CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Analysis and detection of security vulnerabilities in contemporary software
Authors	Pieczul, Olgierd
Publication date	2017
Original Citation	Pieczul, P. 2017. Analysis and detection of security vulnerabilities in contemporary software. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2017, Olgierd Pieczul. - http://creativecommons.org/licenses/by-nc-nd/3.0/
Download date	2026-11-07 01:42:55
Item downloaded from	https://hdl.handle.net/10468/3975



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Analysis and Detection of Security Vulnerabilities in Contemporary Software

Olgierd Pieczul
M.Sc.



NATIONAL UNIVERSITY OF IRELAND, CORK

Department of Computer Science

Thesis submitted for the degree of Doctor of Philosophy

January 2017

Head of Department: Prof. Cormac Sreenan

Supervisor: Dr Simon Foley

Research supported by IBM

Contents

List of Figures v
List of Tables vii
Abstract ix
Acknowledgements x

1 Introduction **1**
1.1 Contemporary Software Development 1
1.2 Detecting Vulnerabilities as Anomalies 3
1.3 Thesis Layout and Previous Work 6

2 The Dark Side of the Code **8**
2.1 Introduction 8
2.2 A Microblog Application 9
2.3 The Security Gap 14
2.3.1 Accessing arbitrary URLs 14
2.3.1.1 Local network access 15
2.3.1.2 Redirect to a local site 16
2.3.1.3 Local file access 17
2.3.2 Modifying internal state of application 18
2.4 Unexpected Behavior Traces 20
2.5 Conclusion 21

3 Anomaly Detection **23**
3.1 Introduction 23
3.2 Overview of Intrusion Detection Techniques 24
3.3 From Knowledge to Behavior-Based Detection 26
3.3.1 Knowledge based systems 27
3.3.2 Statistical models 28
3.3.3 Expert systems 29
3.3.4 Operation sequence models 30
3.3.5 Process mining 31
3.4 Exploring Sequence-Based Anomaly Detection in Software . . . 31
3.4.1 Data models 32
3.4.1.1 Look-ahead pairs 32
3.4.1.2 Subsequences 34

3.4.1.3 Groups of subsequences 34
3.4.1.4 Frequency-based models 35
3.4.1.5 Richer models 35
3.4.2 Attacks 35
3.5 Conclusion 37

4 Practical Challenges of Anomaly Detection in Contemporary Software 38
4.1 Introduction 38
4.2 Abstraction and Scope 39
4.3 Generating Baseline Activity 41
4.4 Behavioral Reference Model 43
4.4.1 Applicability to modern software platforms 43
4.4.2 Model expressiveness 44
4.5 Conclusion 46

5 Behavioral Norms 47
5.1 Introduction 47
5.2 Events and Traces 47
5.3 Scope and Filtering 48
5.4 Strands and Partitions 50
5.5 Norms 52
5.6 Conclusion 53

6 Exploring Behavioral Norms 54
6.1 Introduction 54
6.2 Norms in HTTP Logs 54
6.2.1 Strands for HTTP events 56
6.2.2 Norms for HTTP traces 57
6.3 N-gram Based Trace Equivalence 59
6.4 Norm Search 61
6.5 Attribute Search 62
6.5.1 N-gram based norm similarity 62
6.5.2 Implementation of attribute search 64
6.6 Attribute Search Evaluation 65
6.6.1 Norms in a simulated system 65
6.6.1.1 Norms similarity and aggregation 68

6.6.1.2 Simulating anomalies . . . 70
6.6.2 Norms in an enterprise system . . . 72
6.6.3 Discussion . . . 75
6.7 Modeling Behavior of Collaborating Systems . . . 75
6.7.1 An online photograph sharing service . . . 76
6.7.2 Norms in online photograph sharing service . . . 77
6.7.3 Provider anomalies . . . 78
6.7.4 Anomalies across multiple collaborating providers . . . 79
6.7.5 Discussion . . . 80
6.8 Conclusion . . . 81

7 Runtime Verification of Java Applications 83
7.1 Introduction . . . 83
7.2 Scope . . . 83
7.3 Modeling Trace Equivalence . . . 85
7.3.1 Approximating norms with n-grams . . . 86
7.3.2 Groups and arrangements . . . 87
7.4 Runtime Verification . . . 88
7.4.1 Verification algorithm . . . 90
7.4.2 Algorithm examples and discussion . . . 91
7.5 Anomaly Manager . . . 95
7.6 Discussion . . . 96
7.7 Conclusion . . . 97

8 Experimental Evaluation 98
8.1 Introduction . . . 98
8.2 Experiment Setup . . . 99
8.3 Building Behavioral Profiles . . . 100
8.4 Vulnerability Tests . . . 101
8.4.1 Vulnerability test results . . . 103
8.4.2 False negatives . . . 105
8.4.3 False positives . . . 106
8.4.4 Results interpretation . . . 107
8.5 False Positive Tests . . . 108
8.6 Discussion . . . 109
8.6.1 Anomaly prevention in practice . . . 109

8.6.2 Additional insights 110
8.7 Conclusion 111

9 Security Vulnerabilities 113
9.1 Dark Side of the Code Revisited 113
9.2 Methodology 114
9.3 Struts Operation 116
9.3.1 OGNL 118
9.3.2 Struts Interceptors 119
9.4 Tracing the Evolution of a Security Control 119
9.4.1 Tampering with OGNL 121
9.4.2 Accessing properties 123
9.4.3 CookieInterceptor 125
9.5 Analysis of Security Control Evolution 126
9.5.1 The dark side of the code 126
9.5.1.1 Report bias 127
9.5.1.2 Security metric bias 128
9.5.2 Developer's blind spots 129
9.5.3 Opportunistic fix 130
9.5.3.1 Compatibility problems 130
9.5.4 Counter-intuitive mechanism 131
9.5.4.1 Assumptions about consumers 132
9.5.5 Evolution of phenomena 133
9.6 Conclusion 134

10 Conclusion 135

A Vulnerability Test Cases 140
A.1 Test cases structure and setup 140
A.2 Test cases 141

B False Positive Test Cases 147

References 149

List of Figures

2.1 A message displayed by the application 9
2.2 Microblog application code: action class 10
2.3 Microblog application code: user class 11
2.4 Application dependency graph 12
2.5 Library method code 13
2.6 Accessing URL through the application: expected operation . . 15
2.7 Accessing URL through the application: internal server 15
2.8 Accessing URL through the application: HTTP redirect 16
2.9 Accessing URL through the application: local files 17
2.10 Traces of message posting and user impersonation 20

3.1 Taxonomy of intrusion-detection systems, Debar et al [1] 24
3.2 Taxonomy of intrusion-detection systems, Axelsson [2] 26
3.3 Database growth curve for learning process, reprinted from [3] . 32

5.1 A sample trace `appTrace` 48
5.2 A filtering of `appTrace` trace defined in Figure 5.1 49
5.3 Set of strands from the example in Figure 5.2 51

6.1 `httpLog` trace of HTTP requests $\langle h_1, \ldots, h_{12} \rangle$ 55
6.2 Strands from `httpLog` partitioned by attribute `item` 56
6.3 Strands from `httpLog` partitioned by attributes `item` and `user` 57
6.4 A behavioral norm from `httpLog` partitioned by attribute `item` for operations {`method, action`} 58
6.5 Behavioral norms from `httpLog` partitioned by attributes {`user, item`} and operations {`method, action`} 58
6.6 Strands from the HTTP log for cart creation and updates 59
6.7 Pseudo-code of the attribute search algorithm 64
6.8 A fragment of a log from simulated system 65
6.9 A sample user behavior scenario for simulated system (simplified) 66
6.10 Pseudo-code of norm aggregation algorithm 69
6.11 Number of aggregate norms for different norm similarity thresholds (simulated system) 69
6.12 A fragment of a log from a simulated system including roles . . 70

6.13 Number of aggregate norms for different norm similarity thresholds for two configurations of simulated system with access control . 71
6.14 A fragment of a log from enterprise system 72
6.15 Number of aggregate norms for different norm similarity thresholds (enterprise system) 74
6.16 A partial log from the photo hosting service 76
6.17 Norms for user's collaboration with photo hosting service provider 78
6.18 A log of two collaborating systems 79

7.1 A trace fragment of message posting and user impersonation . . 84
7.2 Traces for posting message with `http` and `file` URLs, lower abstraction scope 85
7.3 Sample strands of microblog application 86
7.4 Sample bi-gram sets for strands in Figure 7.3 87
7.5 Pseudo-code of runtime verification algorithm 90
7.6 Java Aspect for the microblog application calls 95
7.7 Java Anomaly Manager integration 96

8.1 Experiment setup 99
8.2 Growth in behavioral norms 101

9.1 A sample MVC Struts application code 117
9.2 OGNL context in the example application 118
9.3 Phenomena life cycle 134

List of Tables

4.1 Execution traces at different levels of abstraction 39

6.1 Norm model similarity calculation example 63
6.2 Some norms in the simulated system 67
6.3 Some norms in the enterprise system 73

7.1 N-gram groups 89
7.2 N-gram group arrangements 89
7.3 Runtime verification: anomaly-free sequence 92
7.4 Runtime verification: anomalous sequence with an unknown event 92
7.5 Runtime verification: anomalous sequence with an unknown n-gram 93
7.6 Runtime verification: anomalous sequence with an illegal arrangement 94

8.1 Attack outcomes on different versions of Struts 104

9.1 Security mechanism evolution: 2004–2015 120

B.1 False positive test cases outcomes on different versions of Struts 147

I, Olgierd Pieczul, certify that this thesis is my own work and I have not obtained a degree in this university or elsewhere on the basis of the work submitted in this thesis.

Olgierd Pieczul

Abstract

Contemporary application systems are implemented using an assortment of high-level programming languages, software frameworks, and third party components. While this may help to lower development time and cost, the result is a complex system of interoperating parts whose behavior is difficult to fully and properly comprehend. This difficulty of comprehension often manifests itself in the form of program coding errors that are not directly related to security requirements but can have an impact on the security of the system.

The thesis of this dissertation is that many security vulnerabilities in contemporary software may be attributed to unintended behavior due to unexpected execution paths resulting from the accidental misuse of the software components. Unlike many typical programmer errors such as missed boundary checks or user input validation, these software bugs are not easy to detect and avoid. While typical secure coding best practices, such as code reviews, dynamic and static analysis, offer little protection against such vulnerabilities, we argue that runtime verification of software execution against a specified expected behavior can help to identify unexpected behavior in the software.

The dissertation explores how building software systems using components may lead to the emergence of unexpected software behavior that results in security vulnerabilities. The thesis is supported by a study of the evolution of a popular software product over a period of twelve years. While anomaly detection techniques could be applied to verify software verification at runtime, there are several practical challenges in using them in large-scale contemporary software. A model of expected application execution paths and a methodology that can be used to build it during the software development cycle is proposed. The dissertation explores its effectiveness in detecting exploits on vulnerabilities enabled by software errors in a popular, enterprise software product.

Acknowledgements

I wish to express my sincere appreciation to those who have contributed to this thesis and supported me over the last few years.

In particular, I wish to express my gratitude to my supervisor Simon Foley for the opportunity to learn from his knowledge and experience. His high standards combined with enthusiastic support created an invaluable environment to develop myself as a researcher.

I would like to thank my examiners, John Clark and Barry O'Sullivan for a comprehensive, but enjoyable, examination and many suggestions that helped me to improve my thesis.

I would also like to thank IBM Ireland for sponsoring my research and, especially, Mike Roche for his valuable advice and guidance.

Last, but not least, special thanks to my wife Izabela. Her love, support and encouragement provided me with energy to keep up with all the hard work.

Chapter 1

Introduction

1.1 Contemporary Software Development

Contemporary application systems are implemented using an assortment of high-level programming languages, software frameworks and third party components. Current software frameworks enable developers to focus on the high-level functionality of an application by hiding low-level details. System infrastructure details such as DBMS, local file systems and memory are encapsulated as object storage; network connectivity is abstracted in terms of remote resource access, user interaction and presentation is supported via a range of standard interfaces.

While this may help lower development time and cost, the result is a complex system of interoperating parts whose behavior is difficult to fully and properly comprehend. This difficulty of comprehension often manifests itself in the form of program coding errors that are not directly related to security requirements but can have a significant impact on the security of the system [4, 5].

Interoperation of components, often implicit and outside of the high-level application logic, may lead to enabling unintended and unexpected execution paths in the system. Also, at the high level of abstraction, it may be difficult to anticipate if and what low-level security controls should be considered. For example, while an application may enforce the correct access controls in its high-level logic, its programmer may have mistakenly relied on the software development framework providing particular code injection defenses; alternatively, the framework developer may have mistakenly relied on its consumers implementing their own injection defenses, or, simply, that nobody had anticipated and/or understood

the injection vulnerability.

The software industry's approach to vulnerabilities includes security quality assurance processes such as code reviews, static analysis and penetration testing. This approach may be quite effective in identifying typical and expected programming errors such as buffer overflow and code injection. These simple vulnerabilities often appear as easy to identify coding patterns or as a specific application response to a typical malicious input. The software industry has developed standard lists of typical problems, such as OWASP Top 10 [6] or SANS 25 [7], together with methodologies to identify them and best practices to avoid them. As a manual review is often time-consuming and costly, there is a range of automated tools that assist in identifying these types of problems through static analysis of the source code, or dynamic analysis of application operation.

In contrast, vulnerabilities resulting from, often accidental, misuse of an integrated software component are very difficult to identify. In the case of a manual review, the reviewer must be aware of the possibility that a particular component used in a given way may result in an unintended behavior of the application. However, when programming at a high level of abstraction it is rarely possible to cover the subtleties of all of the numerous components and their interoperation. Even simple, and what may seem, obvious problems resulting from component usage may be difficult to identify. For example, JSON Web Token (JWT) is a popular standard for signed access tokens transferred between web parties. Naturally, the software community developed a number of libraries that allow creation, verification and processing of the tokens. However, in-line with the standard, but unexpectedly to the consumers [8], the libraries often accepted tokens with a `none` signature algorithm and empty signature, as valid. Surely, the developer might have included a check to verify the signature algorithm, however, that would have required them to be aware of the problem in the first place. Similarly, the reviewer may not expect that the library, when called to verify the token with a specific key, will accept tokens with no signature. Also, any tool that assists in software vulnerability detection would not be able to find the bug unless specifically configured for this problem. This is not unlikely as the problem was generally unknown and eventually considered a vulnerability in a number of JWT libraries [8].

