiqin Li. A flexible access control mechanism supporting large scale distributed collaboration. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 1, pages 500 – 504 Vol.1, may 2004.

[72] Timothy E. Levin, Cynthia E. Irvine, and Thuy D. Nguyen. A least privilege model for static separation kernels. 2012.

[73] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[74] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[75] Hyun-il Lim, Heewan Park, Seokwoo Choi, and Taisook Han. Detecting theft of java applications via a static birthmark based on weighted stack patterns. *IEICE - Trans. Inf. Syst.*, E91-D:2323–2332, September 2008.

[76] Listserv. Lists with 10,000 subscribers or more. Online, July 2011.

[77] Pingchuan Ma. Log Analysis-Based Intrusion Detection via Unsupervised Learning. Master's thesis, School of Informatics, The University of Edinburgh, 2003.

[78] M. Madou, B. Anckaert, and K. De Bosschere. Code (De) Obfuscation. 2005.

[79] Matias Madou, Ludo Van Put, and Koen De Bosschere. Loco: an interactive code (de)obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, pages 140–144, New York, NY, USA, 2006. ACM.

[80] P.K. Manadhata and J.M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, may-june 2011.

[81] David Mazières and M. Frans Kaashoek. The design, implementation and operation of an email pseudonym server. In *Proceedings of the 5th ACM conference on Computer and communications security*, CCS '98, pages 27–36, New York, NY, USA, 1998. ACM.

[82] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[83] Gary McGraw and John Viega. Software security principles, Part 3: Controlling access - Least privilege and compartmentalization. `http://www.ibm.com/developerworks/library/se-priv/index.html`, November 2000.

[84] Tony A. Meyer and Brendon Whateley. Spambayes: Effective open-source, bayesian based, email classification system. In *CEAS*, 2004.

[85] Akito Monden, Satoshi Okahara, Yuki Manabe, and Kenichi Matsumoto. Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations. *IEEE Software*, pages 42–47, 2011.

[86] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. *Computer Security Applications Conference, Annual*, 0:421–430, 2007.

[87] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 1:1–1:13, New York, NY, USA, 2010. ACM.

[88] Andre Muscat. A log analysis based intrusion detection system for the creation of a specification based intrusion prevention system.

[89] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 314–318, New York, NY, USA, 2005. ACM.

[90] Makoto Nagao and Shinsuke Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *In COLING-94*, pages 611–615, 1994.

[91] Beng Heng Ng, Alexander Crowell, and Atul Prakash. Adaptive Semi-Private Email Aliases. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, AsiaCCS '12, 2012.

[92] Beng Heng Ng, Xin Hu, and Atul Prakash. A Study on Latent Vulnerabilities. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 333–337. IEEE, 2010.

[93] Beng Heng Ng and Atul Prakash. Exposé: Discovering Potential Binary Code Re-Use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, 2013.

[94] Cormac O'Brien and Carl Vogel. Spam filters: bayes vs. chi-squared; letters vs. words. In *Proceedings of the 1st international symposium on Information and communication technologies*, ISICT '03, pages 291–296. Trinity College Dublin, 2003.

[95] Sejong Oh and Soeg Park. An Integration Model of Role-Based Access Control and Activity-Based Access Control Using Task. In Bhavani Thuraisingham, Reind Riet, KlausR. Dittrich, and Zahir Tari, editors, *Data and Application Security*, volume 73 of *IFIP International Federation for Information Processing*, pages 355–360. Springer US, 2002.

[96] Tom Olzak. Permissions Creep: The Bane of Tight Access Management. `http://olzak.wordpress.com/2009/10/01/permissions-creep/`, October 2009.

[97] Jaehong Park and Ravi Sandhu. The UCONABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[98] George M. Penn. Calling Conventions Part I (Passing Integral Arguments). Online, 2007.

[99] C.E. Phillips. *Security assurance for a resource-based RBAC/DAC/MAC security model*. PhD thesis, University of Connecticut, 2004.

[100] Postfix. Postfix. `http://www.postfix.org/`, 2011.

[101] Vipul Ved Prakash and Adam O'Donnell. Fighting Spam with Reputation Systems. *Queue*, 3:36–41, November 2005.

[102] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya K. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

[103] David Recordon and Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[104] Thomas W. Reps, Junghee Lim, Aditya V. Thakur, Gogul Balakrishnan, and Akash Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *CAV*, pages 41–56, 2010.

[105] Gary Robinson. A statistical approach to the spam problem. *Linux J.*, 2003:3–, March 2003.