It could be debatable whether the problem with the JWT implementation was, in fact, a library vulnerability. RFC 7519 prescribes that every implementation must support the `none` algorithm. The root cause of this issue should be rather

viewed in terms of a misunderstanding between a developer of a library and a developer of an application. The library developer may assume that the consumer will verify that the token is signed with the right algorithm and focuses on just verifying the signature, however specified. The developer of an application using the library, in turn, may expect the library to ensure that the token is verified against a specific key they provided to the library. While technically the problem should probably be attributed to bad API design and poor documentation of libraries, it demonstrates the general problem of development using components. It creates a potential gap between the expected behavior of an application (for example, that it verifies tokens using the algorithm for the provided key) and the actual behavior (for example, that it verifies tokens, regardless of the algorithm). The higher the abstraction, and more abstraction layers are included, the larger the potential gap. For example, it is common that the application does not use the token verification library directly, but rather some authentication framework, that in turn uses the library.

One could argue that it is the responsibility of the consumer to understand the underlying standard, and comprehensively review the documentation and the source code, if available, of the component. However, this would defeat the very purpose of building the application from components in order to separate low-level concerns from the core application logic. In practice, during application development, third-party components are not routinely reviewed or tested for vulnerabilities. Some of the security vulnerabilities that gained notoriety in recent years, such as Heartbleed and Shellshock, were deployed for many years before they were discovered, despite being used by a large number of consumers and developers. Also, the number of components used by contemporary software and its dependencies is so large that even just managing them has emerged as a challenge [9], never mind considering their security individually or when combined together. For example, at the time of writing, the latest (December 2016) versions of common web frameworks, `struts2`, `play` and `spring-web` add a further 24, 29 and 44 dependencies (respectively) to the application.

1.2 Detecting Vulnerabilities as Anomalies

Our hypothesis is that many software vulnerabilities can be attributed to programming errors that enable unexpected software behavior. As these vulnerabilities result in different from the intended activity of the application, it is

worth investigating ways to systematically detect their exploitation. Runtime verification [10] of software execution against a specified expected behavior may help identify unexpected behavior in the software. For small applications, limited requirements on expected behavior can be specified a priori, for example as a temporal proposition [10]; however, this approach does not scale when the requirement is to constrain the behavior across large complex systems of interoperating components.

An alternative strategy used by anomaly detection techniques [11] is to learn a behavioral reference profile from system logs of past normal behavior and use this profile at runtime to validate application operation. In this context, the unexpected operation of the application under an attack resulting from a vulnerability could be considered an execution anomaly. Anomaly detection techniques have been applied to many types of system activity, including: network traffic [12], program execution context [13] or sequences of program operations such as operating system calls [14, 15] or JavaScript [16]. While the monitoring of operation sequences is the most commonly proposed technique for detecting software vulnerabilities, this approach has only been demonstrated in the context of operating system calls in rather small applications. It is an open question if it could be applied to modern software built as an arrangement of interacting components. The practical challenges of applying these techniques to contemporary enterprise software, such as dealing with complexity introduced by a large number of components and alignment with software release processes focused on fast and continuous delivery, have not been considered.

In this dissertation, we study the nature of vulnerabilities resulting from unexpected application behavior due to component misuse and their detection as execution anomalies. Focusing on Java enterprise applications, we present typical vulnerabilities resulting from component misuse and demonstrate how easy it is for the programmer to accidentally introduce them in the code. We also provide an in-depth case study of Apache Struts security controls, vulnerabilities, and their remediation over twelve year period to confirm our findings. We also analyze what effect the exploitation of such vulnerabilities has on the operation of an application, in particular, if any deviation from the typical application activity can be observed. This leads to an observation that for different types of vulnerabilities the deviation can only be identified at some specific view of application operation, such as input/output operations or sequences of method calls of a specific component.

In this dissertation, we in investigate what are the challenges of applying anomaly detection in large-scale contemporary enterprise software. Current research of anomaly detection applied to software execution and vulnerabilities is typically considered in the context of system calls [14, 15, 17–19] and relatively simple UNIX-style applications such as `sendmail`, `lpr` or even `rm`. While the system-call view of application operation may be appropriate for such small programs, it is not appropriate for complex systems including the platform, application servers, frameworks and libraries. In addition, the operation can be viewed at multiple levels of abstraction from the logs of user actions to operating system calls. The combined activity of all these elements is too extensive to be useful in practice. We discuss the challenges of selecting the appropriate level and reducing the scope of the system activity. Also, building a reference model of the normal behavior of a system requires collecting the baseline activity of the system in a comprehensive manner. As today's software is developed, built and deployed fast, often on a daily basis, and in a highly automated manner, the process for establishing the baseline behavior has to operate in a similar way. We investigate potential mechanisms for acquiring a normal expected activity of an application based on existing software development tools and practices such as functional testing or security scanning.

Further, we consider the characteristics of the abstraction that would be appropriate for modeling the behavior of contemporary systems for the purpose of anomaly detection. We observe that many contemporary systems, such as web applications, operate by processing distinct transactions. Therefore, we propose a model of behavioral norms: a generic framework for inferring repeating transaction-like patterns of behavior from the logs of system operation. We propose a practical interpretation of the model using set of short-range correlations between subsequent events (n-grams) and a mechanism to automatically identify model parameters. We also discuss potential applications of the model, beyond anomaly detection, such as monitoring system changes or identifying underlying, emergent behaviors.

Finally, we discuss the implementation and integration strategies that enable runtime verification of application activity using the model. The approach has been evaluated by considering its effectiveness in identifying code vulnerabilities across the twenty-six versions of Apache Struts over a period of five years.

The contributions contained within this dissertation are as follows:

- the introduction of the phenomenon of "dark side of the code", the unexpected behavior of the application resulting from accidental misuse of the software components, and a longitudinal study to confirm its existence in real-world systems (Chapters 2 and 9);
- a formal model of behavioral norms with a general purpose framework for inferring transaction-like behavioral patterns from system logs (Chapters 5 and 6), and
- an analysis of the challenges of applying anomaly detection to complex contemporary software systems including a prototype implementation evaluated in a large-scale experiment (Chapters 4, 7 and 8).

1.3 Thesis Layout and Previous Work

This dissertation is organized as follows. In Chapter 2 we discuss the dark side of the code and introduce a running example of contemporary software development to illustrate how easy it is for a programmer to unwittingly introduce a programming flaw/security vulnerability. Chapter 3 reviews existing anomaly detection techniques with the focus on those techniques that could have the potential to be useful in detecting anomalies in software execution. Chapter 4 considers the key challenges encountered in applying such techniques to contemporary application software. In Chapter 5, we present the model of behavioral norms and in Chapter 6 we explore it in the context of possible applications. In Chapter 7 we present the interpretation of the behavioral norms model applied to traces of Java applications, a runtime verification algorithm that is based on the model and practical issues related to its implementation. Chapter 8 discusses the experimental evaluation of the runtime verification mechanism, based on the full set of nineteen vulnerabilities reported for twenty-six versions of Apache Struts over a five year period. Chapter 9 presents the results of a longitudinal study on root causes of the security vulnerabilities in Apache Struts over 12 years.

Earlier versions of the work described in this dissertation were published in the following peer-reviewed papers.

- O. Pieczul and S. N. Foley. Discovering emergent norms in security logs. In *2013 IEEE Conference on Communications and Network Security (CNS - SafeConfig)*, pages 438–445, 2013.
- O. Pieczul and S. N. Foley. Collaborating as normal: Detecting systemic anomalies in your partner. In *Security Protocols XXII: 22nd International Workshop, Cambridge, UK, March 19-21, 2014, Revised Selected Papers*, pages 18–27. Springer International Publishing, 2014.
- O. Pieczul, S. N. Foley, and V. M. Rooney. I'm OK, You're OK, the System's OK: Normative security for systems. In *Proceedings of the 2014 Workshop on New Security Paradigms*, NSPW '14, pages 95–104, New York, NY, USA, 2014. ACM.
- O. Pieczul and S. N. Foley. The dark side of the code. In *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, pages 1–11. Springer International Publishing, 2015.
- O. Pieczul and S. N. Foley. The evolution of a security control. In *Security Protocols XXIV: 24th International Workshop, Brno, Czech Republic, Revised Selected Papers*. Springer International Publishing, 2016. (to appear).
- O. Pieczul and S. N. Foley. Runtime detection of zero-day vulnerability exploits in contemporary software systems. In *Data and Applications Security and Privacy XXX: 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*. Springer International Publishing, 2016.

Chapter 2

The Dark Side of the Code

"Beware of bugs in the above code; I have only proved it correct, not tried it."

—Donald Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion, 1977

2.1 Introduction

Given the complexity of contemporary applications, and the manner of their development, we argue that there will always be some aspect of their behavior (ranging from application level to low-level system calls) that a programmer may not have fully considered or comprehended. We refer to this as the *dark side of the code*: the security gap that can exist between the behavior expected by the programmer and the actual behavior of the implemented code. Improper or incomplete comprehension means that security controls may not have been considered this the security gap and, as a consequence, the unexpected behavior arising from the code may give rise to a security vulnerability.

Even if the developer is aware of a particular coding vulnerability, they may fail to correlate it with their own programming activity unless it is explicitly highlighted [26]. It is, therefore, not surprising that all of the OWASP Top 10 security risks [6] relate to implementation flaws, with the majority in the form of common coding mistakes. In addition, as the applications grow larger, the understanding of old, legacy or just unchanged code decreases [27].

One might argue that encapsulation and programming by contract [28] could eliminate these security gaps; or, that one might attempt to model all unexpected behaviors in the security gap in terms of a Dolev-Yao style attacker [29, 30] and verify that the application code is in turn robust to failure against this attacker. However, these approaches still require a full and proper comprehension of system components and their interoperation (in terms of formal specification) which, in itself, can have security gaps [31], let alone the challenge of scaling these techniques to contemporary application systems.

In this chapter, we introduce a running example to explore the coding of a contemporary application. Despite being simple, the example is sufficient to illustrate some challenges in using these frameworks and to demonstrate unexpected vulnerabilities arising from the security gap. Further, we present the security gap between the expected and the actual behavior of the application as a source of security vulnerabilities. Finally, we demonstrate how traces of application execution could be used to identify exploitation of vulnerabilities that arise from the security gap.

2.2 A Microblog Application

A microblog social networking application provides an on-line facility for users to post/share short text messages. The application provides a web interface and a REST API for integration with various types of clients, including desktop browsers and mobile devices. In addition, it parses posted messages, looking for any links and creates snapshots that are displayed with a message. For example,

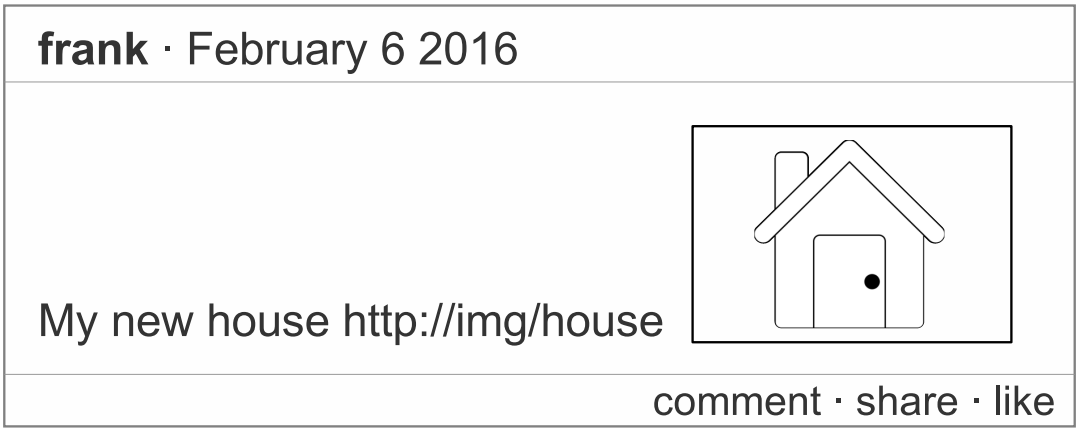


Figure 2.1: A message displayed by the application

Figure 2.1 depicts a message *"My new house http://img/house"* posted by user `frank`, displayed by the application. The application is built using Apache Struts, a popular Model-View-Controller framework for J2EE. Struts abstracts application logic from HTTP request processing flow by encapsulating it into objects called *actions*. Figure 2.2 shows how the action responsible for posting a new message is implemented in just a few lines of code.

```
1  public class PostAction extends ApiAction {

3      private String text;
4      public void setText(String text) {
5          this.text = text;
6      }

8      public String execute() throws Exception {
9          Message message = new Message(text);
10         message.setAuthor(getUser());
11         String[] urls = TextUtils.getUrls(text);
12         for (String URL : urls) {
13             message.addLink(url);
14             Image snapshot = WebUtils.snapshot(url)
15             if (snapshot != null)
                   message.addImage(snapshot);
16         }
17         DAO.add(message);
18         return SUCCESS;
19     }

21     public User getUser() {
22         return session.get("user");
23     }
24 }
```

Figure 2.2: Microblog application code: action class

Actions can be mapped to specific URL paths, such as `/api/post`. Struts handles routine tasks, including parameter validation, authentication, session handling and CSRF protection. These actions enable Struts to set the HTTP

parameters using setters. For example, a call to the application including the `text` parameter, such as

```
/api/post?text=My+new+house+http://img/house
```

results in Struts calling a `setText` method to provide the parameter value to the action. This facilitates the separation of business logic from HTTP processing concerns. Thus, rather than dealing with low-level operations such as accessing parameters by name, the developer need only implement a public setter. This makes the code reusable, easier to maintain, document and test.

Lines 10 and 21 of Figure 2.2 refer the User class. It is defined as a simple container for user properties such as a name and a unique user identifier. Figure 2.3 provides a listing of the User class. It includes value fields such as user name and identifier with public methods to get and set them, commonly referred to as *getters* and *setters*. The application uses the User class as a container to encapsulate all user properties. The application records a current user in the server's session as an instance of `User` class. This value is available to the action class

```
1  public class User {
2      private String name;
3      public String getName() {
4          return name;
5      }

7      public void setName(String name) {
8          this.name = name;
9      }

11     private String id;
12     public String getId() {
13         return id;
14     }

16     public void setId(String id) {
17         this.id = id;
18     }
19 }
```

Figure 2.3: Microblog application code: user class

through the `session` variable set by Struts. Action class encapsulates access to the current user with the `getUser` getter (Figure 2.2, Line 22).

While the high-level application code is clear and easy to follow, the program abstractions that are used mean that the typical programmer will not overly concern themselves with the specifics of the low-level behavior of the underlying framework infrastructure. For example, at Line 17 the application uses the persistence framework to save the message. The developer expects that the framework will make a connection to a database (or reuse an existing one), formulate an SQL statement from the message object fields and execute it.

Similarly, tasks such as parsing the message in order to look for web links or obtain a snapshot for a URL are performed by third party libraries `TextUtils` and `WebUtils`. Figure 2.4 shows a dependency graph of the application. Note

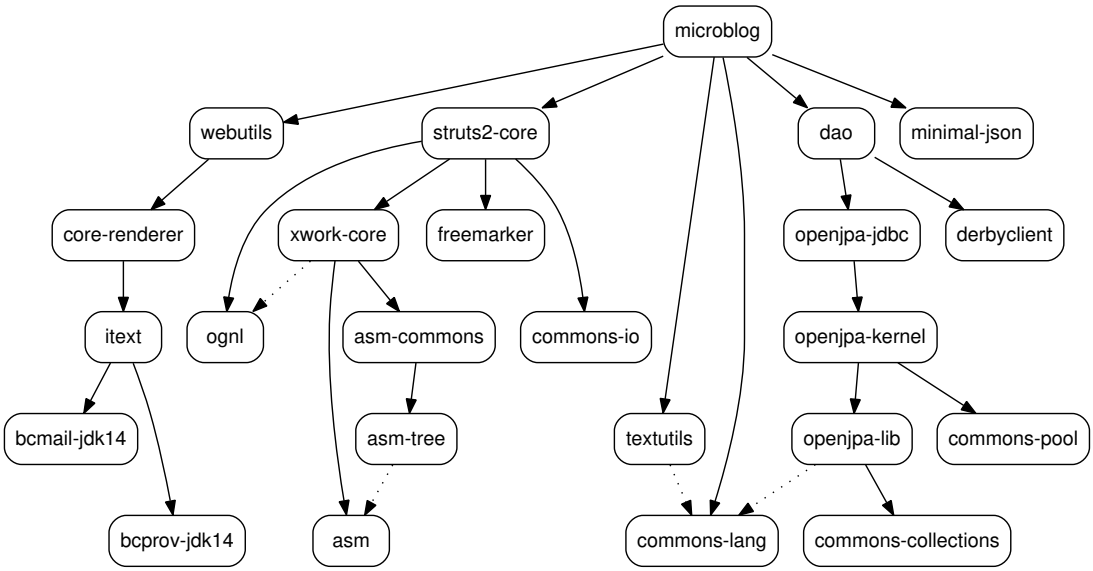


Figure 2.4: Application dependency graph

that it includes only explicit dependencies of the application and its libraries. The application server (with its own dependencies) and Java runtime are not included. Also, a number of dependencies for common tasks, such as logging or servlet API, are not included in the graph.

Documentation, such as Javadoc provided with the `WebUtils.snapshot` source code in Figure 2.5, gives only a limited information about the method behavior. From this, the developer can learn that it accesses the website specified by the URL, renders it, and returns an image of specified dimensions. The programmer can expect that the library will verify the correctness of the provided

address (as an exception is thrown for an "incorrect URL"), and that it will check for "communication problems" while the website is being accessed.

Studying the source code of the `WebUtils.snapshot` method in Figure 2.5, we can see that the library, used by the application, is also implemented at a similarly high level of abstraction. All logic related to accessing the remote web-

```
1  /**
2  * Create an image snapshot for a website
3  * @see    #render(InputStream)
4  *
5  * @param  website URL address of the website
6  * @param  w       image width
7  * @param  h       image height
8  * @return         Image containing website snapshot
9  * @throws IOException communication problem
10 * @throws MalformedURLException incorrect URL
11 */

13 static public Image snapshot(String website, int w, int h)
       throws IOException, MalformedURLException {
14     URL url = new URL(website);
15     URLConnection connection = url.openConnection();
16     InputStream input = connection.getInputStream();
17     Image image = RenderEngine.render(input, w, h);
18     return image;
19 }
```

Figure 2.5: Library method code

site in order to create the snapshot is covered in Lines 13–15 using the Java Platform API `URL` and `URLConnection` classes. Looking at the first lines of its documentation, the developer learns that a *"Class `URL` represents a Uniform Resource Locator, a pointer to a 'resource' on the World Wide Web"*. The documentation informs its reader that in the case of a malformed URL, the constructor will throw an exception. Furthermore, the documentation specifies that `URL.openConnection` method returns *"a connection to the remote object referred to by the URL"*. The `URLConnection` class is explained in the documentation as: *"The abstract class URLConnection is the superclass of all classes*

that represent a communications link between the application and a URL". The `URLConnection.getInputStream` *"returns an input stream that reads from this open connection"*. The documentation also states that *"if the read timeout expires before data is available for read"*, a `SocketTimeoutException` is thrown. The language of the documentation also abstracts the low level details by referring to fairly generic concepts such as connections, communication links or remote objects.

2.3 The Security Gap

The convenience of using abstractions and their ability to handle security threats relieves the developer from having to consider much of the low-level details. For example, because object persistence frameworks do not require construction of SQL queries, the programmer need not consider sanitizing user input in order to prevent injection attacks. Similarly, allowing the MVC framework to provide the Web presentation layer can reduce programmer concerns about application output interfering with the output context, such as HTML, XML and JSON [32]. This does not excuse the programmer from considering security issues entirely, rather the emphasis is on the security controls that are relevant to the application code.

2.3.1 Accessing arbitrary URLs

Regardless of the effectiveness of the programming abstractions, it is reasonable to expect that the developer does understand some of the underlying system operation in order to identify possible threats and to counter them with adequate security controls. For example, although not directly referenced in the application code, it may be anticipated that the application will communicate over HTTP with the remote website in order to create a snapshot. Figure 2.6 depicts a typical flow of operations that relate to creating a website snapshot and delivering an image to the user. A user sends a URL, the application connects to the server and accesses a website, generates an image and sends it to the user. Thus, the application should be permitted to make HTTP connections that are, to some degree, controlled by application users through the URLs they enter, and this may be a security threat.

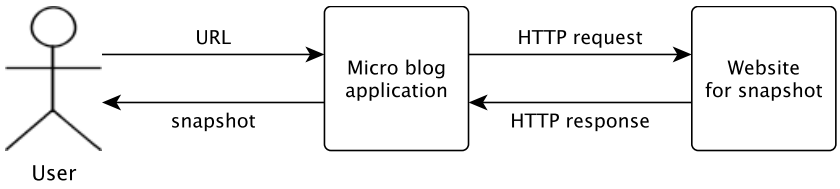


Figure 2.6: Accessing URL through the application: expected operation

2.3.1.1 Local network access

The application could be used to access systems—in the local network where it is hosted—that are not normally accessible from the Internet. A malicious user may, by posting a message with the URL in the local network, such as `http://10.0.0.1/router/admin`, attempt to access systems that he should not have access to.

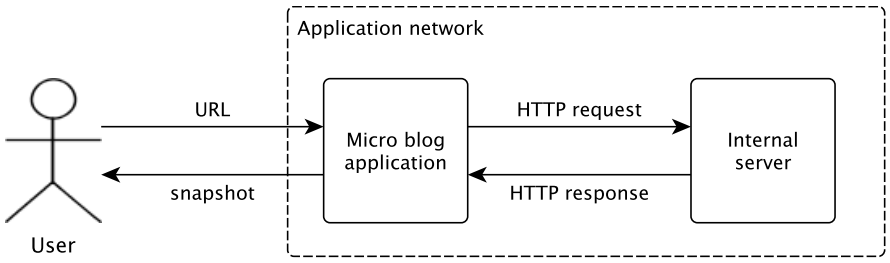


Figure 2.7: Accessing URL through the application: internal server

In order to address this threat, the developer can code a security control in the application that verifies that the URL's host does not point to a local IP address, before calling the library to create a snapshot. For example, the `execute` method can include:

```
InetAddress a = InetAddress.getByName(url.getHost());
if (a.isSiteLocalAddress()) throw new SecurityException();
```

To a casual reader, the microblog application (or even the `WebUtils` library) code does not openly perform TCP/IP operations. The above threat was identified based on the programmer's expectation of low-level application behavior. Correlating high-level application behavior (accessing URLs) with the threat (user-controlled network traffic) is a human task and, as such, is prone to human error. Failure to implement adequate security controls may not necessarily mean that the developers are unaware of the threat or neglect security. As observed previously, despite understanding a security problem, a developer may unwittingly write code containing a vulnerability [26]. The cognitive effort that

is required to anticipate security problems is much greater if the details are abstracted and the system contains interoperating components whose behavior is not considered or understood.

2.3.1.2 Redirect to a local site

Consider again the microblog application extended with the security control to prevent local URL access. Despite appearances, an attacker can bypass this check as follows. We first note that the HTTP protocol (RFC 2616) allows a server to redirect the client to another URL in order to fulfill the request through a defined status code (such as 302) and a header. As depicted in Figure 2.8 the attack could be performed as follows.

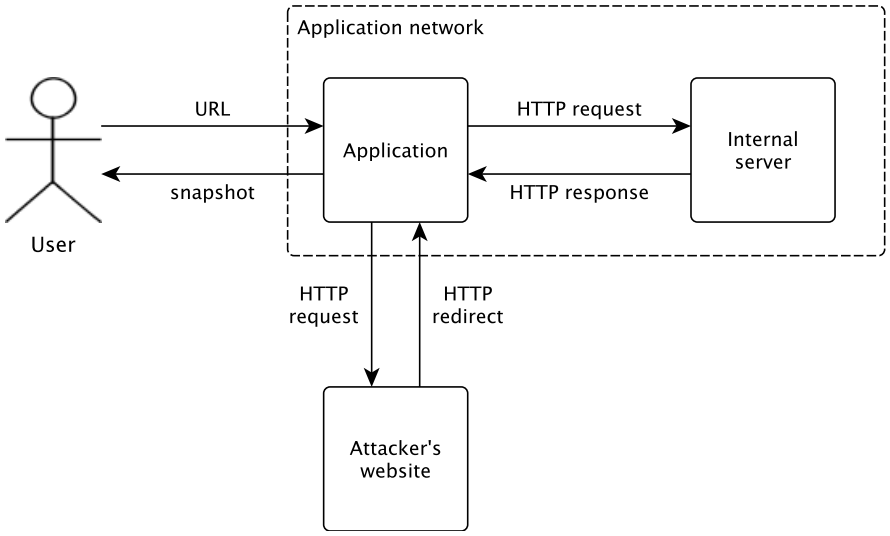


Figure 2.8: Accessing URL through the application: HTTP redirect

- An attacker sets up a website that redirects to the local target machine and posts a message with URL to that website.
- The attacker's website (public) URL will be accepted as not local and `WebUitls.snapshot` is called.
- The Java library will access the website through `url.openConnection` in the implementation of `snapshot`, follow the redirection, effectively connecting to a local address.

In order to prevent this attack, it is necessary for the programmer to modify the utility library to explicitly handle redirects and verify the IP address each time, before accessing the URL. This approach, however, may suffer a TOCTTOU

vulnerability. In this case, there is a time gap between the verification of the IP address and the HTTP connection to the corresponding URL. Within that time gap, the mapping between the hostname and the IP address may be modified. While past responses will typically be cached by the resolver, the attacker may prevent the caching by creating a record with lowest possible Time To Live value supported by Domain Name System, that is, 1 second. This is a variant of a DNS Rebinding attack [33].

Perhaps, and rather than trying to implement the network-related security controls in the application, a better strategy is to consider this a matter of the system's network configuration. In this case, it should be the systems and network administrators, not the developers, that handle the problem by implementing adequate firewall rules. While transferring administrative burdens to the consumer is a common practice [34], it also pushes the abstraction further and may make the threat equally difficult to identify.

2.3.1.3 Local file access

Regardless of how this network protection is implemented, the web application still contains an even more serious and unexpected vulnerability. It allows application clients to access custom files from the web server's file system. The access mechanism, called scheme, of a Uniform Resource Identifier can be `file`, in addition to `http` and `https`. In this case, as depicted on Figure 2.9,

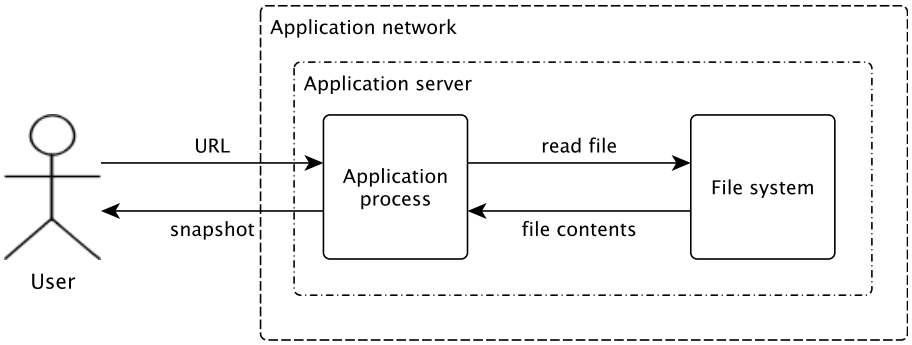


Figure 2.9: Accessing URL through the application: local files

- the application client posts a message with URL `file:///etc/passwd` and,
- the application generates an image representing file contents.