[106] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[107] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. 1975.

[108] Pierangela Samarati and Sabrina de Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin / Heidelberg, 2001.

[109] R.S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40 –48, sept. 1994.

[110] Riccardo Scandariato, Koen Buyens, and Wouter Joosen. Automated Detection of Least Privilege Violations in Software Architectures. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 150–165, Berlin, Heidelberg, 2010. Springer-Verlag.

[111] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, pages 76–85. ACM, 2003.

[112] F.B. Schneider. Least privilege and more [computer security]. *Security Privacy, IEEE*, 1(5):55 – 59, sept.-oct. 2003.

[113] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[114] Karl-Michael Schneider. A comparison of event models for Naive Bayes anti-spam e-mail filtering. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*, EACL '03, pages 307–314, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[115] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 274–283, New York, NY, USA, 2007. ACM.

[116] E. Eugene Schultz, Robert W. Proctor, Mei-Ching Lien, and Gavriel Salvendy. Usability and security an appraisal of usability issues in information security methods. 1 2001.

[117] Jean-Marc Seigneur and Christian Damsgaard Jensen. Privacy recovery with disposable email addresses. *IEEE Security and Privacy*, 1:35–39, November 2003.

[118] Sushant Sinha, Michael Bailey, and Farnam Jahanian. Shades of Grey: On the Effectiveness of Reputation-based "blacklists". In *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE '08)*, pages 57–64, Fairfax, Virginia, USA, October 2008.

[119] Timo Sirainen. Dovecot SSL configuration. `http://wiki2.dovecot.org/SSL/DovecotConfiguration`, April 2012.

[120] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. NAI Labs Report #01-043, NAI Labs, Dec 2001. Revised May 2002.

[121] Michael G. Solomon and Mike Chapple. *Information Security Illuminated*. Jones and Bartlett, 2005.

[122] Spamhaus. The Definition of Spam. Online, 2011.

[123] Haruaki Tamada, Masahide Nakamura, and Akito Monden. Design and evalu-

ation of birthmarks for detecting theft of java programs. In *In Proc. IASTED International Conference on Software Engineering*, pages 569–575, 2004.

[124] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-Ichi Matsumoto. Java birthmarks - detecting the software theft. *IEICE - Trans. Inf. Syst.*, E88-D:2148–2158, September 2005.

[125] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[126] TMDA. Tagged Message Delivery Agent (TMDA). Online.

[127] Scott Treadwell and Mian Zhou. A heuristic approach for detection of obfuscated malware. In *Proceedings of the 2009 IEEE international conference on Intelligence and security informatics*, ISI'09, pages 291–299, Piscataway, NJ, USA, 2009. IEEE Press.

[128] Cognitive Core UG. Damn vulnerable linux.

[129] Steven J. Vaughan-Nichols. The Rise of Personalized Spam. `http://www.itworld.com/security/103709/the-rise-personalized-spam`, April 2010.

[130] Kristina L. Verco and Michael J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proc. of 1st Australian Conference on Computer Science Education*, pages 86–95. ACM, 1996.

[131] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.

[132] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 280–290, New York, NY, USA, 2009. ACM.

[133] Christopher Wee. Audit logs: to keep or not to keep? In *Recent Advances in Intrusion Detection*, 1999.

[134] Scott Wolchok, Randy Yao, and J. Alex Halderman. Analysis of the Green Dam Censorware System. Technical Report CSE-TR-551-09, University of Michigan, 2009.

[135] Xu Xiao, Xiao song Zhang, and Xiong da Li. New approach to path explosion problem of symbolic execution. In *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, pages 301 –304, sept. 2010.

[136] Ru-Gang Xu. *Symbolic Execution Algorithms for Test Generation*. PhD thesis, University of California, Los Angeles, 2009.

[137] Aram Yegenian and Tassos Dimitriou. Inexpensive Email Addresses: An Email Spam-Combating System. In Sushil Jajodia, Jianying Zhou, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Security and Privacy in Communication Networks*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 35–52. Springer Berlin Heidelberg, 2010.

[138] Jonathan A. Zdziarski. *Ending Spam: Bayesian Content Filtering and the Art*

of Statistical Language Classification. No Starch Press, San Francisco, CA, USA, 2005.

[139] Le Zhang, Jingbo Zhu, and Tianshun Yao. An evaluation of statistical spam filtering techniques. *ACM Trans. Asian Lang. Inf. Process.*, 3(4):243–269, December 2004.