This behavior may not have been anticipated by the application developer who, upon reading associated library documentation, understood that the method `WebUtils.snapshot` should be called with a *"website address"* and which throws an exception if that address is *"incorrect"*. Similarly, the library developer might have been misled by the Java `URL`/`URLConnection` documentation and method names referring to *"connections"* and *"sockets"* and did not expect that their code could be used to access regular files. While, the `URL` class Javadoc includes a reference to `file` URL scheme, it is mentioned only once, in one of the constructors' documentation. Other platforms include similar, often misleading, features. For example, the function `file_get_contents` in PHP, despite its name, allows accessing remote resources if a URL is provided as a file name.

To avoid this vulnerability, the developer must implement specific code that checks whether the URL specifies a website address. However, other URL-related problems may emerge. For example, in another part of the application, it may be required to verify whether a URL matches a set of accepted URLs. The Java `equals` method can be used (explicitly or implicitly via the `Set.contains` method) as a standard way to test object equality. Using this method is convenient when comparing URL objects, as it respects that a hostname and protocol are case-insensitive and some port numbers are optional. For example, `http://example.com/`, `http://example.com:80/` and `HTTP://EXAMPLE.COM/` are equal URLs despite their different string representation. What may not be anticipated by the programmer, is that when comparing two URL objects, the `equals` method resolves their hostnames and considers them equal if they point to the same IP address. In this case the URL `http://example.com/` is considered *equal* to `http://attacker.com/`, provided that the attacker has targeted their host to `example.com`'s IP address. This unexpected behavior may lead to security vulnerabilities if URLs are used for white/black listing, or to ensure the Same Origin Policy (RFC 6454). While the behavior of `URL.equals` is documented and the corresponding security issues are considered [35], the developer may not consider checking that part of documentation to be necessary, especially if the code may not explicitly invoke the method.

2.3.2 Modifying internal state of application

The application contains yet another vulnerability, resulting from its interoperation with Struts framework. In Struts, elements of business logic are im-

plemented using *actions*, such as the `PostAction` in Figure 2.2. Actions are separated from the low-level details such as the handling HTTP requests. This allows code to be reusable in different contexts and allows developers to focus on business logic. For instance, Struts processes HTTP request parameters and makes them available to the action. When parameters are processed, the framework looks for action setters that match the parameter names and invokes those action setters in order to set the value. This mechanism is discussed in detail in Chapter 9. For example, the `PostAction` contains public method `setText` to handle the `text` parameter. When client calls `/api/post?text=Hello`, Struts calls `PostAction.setText("Hello")` to provide the value to the action, before invoking `PostAction.execute`.

A developer, familiar with Struts, will most likely be aware that request parameters may be used to call public setters defined by the action. They may not expect, however, that defining a public getter method `getUser` may make the user's session exposed to manipulation. Suppose that this method was not intended to be a part of the action's interface, rather, it is implemented by the programmer as a means to provide convenient access to the user object from the session. The method returns a `User` object, which is a container for various user attributes, such as name and identifier, and corresponding getters and setters. The combination of a public getter returning the `User` object and public setters within User class makes it possible to manipulate user information request parameters. For example, an attacker Alice can invoke `/api/post?text=Hello&user.id=frank` in order to modify the user-id stored in the session to `Frank` and submit a message on Frank's behalf. The developer may not be aware, that Struts method names are interpreted as OGNL [36] expressions and, and that `user.id=frank` is translated into a sequence of method calls corresponding to `getUser().setId("frank")`.

This oversight by the developer illustrates how easy it can be to program an unexpected execution path that compromises security. Accessing object fields through a chain of getters and setters is a useful and documented feature of Struts. However, it can come as a surprise to a programmer, who is focused more on application-level development, that something as intuitive as implementing a getter for the current user results in a security vulnerability.

This problem is further highlighted by an almost identical vulnerability (CVE-2014-0094) that was found within Struts itself. Even though the consequences of exposing a public getter may have been clear to the Struts developers, it is easy

to overlook the fact that every Java object (and therefore, every Struts action) also contains a `getClass` method. This results in an unintended exposure of an action's Class object through request parameters. By crafting a parameter, such as `class.classLoader...`, the attacker can manipulate the application's class loader and, as a result, execute custom code on the server hosting the application [24]. Unfortunately, even after the problem was discovered (CVE-2014-0094), the implemented remedy was incomplete. The initial remedy blacklisted the `class` parameter, however, it did not consider uppercase parameters (such as `Class`). Eventually, three more vulnerabilities (CVE-2014-0112, CVE-2014-0113 and CVE-2014-0116) were reported on incomplete remedies before the issue was believed addressed. The evolution of Struts vulnerabilities related to processing request parameters is discussed in detail in Chapter 9.

2.4 Unexpected Behavior Traces

The vulnerabilities in the microblog application result from the unexpected, though completely valid, operation of the application. This activity may mean that the application executed a different code path than was intended by the developer. For example, Figure 2.10 presents two traces of application activity. These traces include Java method calls of, or by, the microblog application

```
ParamsInt: PostAction.setText("Hello")

Dispatcher: PostAction.execute()
PostAction: PostAction.getUser()
PostAction: Message.setAuthor([USER])
PostAction: TextUtils.getUrls("Hello")
PostAction: DAO.add([MESSAGE])
PostAction: PostAction.return(SUCCESS)
```

```
ParamsInt: PostAction.setText("Hello")
ParamsInt: PostAction.getUser()
ParamsInt: User.setId("frank")
Dispatcher: PostAction.execute()
PostAction: PostAction.getUser()
PostAction: Message.setAuthor([User])
PostAction: TextUtils.getUrls("Hello")
PostAction: DAO.add([MESSAGE])
PostAction: PostAction.return(SUCCESS)
```

Figure 2.10: Traces of message posting and user impersonation

in a form of `Caller: Class.method(arguments)`. For example, the first line captures `ParamsInt` class calling `PostAction.setText` method . The left-hand side trace captures the posting of a message with a text "Hello". The right-hand side trace captures the posting of the same message while exploiting the getter vulnerability as presented in Section 2.3.2. It includes two additional method calls, corresponding to setting the user identifier.

This example demonstrates that it may be possible to identify an attack ex-

ploiting a vulnerability that arises from a programming error, by comparing the application's execution trace with a trace of typical, normal behavior. Our research hypothesis is that this could be generalized into a systematic technique to detect high-level application attacks as execution anomalies.

However, there are a number of challenges that should be considered. The example traces were deliberately chosen to include the view of application execution in which the attack is evident, that is, method calls of the application itself. There are many more method calls that are executed for message posting activity, methods of: the application server, Struts framework, numerous libraries and so forth. If a different view was chosen, for example, application server methods, the attack might not have been evident. Also, the comparison was made only against a single, closely matching trace that, apart from the anomalous fragment, included exactly the same order of events and, equal method call arguments. In practice, the record of typical application execution may include numerous traces with varying order of operations and method arguments that may be only relevant for a particular execution. A comparison against a set of past traces may result in frequent false positives, that is identifying harmless execution as anomalous, due to small differences in the trace structure or arguments.

To address this, some level of abstraction has to be introduced in order to allow approximate matching between the past and new traces. However, the approximation can not go too far, as that may, in turn, lead to false negatives, that is considering traces of malicious behavior as normal. For example, a behavior abstracted to just a list of Java classes contained in past traces would not allow the identification of the anomalous behavior in Figure 2.10.

2.5 Conclusion

Given the complexity of contemporary applications, we argue that there will always be a security gap between the code's actual behavior and the behavior expected by the programmer. We refer to this unexpected behavior as the dark side of the code. The cognitive overload on the programmer increases with the level of the programming abstractions used and, in turn, increases the likelihood of errors that lead to security vulnerabilities. The examples presented in this chapter illustrate how programming oversights, in what seems to be trivial, high-level application code, can result in a series of security issues whose identification and prevention requires an in-depth understanding of a framework, low-level libraries

and a number of network protocols. That these kinds of oversights do occur in practice, has been demonstrated in a longitudinal study of the Struts code changes [24], discussed in Chapter 9.

A malicious activity permitted by programming errors could potentially be identified by comparing application execution traces with traces of past, normal behavior of an application. We put forward a hypothesis that attacks attempting to exploit software vulnerabilities arising from the dark side of the code could be detected systematically as execution anomalies. Such detection, however, may be difficult in practice. In Chapter 3 we consider anomaly detection techniques that could be useful for this task.

Chapter 3

Anomaly Detection

3.1 Introduction

> "Exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage."

This hypothesis has been put forward in one of the earliest [37] studies on intrusion detection systems. It matches the observation in Chapter 2, that an exploitation of a vulnerability that arises from a security gap follows a valid, though unexpected, execution path of an application.

This chapter explores the area of intrusion detection [38], in particular techniques that could be potentially useful in identifying unexpected behavior in contemporary application systems. It provides a general overview of the key types and properties of intrusion detection systems. As our focus is on detecting anomalous behavior without a priori knowledge of specific misbehavior patterns, we examine existing techniques focusing on those that support some degree of automation in learning the behavior of the system. We also study the most promising techniques that have already been used to identify anomalies in software execution, and assess their suitability for the vulnerabilities caused by the kinds of programming errors discussed in Chapter 2.

3.2 Overview of Intrusion Detection Techniques

Intrusion detection is a large and increasingly growing field of computer security. Over the years a number of techniques have been developed that focus on different types of intrusions, using a variety of mechanisms. This led to a number of attempts to provide a systematic classification of intrusion detection systems. For example, the taxonomy depicted in Figure 3.1 [1] classifies IDS through four characteristics: detection method, behavior on detection, audit source location and usage frequency and provides key classes for each characteristic.

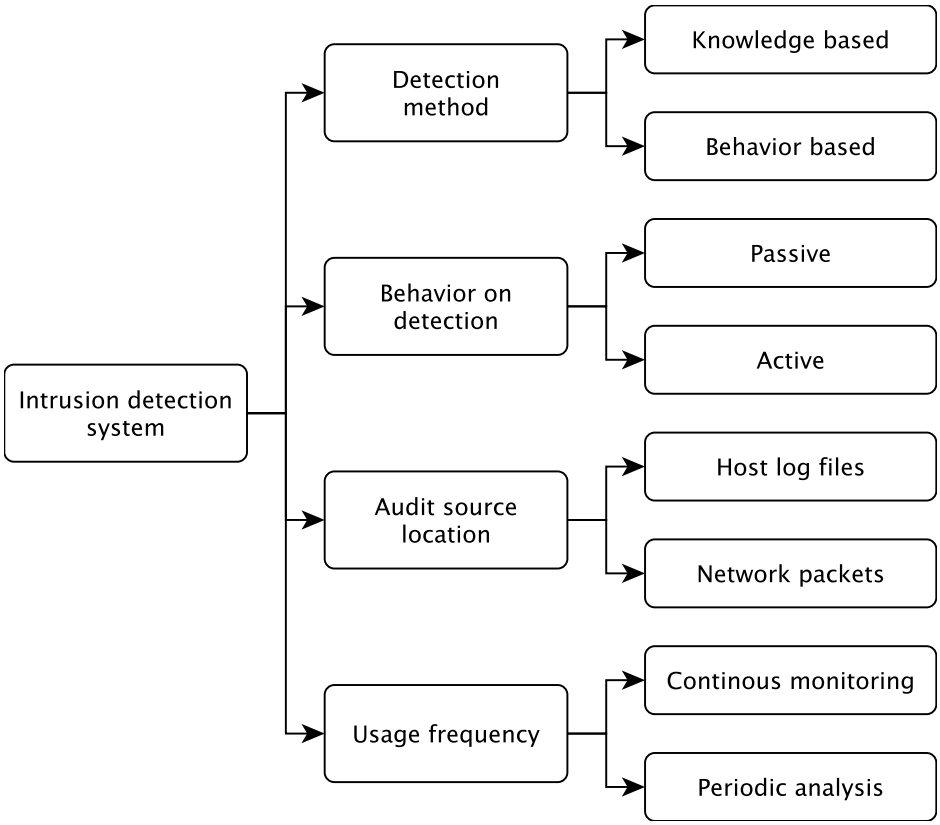


Figure 3.1: Taxonomy of intrusion-detection systems, Debar et al [1]

Each of the classes contains further sub-classes that are not presented in Figure 3.1. For example, the *knowledge-based* detection class includes expert systems, signature analysis, Petri nets and state-transition analysis. This classification was later extended to include more characteristics and classes [39].

Characteristics can describe technical means of detecting an intrusion as well as operational aspects of existing IDS systems. For example, following the categories in Figure 3.1, the *behavior on detection* may be *active*, that is, provide a countermeasure for the attack, or *passive*, for example, issue an alert. This dis-

tinction is typically a characteristic of a specific system and the functions it offers, rather than a property of the underlying detection technique. Some techniques, however, are only suitable for one type of reaction to an intrusion.

Similarly, considering the host or network *audit source location* is not necessarily important. While most of the anomaly detection techniques applied to software are host-based [2, 15], observing network activity of a system may allow for the systematic identification of unexpected behavior. Today, a network is used not only for communication between the system and its environment, such as users or infrastructure, but also for its core, internal operation. Many contemporary applications are built using a micro-service architecture, where a number of independent components communicate over a network using well-defined protocols. Also, many systems use cloud services for storage and processing and software behavior could be analyzed through the interfaces on which they communicate [21].

Another characteristic in Figure 3.1 is *usage frequency*. This characteristic is also not relevant to our investigation. While continuous monitoring may be preferred for the runtime detection of anomalies in software execution, it may be also useful to perform an analysis of software logs periodically or after an incident. Also, some detection techniques are not capable of operating fully on-line and can only identify anomalies when analyzing complete portions of activity, a posteriori.

These three characteristics: *behavior on detection*, *audit source location* and *usage frequency*, are to some extent not relevant to our research as they mostly reflect implementation and operational details of the actual intrusion detection products.

Another taxonomy [2], classifies IDS according to two key groups of properties, depicted in Figure 3.2. Here, the characteristics related to detection techniques are considered separately from other, operational characteristics of the system. Our primary focus is on the "detection method" [1] or "detection principles" [2] characteristic. *Knowledge-based* [1] or *signature* [2] detection focuses on identifying attacks based on knowledge accumulated from known incidents. The goal of *behavior-based* [1] or *anomaly* [2] detection is to identify system behavior that is unusual when compared with activity of the system observed during a known normal state.

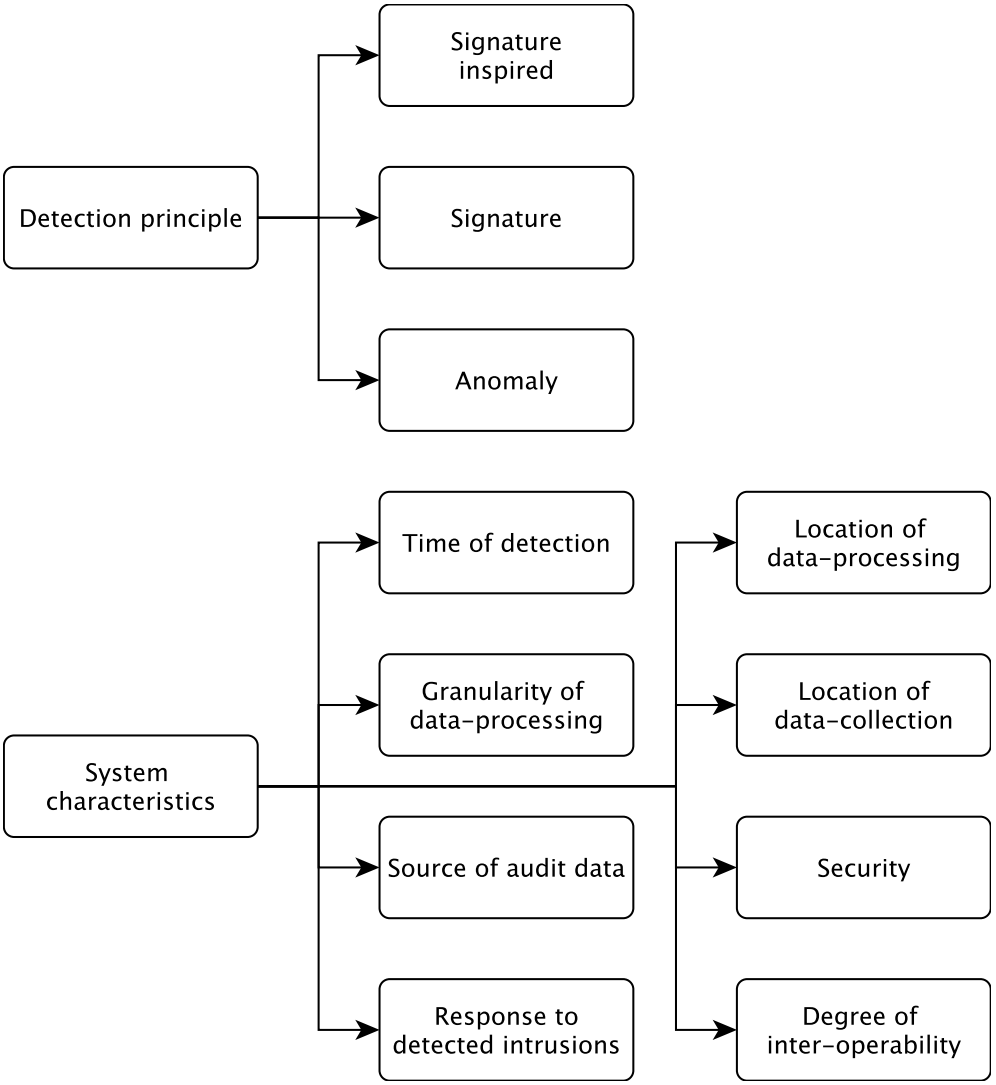


Figure 3.2: Taxonomy of intrusion-detection systems, Axelsson [2]

3.3 From Knowledge to Behavior-Based Detection

In practice, typical intrusion detection systems share aspects on both of knowledge and behavior-based detection methods. In the following, we describe the primary approaches taken when implementing IDS, starting from purely knowledge-based and concluding with mostly behavior-based. We do not follow any particular methodology but discuss common underlying techniques and assess their applicability to detecting vulnerabilities in software.

3.3.1 Knowledge based systems

Rule-based systems [40–42] are the most popular techniques that are used to implement knowledge-based intrusion detection systems. A system contains a set of predefined rules for known attacks or other behaviors that are undesirable. System operation, such as execution traces or network traffic, is monitored for any activity matching pre-configured rules. For example, it is a common practice to prevent root users from accessing their account using the FTP protocol. An attempt to access an account in such a way may indicate system misconfiguration or an attacker's reconnaissance. Intrusion detection systems, such as Snort [41], can be configured to detect such suspicious activity by prescribing a rule, such as the following.

```
alert tcp any any -> any any 21 (content:"user root";)
```

This rule configures Snort to generate an alert if it detects a packet destined to port 21 (FTP) containing the string "`user root`". Even though this is a very simple scenario, this rule is easy to bypass. For example, commands in the FTP protocol are case-insensitive and both spaces and tabs are allowed as delimiters. Strings like "`USER␣ROOT`" or "`user⟨TAB⟩root`", representing the same suspicious activity, will not be captured. To properly match such string, a more advanced rule should be defined:

```
alert tcp any any -> any 21 (flow:to_server,established;\
content:"root"; pcre:"/user\s+root/i";)
```

This simple example shows that even for a simple misuse scenario, defining a comprehensive rule is nontrivial. The user of such intrusion detection system needs to know that the particular activity, such as accessing an FTP server as root, is not intended and should be monitored. This example shows that identifying and properly specifying rules faces the same kind of challenges as avoiding security vulnerabilities, as discussed in Chapter 2. The system administrator does not have to specify all rules themselves. The standard distribution of Snort includes almost 3500 rules created by the community, with over 800 enabled by default. The rule set is continuously revised and the paid rule subscription is updated on a nearly daily basis [43].

Another popular knowledge-based detection technique is using attack *signatures* [41,42]. Rather than using general rules to define the misbehavior, the signature-based system looks for a specific pattern indicating a particular attack, known

as a signature. A signature may describe a sequence of actions that the system performs or data that it receives during the attack. Often, intrusion detection systems provide both rule and signature based matching [41,42]. Rule or signature based IDS can be also applied to software, most commonly as malware detection tools.

The key advantage of using knowledge-based systems is their low false-positive rate and high performance [42, 44]. This is because they typically perform very simple verification using precise criteria defining known attacks. Their main challenge is the inability to detect previously unknown or unexpected misuse scenarios. As new attack techniques or mutations of existing ones are discovered, the set of signatures becomes obsolete and the protection weakens. New attacks need to be continuously recorded and rule sets kept up to date. Also, the modern advanced malware uses polymorphism to modify its behavior making the detection more difficult [45].

The anomaly detection is based on the assumption that a deviation from system's normal behavior may represent an intrusion. The key benefit of this approach is an ability to detect unknown or unexpected attacks. Contrary to knowledge-based systems, the a priori expert knowledge, such as definitions of attacks, does not need to be specified. In practice, however, behavior-based mechanisms include a knowledge-based component which may include specification of what characteristic to monitor, what kind of deviation should be classified as an anomaly, and so forth.

3.3.2 Statistical models

Statistical models [46–49] are among the simplest and most popular ways to define normal behavior. The key operation of such systems is to probe various system characteristics periodically and compare the results with baseline metrics. For example, it may be expected that a certain number of unsuccessful login attempts is normal, so an unusually high number may indicate a brute-force attack. A system may be configured with a value that sets a threshold for the number of failed login attempts that represent normal behavior. If the threshold is reached, the system may generate an alert. A certain amount of knowledge is required, for example: the fact that a particular characteristic (number of unsuccessful logins) is interesting, where the data can be obtained from, and the threshold value. The difficulty in implementing this approach grows with the size of the system and a

number of properties of the system behavior that should be taken into account.

A more useful approach is to automatically infer normal system characteristics from its past operation. For example, the number of unsuccessful logins may be established by analyzing system logs. Normal operation of the system may be modeled in a more fine-grained way. For example, the system may learn the typical working hours and the location from which its users log in, based on previously recorded activity. It may also learn the normal patterns separately for individuals and groups of users [46]. Although it is possible to establish the thresholds automatically, the significance of the characteristics monitored by the system needs to be understood first. In addition, the statistic-based models can capture only relatively simple intrusion scenarios defined as quantity or frequency of discrete actions. Another problem to consider is natural fluctuations in quantities of events due to workload, time of day, the day of week or month [49]. Regardless how well the system can establish the baseline values, it still requires the expert knowledge in order to specify what properties of the system behavior, such as the number of failed login attempts, should be monitored.

3.3.3 Expert systems

Anomaly detection expert systems [13, 50–52] use rules that are generated automatically based on recorded past behavior. For example, Wisdom and Sense [13] uses a system audit log that contains information about individual program executions, such as the name of the program, the user and their role, time of execution, CPU time, etc. The system analyzes the audit log and builds a rule forest in order to identify repeating patterns of correlations between various attributes. Such patterns may vary from generic, such as "the valid terminals are T1, ..., Tn", to very specific: "on Tuesday between 6:00 am and 7:00 am when the user has a system operator privileges and is using terminal T3, only commands that cause very little direct disk activity are used". A step further is to generate rules based on correlation of several subsequent events [50]. The key advantage of this system, compared with statistical models, is that it automatically detects the characteristics of the system that are important and should be included in the model. These rules may reflect intended security policies, as well as characteristics that are not expected or considered important, but useful to consider when asserting system's correct operation. Such rules result from the system configuration, usage patterns or the environment in which it operates. The system performance may be improved by tuning the rules based on operator feedback,

for example when a false positive is detected [52].

Taking this approach allows building a model of system behavior with a limited amount of expert knowledge. However, it requires detailed, expert information about log structure and data types of its elements, criteria to map continuous values, such as time or CPU usage, into discrete clusters. More recently, similar techniques were applied to automatically generate security policies for Java Security Manager [53] or Android [54] that represent permitted access that applications require during the normal operation.

3.3.4 Operation sequence models

A different approach for building behavior models is to look for correlations between consecutive operations. It is based on the assumption that normal system activity can be described as a collection of repeating sequences of operations, and a system under an attack will produce different, previously unknown, sequences. For example, in an order-processing purchase system, it may be normal to first create an order and then issue an invoice. Deviation from that pattern (for example, issuing an invoice without prior order) may be detected as an anomaly.

The seminal application of this technique [14, 17]focuses on sequences of UNIX system calls resulting from program execution. In this model, traces of application's system calls are decomposed into a set of short sub-sequences, known as *n-grams*, capturing short-range correlations between subsequent events. Execution of the application is validated against the set of sequences and, if it does not match it is considered an anomaly. A number of small size experiments, covering a few typical UNIX services with selected vulnerabilities, have been performed. These experiments demonstrate that this technique can, in principle, be useful for the identification of anomalies caused by vulnerabilities. Also, other techniques that use more advanced data models such as machine learning or state machines have been proposed [55–58]. We discuss them in more detail in Section 3.4.1.

The operation sequence based anomaly detection requires less expert knowledge than expert systems, such as Wisdom and Sense [13], as the only characteristic considered is the order of events in a sequence. Some knowledge, however, is required. For instance, it has to be specified which elements of the log represent the operation and should be used to build the model. While much of the research focuses on UNIX system call names, the selection as to which elements of the trace describe the operation is not trivial in other execution environments [20].

Logs may also need to be processed before the behavioral model is built. For example, an application may start new threads to perform some parts of its logic concurrently, causing event sequences to be randomly interleaved. The detection mechanism may have to handle this by explicitly recognizing certain operations [14, 17], and separating the traces accordingly. Also, the configuration of the model, such as the length of the sub-sequence, may vary between platforms and applications and have to be established by experts or through experiments [59].

3.3.5 Process mining

Process mining [60, 61] combines machine learning and data mining with process modeling and analysis. Its purpose is to discover and monitor underlying processes by extracting knowledge from event logs. For example, the audit trails of a workflow management system or the transaction logs of an enterprise resource planning system can be used to discover models describing processes, organizations and products. Process models allow modeling system behaviors in richer structures such as Petri nets [62]. Process mining techniques have been applied to security audits [63]. The model obtained using process mining techniques is useful in detecting violations according to known security requirements, such as authorization, separation of duties or conflict of interest. However, these security requirements have to be prescribed by the experts and are not inferred from the logs. A more general approach to anomaly detection using process mining that could be applied to any level of abstraction is proposed [62], however, it is only demonstrated on rather high-level business processes [62, 64]. In general, while process mining could be potentially applied to detect anomalies in software execution, the practicalities of that approach are mostly unexplored.

3.4 Exploring Sequence-Based Anomaly Detection in Software

The unexpected behavior of an application resulting from an attack caused by a programming error as described in Chapter 2, manifested itself through additional method calls in the execution trace. Therefore, we argue that anomaly detection techniques centered around a temporal ordering of operations such as [3, 14, 20, 25] are suitable for identifying such attacks systematically. In this section, we explore this group of anomaly detection systems.

The operation of the system involves learning the model, such as a set of n-grams, from traces of past executions of the application and then examining new traces (online or offline) against the model. Typically, the learning phase involves processing traces of system activity captured over long periods. As more data is processed, the model becomes increasingly more precise. For example, Figure 3.3 depicts the number of unique n-grams of length used in the experiment in relation to the number of all such n-grams processed, as presented in [3]. The detection

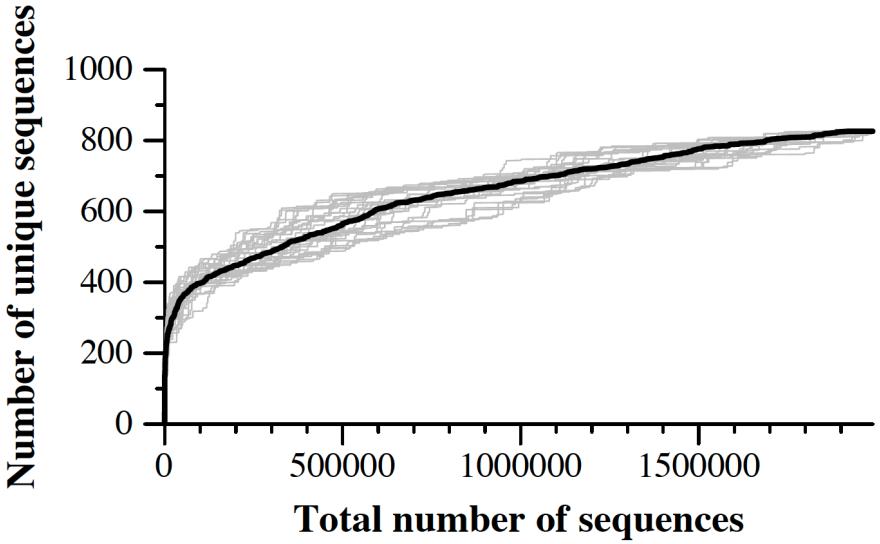


Figure 3.3: Database growth curve for learning process, reprinted from [3]

phase analyzes the execution trace of an application and identifies any deviations from the behavior captured in the model. Typically [3, 14], the anomaly is only reported if a previously-set deviations threshold value is exceeded.

3.4.1 Data models

Sequences of data are used in a number of domains, such as bioinformatics, weather prediction and intrusion detection. This led to the development of several techniques for modeling sequence data and identifying anomalies [65]. In the following sections, we present key sequence-based data models and anomaly detection techniques that have been proposed for software execution [3, 14, 25].

3.4.1.1 Look-ahead pairs

In the original implementation of sequence-based anomaly detection for UNIX systems [14], the model was built using look-ahead pairs. A pair with the look-

ahead value of k is a mapping between a system call name, and the following system calls at positions $1, 2, .., k$. The pairs are constructed by analyzing a sequence using a sliding window of value $k+1$. For example, consider the sequence of system calls ⟨`open, read, mmap, mmap, open, getrlimit, mmap, close`⟩, as presented in [14]. Look-ahead values for the first position of the window are presented in a table below.

call	position 1	position 2	positon 3
`open`	`read`	`mmap`	`mmap`
`read`	`mmap`	`mmap`	
`mmap`	`mmap`		

If the same call name, such as `open` repeats a number of times the look-ahead values are added at the respective positions. The table below represents look-ahead values for the entire sequence.

call	position 1	position 2	positon 3
`open`	`read, gretrlimit`	`mmap`	`mmap`
`read`	`mmap`	`mmap`	`open`
`mmap`	`mmap, open, close`	`getrlimit`	`mmap`
`getrlimit`	`mmap`	`close`	
`close`			

Anomaly detection is performed by validating a sequence against the look-ahead values. For example, a sequence ⟨`open, read, write, close`⟩ is anomalous as it does not match value at position 2.

The length of the sequence is an important parameter and has to be carefully set. When too short, the application activity may not be captured precisely enough to detect anomalies [59]. Too long sequences may make the model overly strict and result in many false positives. At one extreme ($k = 1$) the model would contain just a set of acceptable system calls, at another (k equal to trace length) in one that precisely enforces learning traces.

3.4.1.2 Subsequences

An alternative way to model short-range correlations between subsequent events is to record sequences of given length, also known as n-grams [17]. An n-gram of a sequence is a subsequence of length n. For example, the following depicts the set of n-grams of length 3 for the sequence discussed in previous section.

{⟨`open, read, mmap`⟩, ⟨`read, mmap, mmap`⟩,
⟨`mmap, mmap, open]`, ⟨`mmap, open, getrlimit`⟩,
⟨`open, getrlimit, mmap`⟩, ⟨`getrlimit, mmap, close`⟩}

Detecting anomalies using n-grams is done by examining whether each n-gram from a sequence is included in the previously recorded set of n-grams. To achieve greater precision, n-grams may include markers for beginning and end of a sequence. For example, the set may additionally include ⟨`START, open, read`⟩ and ⟨`mmap, close, END`⟩, with `START` and `END` acting as the markers.

Both sequences and look-ahead pairs are very similar data structures and their efficacy to model the normal behavior of software execution have been compared. While sequences have been first considered more suitable [17], the in-depth analysis [66] revealed that look-ahead pairs may be in fact superior.

3.4.1.3 Groups of subsequences

The data models discussed in previous two sections map application execution into a single set of short-range correlations. In practice, however, systems may operate by executing distinct transactions. For example, a network server typically operates by performing a sequence of operations in response to a client's request. Execution traces of requests of the same type may be similar to each other and different from traces of other types of requests. Rather than capturing all n-grams into a single set, they could be partitioned according to transaction types in order to provide a more precise model [20, 25]. Detecting anomalies in such models requires comparing the sequence against all sets separately, though many non-matching sets may be quickly disregarded to reduce the search scope. Even if the system does not explicitly exhibit transaction-like behavior, partitioning long execution traces may be done using several criteria such as a group of routines, procedures, functions, threads or forked processes or activity intervals [67].

3.4.1.4 Frequency-based models

Techniques discussed so far are focused only on enumerating short sequences with no consideration to how often a sequence appears in the execution trace. In practice, some sequences may be much more common than others and frequency of their appearance may be used to build a more precise model. The relative frequency of n-gram appearance in the normal traces can be recorded [68] but is unsuitable for on-line processing, as the entire sequence is required for comparison. Other models [69] are based on a priori knowledge about frequencies of certain executions during an intrusion in relation to their normal frequency. The requirement of previous knowledge makes them unsuitable to detecting previously unknown intrusions, such as exploitation of unknown software vulnerabilities.

3.4.1.5 Richer models

All techniques presented so far are centered around matching the equality of a certain short sequence of program execution against short sequences that define normal behavior. Rather than capturing the explicit sequences, the key characteristics of temporal order in software execution traces could be learned. For example, the relations between events in system call n-grams [14, 17] could be captured as a set of rules [58], as follows.

```
normal:- p2=open,p7=read
normal:- p2=open,p7=open
...
abnormal:- true
```

The first rule prescribes that if the system call at position 2 is `open` and the call position 7 is `read` than the sequence is normal. The second, that if the system calls at positions 2 and 7 are `open` than the sequence is normal. Finally, if none of the rules are matched, the sequence is considered abnormal. Other techniques include the discovery of sequences of a variable length [55, 56], finite state machines [56] and neural networks [57].

3.4.2 Attacks

The sequence-based techniques introduce some level of approximation of system behavior that can be used to evade the anomaly detection. Some studies [59, 70]

suggest that the successful results in detecting attacks using those techniques [3, 14] were possible because of a fortunate selection of model properties such as sliding window size. They point out that some attacks used in the experiments [3, 14] would not have been detected if the size was lower. Also, they demonstrate that exploits for other vulnerabilities, and slightly modified versions of the exploits used in the experiment, can not be detected with the sliding window size selected in the experiment.

A more deliberate way of subverting the system call anomaly detection is the mimicry attack [71]. Mimicry attacks are centered around purposely modifying the attack in order to be indistinguishable from the normal behavior [72] and thus evading the anomaly detection. The attack for sequence-based models is performed by constructing a malicious sequence of actions that the attacker intends to perform, entirely from the allowed short-range sequences. This could be achieved by adding so-called "nullified" system calls that have no important side effect, such as opening a missing file, between the calls that are of interest to an attacker.

This mimicry technique has a number of limitations. First, performing the attack requires the knowledge about the model of normal behavior for the particular target system [71]. While one could argue that the system should be secure even if the attacker has a full knowledge of its operation, in practice it may be hard or impossible to obtain that information. The operation may vary substantially between application instances and can depend on configuration and the environment [15]. Also, the operations that the attacker intends to inject, must exist in the model in some form. In addition, such attacks are only possible if the application has a vulnerability that allows injection of a precise sequence of operations, such as buffer overflow or cross-site scripting. That, in turn, means that it may be impossible to make the entire sequence free from anomalies, as the "preamble" [73] sequence that enables the injection is not under the control of the attacker.

The feasibility of the attack may also depend on the precision of the model. The longer, and less common, the n-grams are, the harder it may be to craft the malicious sequence. Overly long attack sequences may not be technically suitable for injection, for example, due to limitations of the input that attacker has available to use. We are not aware of any studies on mimicry attacks for the models that include additional context attributes such as caller class [20, 25] or data flow elements [18, 19, 74], though we speculate that such attacks may be very

hard to implement in practice. In addition, implementations that use higher-level operations, such as Java method calls [25] instead of low-level system calls may significantly limit an attacker's ability to inject any sequences that would be useful for performing an attack as they are constrained to the high-level, application-specific operations. Finally, richer models may further limit the possibility of successful mimicry attacks. For example, partitioning the application behavior based on transactions [25] limits the set of available n-grams.

3.5 Conclusion

The behavior-based intrusion detection techniques often require some degree of expert knowledge to guide their operation. We argue that operation sequence based techniques provide the right balance between the behavior and knowledge-based approaches and have been applied to detecting attacks against software components for the last 20 years. However, the existing research in this area tends to be focused on simple software components and a narrow view of application activity through UNIX system calls.

The experimental evidence for the efficacy of these techniques is limited. Typically, their processing speed, model size, accuracy and false positive rate is compared to the previously published work, in particular to [14, 17]. The comparison is often based on the same datasets, that is, traces of UNIX programs such as `sendmail`, `lpr` and `inetd`, but the assessment criteria differ and the results are difficult to compare. The only attempt to evaluate multiple techniques more systematically [3] is also focused on the same few applications and corresponding vulnerabilities. Taking the narrow view (UNIX system calls), and the small scale of the experiments performed, it is difficult to make any definite conclusions about the efficacy of these techniques in identifying software execution anomalies in general, outside the scope of UNIX system calls of a few selected programs. Applicability of the anomaly detection techniques to large-scale, layered systems running on modern software platforms remains mostly unexplored.

Chapter 4

Practical Challenges of Anomaly Detection in Contemporary Software

4.1 Introduction

As discussed in Chapter 2, an application system program has a security gap when a developer's misunderstanding means that an attacker can exploit the difference between its expected behavior versus its actual behavior and, for which security controls do not exist. This chapter considers how the security gap might be reduced by checking the runtime behavior of the component against a model of its expected behavior. Runtime verification [75] is the process of observing system execution and validating that specified properties are upheld, or that the execution is consistent with a testing oracle. For example, a predicate stating that each `acquire` has a matching `release` in a (re-entrant) Lock class [76]. However, runtime verification alone cannot address the security gap: a typical developer faces similar challenges when encoding all properties of the expected behavior for the runtime verification, as when implementing the system with no security vulnerabilities.

As discussed in Chapter 3, the use of anomaly detection in software execution has tended to focus on relatively small-scale homogeneous applications [14]. However, the example in Chapter 2 illustrates that even very simple contemporary

applications are large systems of interconnected components. The applicability of these anomaly detection techniques has not been explored for modern software development and delivery. This chapter considers the key challenges encountered in applying anomaly detection techniques to contemporary application software. Without loss of generality, the discussion is focused on a common enterprise scenario of a web application built on a high-level software platform (Java) and an MVC framework (Apache Struts), such as the microblog web application presented in Chapter 2.

4.2 Abstraction and Scope

The activity of an application is observed as a sequence of events, such as a system log. These observations can be made at different levels of abstraction, as depicted in Figure 4.1. For example, the microblog application activity could be

Abstraction level	**Sample events**
User actions	`User edit post` `User post submission`

HTTP requests	`GET /ui/components/editor`
	`GET /ui/style/editor.css`
	`GET /ui/static/emoji.jpg`
	`POST /api/post?text=Hello`

Method calls	`ParamsInt: PostAction.setText("Hello")`
	`Dispatcher: PostAction.execute()`
	`PostAction: PostAction.getUser()`
	`PostAction: HashMap.get("user")`

Virtual machine	`> BCU checkForExistingClass:`
	`  classnamePtr=00007FFAE019A286,`
	`  classLoader=00007FFAE004AC88`
	`> BCU createROMClassFromClassFile:`
	`  loadData=00007FFAE87F4EB0`
	`> BCU build ROMClass start`
	`  [classname=java/util/Collection]`

System calls	`open("/usr/lib/jvm/"..., O_RDONLY) = 3`
	`read(3, "\177ELF\2\1\1\0\0." ..., 832) = 832`
	`fstat(3, {st_mode=S_IFREG|0755, ...}) = 0`

Table 4.1: Execution traces at different levels of abstraction

observed as a series of high-level user actions, such as editing, posting or viewing a message. At a lower level of abstraction, the operations can be observed as HTTP calls to the application server. Further lower levels of abstraction describe the activity of the application code and its libraries, Java virtual machine, operating system, and so forth. The objective is to use the observed activity/log of the system to build a behavior reference model for use in anomaly detection. The challenge is determining a level of abstraction that enables anomalous execution paths to identify security vulnerabilities. While, without a loss of generality, we focus on different abstraction levels in context of software systems monitoring, in other event sources, such as SIEM, different criteria could be used for selecting interesting events.

Security vulnerabilities occur at different levels of abstraction. Building reference models from observations of low-level interactions, such as operating system calls [14, 77, 78] or JavaScript [16] calls has been shown to be quite effective in detecting specific type of exploits, such as buffer overflow vulnerabilities or cross-site scripting. However, the anomalous execution paths in the microblog example may not be detected at the level of abstraction comparable with [14, 77, 78]. At that level, application activity can be only viewed through network connections, file access, database operations, memory and process management, and so forth. A high-level anomaly, such as modifying a server-side session though an HTTP request parameter, described in Chapter 2, may not be reflected in that view. Similarly, making observations based on HTTP requests may not be sufficient. Web application firewalls that look for anomalies in the request structure, such as new or absent attributes [18], may detect the session manipulation attack, as it involves adding a request parameter that is not normally used by the application. However, the second attack presented in Chapter 2, that is accessing local files through website preview, can not be detected though such a simple analysis. A malicious input (the `file` URL) is a part of free-text data of the message content and, as such, is not normally examined by web firewalls. A rule-based web application firewall could be configured to explicitly detect types of URLs that are not allowed, but this would require the awareness of the particular problem of URL handling by the application, which is not the case we are considering here.

We propose the analysis algorithm [20], described in Chapter 6, that discovers a level of event abstraction that provides good anomaly-detection accuracy in transaction-like behaviors; it has proven quite effective in discriminating actions (methods) that remain static from the parameters that can change within a transaction.

In our experiments, discussed in Chapter 8, we found that observing (Java) application behavior in terms of its method calls and permission checks is most effective in distinguishing the unexpected execution paths that lead to software vulnerabilities. In practice, however, it is unrealistic to monitor every underlying method call invoked as a consequence of application operation. For example, a few lines of the microblog application Java code in Figure 2.2, Chapter 2, can generate several thousands of method calls. For comparison, a simple "Hello World" Java program execution involves over 100,000 method calls, with almost 3000[1] just to print the message. Therefore, the scope of observation should be restricted to methods that belong to the specific component(s) of interest.

4.3 Generating Baseline Activity

A baseline of observed activity must be obtained in order to build the behavior reference model for the software component. In order to be applicable to an enterprise environment, the technique of establishing baseline activity has to be systematic, repeatable and aligned with application development/deployment life cycle. Most importantly, the baseline generation must provide sufficient coverage for the normal/expected behavior of the application.

A common position is that a satisfactory baseline can be obtained as a recording of "normal" activity obtained through monitoring application operation in its target environment [14–16]. For example, the reference profile for `lpr` in [3] was built based on traces of 15 months of operation in one environment and two weeks in another. Such extensive data collection time may provide an excellent coverage of the normal activity, but has practical limitations for enterprise software and environments. As the software industry adopts faster and more automated delivery approaches, and software releases are as frequent as on a daily basis, building a (normal) baseline based on observation of an application in the production environment is impractical. It would require the application to be active for a significant length of time without anomaly detection. It also may be impossible to obtain a sufficient baseline before it becomes obsolete due to changes to the application or updates to the underlying components.

We argue that it is more appropriate to obtain the baseline and corresponding behavioral model during the software testing phase. In [23] we suggest that unit

[1] IBM J9 JVM, 1.7

tests might be used to generate the necessary system logs. Such baseline is useful if the required scope and level of abstraction is consistent with the unit tests. Notwithstanding coverage, unit tests are typically constructed for a specific component, in isolation, and may only be suitable for relatively simple components or utility libraries. An alternative approach is using functional and integration tests to generate the baseline. Enterprise software is often subjected to extensive testing, usually highly automated and with controlled coverage. Also, in popular continuous delivery models, automated testing is one of the critical elements of software development, with the amount of code implemented for testing often exceeding that of an actual application. Traces resulting from such software tests may provide a reliable and systematic baseline model. However, as with unit tests, the functional test coverage is usually built only to a certain satisfactory level but it is rarely complete. Additional coverage may be obtained using fuzz testing or application scanners that automatically exercise an application in order to perform non-functional types of testing such as security or performance. The application scanners often use functional test execution as its starting point and then extend the coverage by exploring the application further in an automated fashion. In summary, the coverage is depending of the chosen test-regime chosen and in practice different test-regimes can be used to achieve different level of coverage based on needs and cost.

In our experiments (Chapter 8) we found that using an application scanner allows the system to be explored in a comprehensive, unbiased and repeatable manner and was the most effective for our task [25, 54]. The scanner explores the application extensively in order to discover its structure, parameters, cookies and so forth, and also engages in a series of tests in an attempt to discover security vulnerabilities. While the scanner interacts via high-level requests to the application's external interfaces, a system log/trace of the consequent low-level operations on the underlying system is generated. The scanner alone may not be sufficient in larger, real-world configurations. Evaluating the efficacy of automated scanning and combining multiple development-time techniques to acquire baseline behavior is a subject for future research.

4.4 Behavioral Reference Model

4.4.1 Applicability to modern software platforms

Nearly all of the techniques presented in Chapter 3 are discussed in the context of UNIX system calls, following the first, seminal work in this area [14]. The view of application behavior through system calls is attractive as it captures a fairly narrow, but important part of application activity, often related to security, such as performing input/output operations and executing processes. However, this view may not be useful for modern applications that are based on layered software platforms. While the focus may be on the behavior of the web application itself, the trace of system calls may be mostly reflecting the activity of underlying components, such as execution environment, application server and web application framework. The high-level behavior of the application, which may be important for analysis may not be represented at all in such a low-level trace. Even if some portion of that behavior is captured using system calls, it may be insignificant in comparison with the activity of the platform that supports it. It may be required to capture an application's activity at a different level of abstraction and restrict it to the application-specific operations. Software platforms may provide other integration points that resemble system calls but operate at a level more suitable for the analysis of application behavior. For example, the Security Manager that is invoked for every security-related operation could be a natural integration point for Java applications [53]. Still, a large portion of the behavior, such as the one described in the previous sections, can not be captured. An alternative approach may be to use a more fine-grained trace of Java method calls.

There is a substantial difference between system calls and high-level language method calls. The number of system calls is rather limited, with around 300 for typical Linux and BSD systems, and the same set of common calls is used by most of the applications. In comparison, the application may use thousands of methods and many of them may be specific to the application itself or even certain part of the code. This difference may impact the efficacy of the anomaly detection and the configuration parameters. For example, a typical length of the n-grams used in the experiments focused on system calls [15] was in the range of 10. It may be possible that much shorter sequences may be suitable to model the higher-level operation of the system. This is due to the fact that, intuitively, the shorter sequences from a large set of application-specific calls may have the same entropy as the longer sequences from a smaller set of common general-purpose

system calls.

4.4.2 Model expressiveness

Data models used to represent the software behavior capture various correlations to record temporal ordering of operations. The expressiveness of the data model, that is how well it represents the actual activity of the software, impacts the efficacy of the corresponding techniques in identifying different types of anomalies in program execution. The simplest of the models, focused on short-range correlations of actions, can be quite effective in detecting previously unknown sequences of actions. These models are particularly efficient in detecting foreign code execution due to attacks, such as the buffer overflow [14] or cross-site scripting [16], involving execution of operations that are not covered in the baseline. The focus is on the behavior that is new, such as an additional sequence of system calls. However, software vulnerabilities do not necessarily have to manifest themselves through additional elements in the system traces. For example, a typical sequence for a business transaction may contain a segment representing the execution of a security mechanism. An attack may represent itself by a sequence in which this segment is missing. Such a sequence may be accepted as it does not introduce anything unknown. While some models, such as finite state machines [56], could in principle detect such anomalies, theses type of attacks are mostly unexplored.

In many applications, activity may be contained in finite and separate units of work, where a sequence of operations that has a clear beginning and end, and follows some repeatable pattern. Application behavior may contain multiple such transaction-like work units. For example, operation of the microblog application at a method call level, can be modeled as a collection of different kinds of transaction behaviors, such as posting a message or reading a message. The anomaly detection techniques discussed in Chapter 3, with the exception of those discussed in Section 3.4.1.3, aggregate the entire behavior of the system. While the system may run a number of distinct processes, only a single blended behavior of the entire system is captured. For example, an application providing 50 REST APIs could be more precisely represented as 50 separate, smaller models than a single model that includes the combined behavior of all of them. Blending transactions may reduce anomaly detection accuracy of techniques that use simple data structures, such as sets of n-grams, because any input sequence is matched to all known n-grams, regardless if they ever appear in this arrangement.

The majority of operation sequence anomaly detection techniques focus solely on the control flow and do not consider any additional context, such as call arguments or relationship between operations, that may be available in the trace. This limits the ability of the system to identify attacks that do not modify the control flow of the application [79]. Such attacks often result from the vulnerabilities, such as injection or insecure direct object reference, that are among the most common in contemporary software [6]. For example, consider the following trace fragment, produced by a variant of the Java-based microblog system presented in Chapter 2. The trace has a format of `CallerClass: Class.method(arguments)`.

```
SessionHandler: PostAction.setUser("frank")
PostAction: Message.setAuthor("frank")
```

The first method call represents the web framework processing the session and providing a user identifier from the server-side session to the application. The second method call represents the application setting the author of a message. If only the class and method are considered as an operation, the sequence used by the model takes form of ⟨`PostAction.setUser, Message.setAuthor`⟩.

However, the fact that the value of the argument (`frank`) is the same in both method calls may be an important part of the application's normal behavior. For example, an application may contain a programming error that derives the second attribute from a request parameter, rather than current user's object. As the request parameter could be modified by the user, they could exploit the vulnerability and post a message on behalf of someone else. In this case, the trace would be as follows.

```
SessionHandler: PostAction.setUser("frank")
PostAction: Message.setAuthor("alice")
```

Even if only the control flow of the application is considered, the decision on what elements of the traces are considered as representing that flow may have an impact on the expressiveness of the model. For instance, the fact that methods are called by specific classes (`SessionHandler` and `PostAction`) may be also an important part of application's normal behavior. A different vulnerability may involve allowing alternative inputs for the current user. For example, the application developer could mistakenly enable an HTTP cookie handler in the application framework, in addition to a session handler. A malicious user may exploit this by disregarding the session and adding an HTTP cookie named `user` and set its value to a different user identifier. The web framework would then process the cookie instead of the session, resulting in the following trace.

```
CookieHandler: PostAction.setUser("alice")
PostAction Message.setAuthor("alice")
```

In order to identify these two anomalies, the behavioral model should consider caller class as a part of the operation, and the fact that the argument of two subsequent calls, under normal circumstances, is the same. This problem is partially addressed by models that focus on data flow [18, 19] or combine data flow with control flow [74, 78]. These techniques take into account system call attributes and provide heuristics to match their value based on typical types, for example by identifying files in the same directory. Other techniques use additional context attributes, such as caller class, in its model of normal behavior [20, 25] and as considered in Chapter 8.

Correlations between sequences of operations and their target values can be used to discover repeating patterns of behavior and [20] uses this to develop an automated technique to partition a system trace into a collection of behavioral norms. In our experiments we use these *behavioral norms* [20], described in Chapter 5, to provide a reference model for anomaly detection. A collection of behavioral norms is generated, from the baseline activity log of Java events, for a given scope and level of abstraction, each corresponding to a sequence of method calls parameterized by common target attributes.

4.5 Conclusion

Using anomaly detection techniques in order to identify vulnerabilities in contemporary software is the area mostly unexplored in the existing research. Focusing on the common Java-based web application scenario from Chapter 2, we explored the key challenges of putting anomaly detection into use for runtime verification of complex, layered software, built according to the modern development principles. One of the challenges is the identification of the right level of abstraction to monitor application behavior and selection of the scope that could provide the view for identifying anomalies caused by vulnerabilities accidentally introduced by programmers. We explored the approaches for establishing the baseline application behavior and observed that automated application testing and scanning is the most practical way to acquire it in a typical software development life cycle. Finally, based on the review of anomaly detection techniques applicable to software from Chapter 3, we consider the characteristics of the anomaly detection techniques that make them suitable for our task.

Chapter 5

Behavioral Norms

5.1 Introduction

In this chapter, we present a refined formal definition of the model of behavioral norms that was first introduced in [20]. The model is specified using the Z notation [80] and has been syntax and type checked using the *f*UZZ checker [81].

5.2 Events and Traces

To illustrate the model, Figure 5.1 gives a log `appTrace` of the microblog application `PostAction` execution from Chapter 2. Note, for ease of understanding, many of the events for Java method calls in that execution have been discarded. Section 5.3 considers how the selection of events is done in a systematic way. The trace depicts a Java method call log of events $\langle e_1, \ldots, e_{20} \rangle$.

An *event* represents an observation of some interaction with a system. We focus on events drawn from system logs, for example,

```
34  T2 org.struts.Dispatcher net.micro.PostAction   setText      "Hello world"
```

denotes an event from a web application. Let *Event* denote set of all possible events.

$$[Event]$$

The different characteristics of an event can be described in terms of its *attributes*:

	id	thr	caller	class	method	argument
e_1	34	T2	com.web.HttpHandler	org.struts.Dispatcher	dispatch	[Request]
e_2	34	T2	org.struts.Dispatcher	net.micro.PostAction	setText	"Hello world"
e_3	34	T2	org.struts.Dispatcher	net.micro.PostAction	execute	
e_4	34	T2	net.micro.PostAction	net.micro.PostAction	getUser	
e_5	34	T2	net.micro.PostAction	java.lang.HashMap	get	"user"
e_6	34	T2	net.micro.PostAction	net.micro.Message	setAuthor	[USER]
e_7	34	T2	net.micro.PostAction	org.utils.TextUtils	getUrls	"Hello world"
e_8	34	T2	net.micro.PostAction	com.data.DAO	add	[MESSAGE]
e_9	34	T2	net.micro.PostAction	net.micro.PostAction	return	SUCCESS
e_{10}	34	T2	com.web.HttpHandler	com.web.HttpHandler	sendResponse	[Response]
e_{11}	39	T2	com.web.HttpHandler	org.struts.Dispatcher	dispatch	[Request]
e_{12}	39	T2	org.struts.Dispatcher	net.micro.PostAction	setText	"Hi there!"
e_{13}	39	T2	org.struts.Dispatcher	net.micro.PostAction	execute	
e_{14}	39	T2	net.micro.PostAction	net.micro.PostAction	getUser	
e_{15}	39	T2	net.micro.PostAction	java.lang.HashMap	get	"user"
e_{16}	39	T2	net.micro.PostAction	net.micro.Message	setAuthor	[USER]
e_{17}	39	T2	net.micro.PostAction	org.utils.TextUtils	getUrls	"Hi there!"
e_{18}	39	T2	net.micro.PostAction	com.data.DAO	add	[MESSAGE]
e_{19}	39	T2	net.micro.PostAction	net.micro.PostAction	return	SUCCESS
e_{20}	39	T2	com.web.HttpHandler	com.web.HttpHandler	sendResponse	[Response]

Figure 5.1: A sample trace `appTrace`

operations, parameters and other information about the event. For example, the Java method call log could be described in terms of attributes: request **id**, the **thr**ead in which call was made, the **caller**, the called **class** and **method** names as well as the first method **argument** if any. While an event may be regarded as defined in terms of a collection of attributes, we do not prescribe any particular attributes or structure on an event.

A *trace* is a sequence of events. Let *Trace* define the set of all traces.

$$Trace == \text{seq}\, Event$$

For example, Figure 5.1 depicts a trace of web application events $\langle e_1, e_2 \ldots, e_{20} \rangle$.

5.3 Scope and Filtering

A scope is used to hide events or to remove events/attributes that are not considered of interest. Let *SCOPE* be the set of all possible partial functions that map events to their abstract form.

$$SCOPE == Event \nrightarrow Event$$

For example, `microblogScope`: $Event \pfun Event$ is a partial function defined for all events with either caller or class in the microblog application package (`net.micro.*`) that maps to events including `caller` and `class` (restricted to the class name only), `method` and `argument` attributes.

The *filter* function applies a scope to a given trace.

$$\begin{array}{|l} filter : Trace \times SCOPE \fun Trace \\ \hline \forall\, trace : Trace; scope : SCOPE \bullet \\ \quad filter(trace, scope) = squash(trace \fatsemi scope) \end{array}$$

In Z, a trace of type (seq *Event*) is encoded as a function of type ($\nat \pfun Event$), and, therefore, the filter of a trace for a given scope is the sequential composition of that trace and scope, compacted to a sequence using the *squash* function [80]. For example, the trace in Figure 5.2 is a filtered version of a trace in Figure 5.1: *filter*(`appTrace, microblogScope`).

	id	**caller**	**class**	**method**	**argument**
f_1	34	Dispatcher	PostAction	setText	"Hello world"
f_2	34	Dispatcher	PostAction	execute	
f_3	34	PostAction	PostAction	getUser	
f_4	34	PostAction	HashMap	get	"user"
f_5	34	PostAction	Message	setAuthor	[USER]
f_6	34	PostAction	TextUtils	getUrls	"Hello world"
f_7	34	PostAction	DAO	add	[MESSAGE]
f_8	34	PostAction	PostAction	return	SUCCESS
f_9	39	Dispatcher	PostAction	setText	"Hi there!"
f_{10}	39	Dispatcher	PostAction	execute	
f_{11}	39	PostAction	PostAction	getUser	
f_{12}	39	PostAction	HashMap	get	"user"
f_{13}	39	PostAction	Message	setAuthor	[USER]
f_{14}	39	PostAction	TextUtils	getUrls	"Hi there!"
f_{15}	39	PostAction	DAO	add	[MESSAGE]
f_{16}	39	PostAction	PostAction	return	SUCCESS

Figure 5.2: A filtering of `appTrace` trace defined in Figure 5.1

5.4 Strands and Partitions

An *event equivalence* relation $_ \sim _ : \mathit{Event} \leftrightarrow \mathit{Event}$ defines classes of events that are considered to have common characteristics. Let $\mathit{Eq}\mathcal{E}$ define the set of all event equivalence relations.

$$\mathit{Eq}\mathcal{E} == \{_ \sim _ : \mathit{Event} \leftrightarrow \mathit{Event} \mid \forall\, a : \mathit{Event};\ b : \mathit{Event};\ c : \mathit{Event} \bullet \\ a \sim a \land a \sim b \Leftrightarrow b \sim a \land (a \sim b \land b \sim c) \Leftrightarrow a \sim c\}$$

For example, HTTP events

```
34 Dispatcher PostAction     execute
34 PostAction PostAction     getUser
```

are defined as `class`-equivalent as they both represent methods within the same Java class. Alternatively, the events

```
34 Dispatcher PostAction     setText
34 Dispatcher PostAction     execute
```

are defined as `caller`-equivalent as they refer to methods invoked from the same caller class. Event equivalence does not need to be defined based on equality of attributes. For example, all events in the trace can be considered equivalent as the caller or class belong to the same Java package. Also, an event equivalence relation may consider multiple attributes.

A *strand* is a trace of events that share a common characteristic. Given an equivalence relation $\sim$ defined over events then define $\mathit{Strand}(\sim)$ to be the set of all possible traces of equivalent events. Note, $\mathrm{ran}(g)$ denotes range of the function g.

$$\mathit{Strand} : \mathit{Eq}\mathcal{E} \longrightarrow \mathbb{P}\ \mathit{Trace}$$

$$\forall _ \sim _ : \mathit{Eq}\mathcal{E} \bullet \\ \mathit{Strand}(_ \sim _) = \{t : \mathit{Trace} \mid (\forall\, a, b : \mathrm{ran}(t) \bullet a \sim b)\}$$

For example, if $\sim_{\{\texttt{id}\}}$ denotes event equivalence based on equality of the `id` attribute then traces $\langle f_1, f_2, f_3\rangle$ and $\langle f_2, f_4, f_6\rangle$ of events from Figure 5.2 are members of $\mathit{Strand}(\sim_{\{\texttt{id}\}})$, while trace $\langle f_8, f_9\rangle$ is not a member.

Any trace can be *partitioned* into a set of strands that preserve the event ordering

in the original trace. Define function $prtn(\sim, t)$ to be the partitioning of the trace t into a set of strands according to the event equivalence relation $\sim$. For example, by event `id`-equivalence, the log is partitioned into two strands $\langle f_1, .., f_8 \rangle$ and $\langle f_9, .., f_{16} \rangle$, reflecting two separate HTTP requests with identifiers 34 and 39.

In practice different types of equivalence relations, not based on attribute equality, can be used based on need and specific use case. For example, events containing a path to a file could be considered equivalent if the path points to the same directory.

For simplicity of presentation, subsequent examples are scoped to only four attributes, the `caller`; the `class` that has been called; the `method` name and, the first method `argument`, if any, in a `Caller: class.method(argument)` format. Using this format, the trace from Figure 5.2 is depicted as the set of two strands in Figure 5.3.

```
{⟨Dispatcher: PostAction.setText("Hello world"),
  Dispatcher: PostAction.execute,
  PostAction: PostAction.getUser,
  PostAction: HashMap.get("user"),
  PostAction: Message.setAuthor([USER]),
  PostAction: TextUtils.getUrls("Hello world"),
  PostAction: DAO.add([MESSAGE]),
  PostAction: PostAction.return(SUCCESS)⟩,
 ⟨Dispatcher: PostAction.setText("Hi there!"),
  Dispatcher: PostAction.execute,
  PostAction: PostAction.getUser,
  PostAction: HashMap.get("user"),
  PostAction: Message.setAuthor([USER]),
  PostAction: TextUtils.getUrls("Hi there!"),
  PostAction: DAO.add([MESSAGE]),
  PostAction: PostAction.return(SUCCESS)⟩}
```

Figure 5.3: Set of strands from the example in Figure 5.2

5.5 Norms

A behavioral *norm* is a set of traces that is considered to define a comparable behavioral pattern. For example, traces $\langle f_1, f_2 \rangle$ and $\langle f_9, f_{10} \rangle$ from the log in Figure 5.2 represent comparable `PostAction.setText/PostAction.execute` Struts action dispatcher behavior and are (behaviorally) different to $\langle f_2, f_3 \rangle$.

A trace equivalence relation defines classes of traces having comparable behavior. Let $Eq\mathcal{T}$ define the set of all trace equivalence relations.

$$Eq\mathcal{T} == \{_ \approx _ : Trace \leftrightarrow Trace \mid \forall\, a : Trace; b : Trace; c : Trace \bullet \\ a \approx a \land a \approx b \Leftrightarrow b \approx a \land (a \approx b \land b \approx c) \Leftrightarrow a \approx c\}$$

Given a trace equivalence relation $\approx$ then $Norm(\approx)$ defines the set of all possible norms based on $\approx$.

$$Norm : Eq\mathcal{T} \longrightarrow \mathbb{P}(\mathbb{P}\ Trace)$$

$$\forall _ \approx _ : Eq\mathcal{T} \bullet \\ Norm(_ \approx _) = \{c : \mathbb{P}\ Trace \mid \forall\, t_1, t_2 : c \bullet t_1 \approx t_2\}$$

For example, the set of traces $\{\langle f_1, f_2 \rangle, \langle f_8, f_9 \rangle\}$ could represent an action dispatcher norm drawn from Figure 5.2, characterizing a fragment of request processing sequence.

A set of traces is partitioned by trace equivalence relation into a set of norms. Define $prtn(\approx, T)$ to be the partitioning of the set of traces T into a set of norms based on the trace equivalence relation $\approx$. Norms are defined in terms of the application of a trace equivalence relation to a strand partition of a log. Formally, given a log l, an event equivalence relation $\sim$ and a strand equivalence relation $\approx$ then the set of norms is defined as $prtn(\approx, prtn(\sim, l))$.

The two strands depicted in Figure 5.2 give a view that reflects an `id` attribute-centric view of the method call log. Within each strand, there is a common pattern of behavior, specifically, a sequence of methods that are invoked in order to post a message. This repeating pattern can be characterized in terms of the norm based on the trace equivalence relation defined as equality of traces with respect to `{caller, class, method}` attributes. In this case, the strands in Figure 5.3 are equivalent and the set of strands in that figure also depicts a norm.

Intuitively, the above represents a transaction-style behavior pattern in the events of the Java method log. The norm is a collection of strands that have an equivalent behavior pattern, each carried out on an identified target (in this case, `id`).

The set of norms for a particular log could be interpreted as the *profile* of system behavior captured by the log. Note that due to very short log we used to illustrate the model, the profile includes only a single norm. In practice, software execution logs are longer and result in a larger number of norms. Chapter 7 provides examples for further norms for the microblog application. In the next chapter, we demonstrate how different event and trace equivalences result in different profiles, reflecting different kinds of patterns of behavior within the system [20].

5.6 Conclusion

Behavioral norms provide a general-purpose framework for inferring transaction-like patterns of behavior from system logs. The model is based on event equivalence relation that partitions the log into distinct traces, each related to a particular transaction, called strands, and trace equivalence relation that group them into norms. In addition, events and traces can be filtered to allow focus on specific events and attributes, removing parts of the log that do not have to be considered. In following chapters we demonstrate how the model can be interpreted and applied in practice.

Chapter 6

Exploring Behavioral Norms

6.1 Introduction

The behavioral norms, introduced in Chapter 5, provide a general purpose modeling framework for transaction-style behaviors. The introduction of the model focused on low levels of abstraction such as method calls. In this chapter, we explore the model at higher levels of abstraction. We first present how different selections of model equivalence relations can lead to identifying different views into system behavior when observed through an HTTP log. Further, we describe and evaluate a technique to automatically identify model parameters for such traces. Finally, we discuss potential uses of behavioral norms to model a behavior of multiple collaborating systems.

6.2 Norms in HTTP Logs

Figure 6.1 depicts an HTTP log of events $\langle h_1, \ldots, h_{12} \rangle$ generated, for example, by a web-based order processing system. Each line defines an HTTP request event described using the Common Log Format [82] with attributes: remote `host`, `user`, [`time`], "`request`" line, response `status` and length in `bytes`.

Suppose that the `request` portion of an HTTP request event is defined in terms of the attributes: HTTP `method`, `action` (the first part of the URI) and

	host	user	time	request	status	bytes
h_1	10.20.3.11	frank	[05/Nov/2012:09:11:26]	"PUT /order/4c4712 HTTP/1.1"	200	1724
h_2	10.43.9.1	alice	[05/Nov/2012:13:18:46]	"PUT /order/1d261e HTTP/1.1"	200	4354
h_3	10.1.12.1	lucy	[05/Nov/2012:16:30:16]	"GET /order/4c4712 HTTP/1.1"	200	6356
h_4	10.1.12.1	lucy	[05/Nov/2012:16:32:32]	"PUT /invoice/4c4712 HTTP/1.1"	200	2326
h_5	10.1.12.1	lucy	[05/Nov/2012:17:46:06]	"GET /order/1d261e HTTP/1.1"	200	8320
h_6	10.20.3.11	frank	[05/Nov/2012:17:47:33]	"GET /invoice/4c4712 HTTP/1.1"	200	2925
h_7	10.1.12.1	lucy	[05/Nov/2012:17:48:35]	"PUT /invoice/1d261e HTTP/1.1"	200	221
h_8	10.20.3.11	frank	[06/Nov/2012:09:10:07]	"PUT /order/61ec0c HTTP/1.1"	200	3327
h_9	10.76.13.8	alice	[06/Nov/2012:09:58:48]	"GET /invoice/1d261e HTTP/1.1"	200	6366
h_{10}	10.1.12.2	lucy	[06/Nov/2012:14:34:31]	"GET /order/61ec0c HTTP/1.1"	200	2727
h_{11}	10.1.12.2	lucy	[06/Nov/2012:14:47:20]	"PUT /invoice/61ec0c HTTP/1.1"	200	9326
h_{12}	10.20.3.11	frank	[06/Nov/2012:16:01:45]	"GET /invoice/61ec0c HTTP/1.1"	200	332

Figure 6.1: `httpLog` trace of HTTP requests $\langle h_1, \ldots, h_{12} \rangle$

`item` (the second). This way, the request line could be viewed as `"method /action/item HTTP/1.1"`. The names of the attributes correspond with the role they have in the event. The names of attributes are introduced to improve the readability of the following discussion. We do not assume that it is required that their role is understood and they could be well referred to as the first and the second part of the URI. For example, in Figure 6.1 the event h_1 represents the action of putting the order identified as 4c4712.

In Chapter 5 we introduced event scoping and trace filtering. We often want to scope an event by specific attributes. Let *AttributeName* denote set of all event attributes

$$[\mathit{AttributeName}]$$

and $e@A$ denote the attribute scoping function that scopes an event e to the set of attribute names in A.

$$_@_ : \mathit{Event} \times (\mathbb{P}\, \mathit{AttributeName}) \longrightarrow \mathit{Event}$$

For example, $h_1@\{\texttt{method, action}\} = \texttt{[PUT, order]}$. For clarity of presentation events are represented as collections of attribute values enclosed in `[square brackets]`.

Similarly, a trace, strand and norm attribute scoping function $X@A$ is defined as the attribute scoping (based on attributes in A) of events in the trace, strand and norm X respectively. For example, the following listing depicts `scopedLog` defined as `httpLog@{user,method,action,item}`.

```
scopedLog = httpLog@{user,method,action,item} =
 ⟨[frank, PUT, order, 4c4712], [alice, PUT, order, 1d261e],
  [lucy, GET, order, 4c4712], [lucy, PUT, invoice, 4c4712],
  [lucy, GET, order, 1d261e], [frank, GET, invoice, 4c4712],
  [lucy, PUT, invoice, 1d261e], [frank, PUT, order, 61ec0c],
  [alice, GET, invoice, 1d261e], [lucy, GET, order, 61ec0c],
  [lucy, PUT, invoice, 61ec0c], [frank, GET, invoice, 61ec0c]⟩
```

6.2.1 Strands for HTTP events

Strands are defined as traces of equivalent events. Let $\sim_A$ denote the event equivalence relation based on equality of the attributes in A. Given a set of attributes A and events $e, f : Event$ then define

$$e \sim_A f \Leftrightarrow e@A = f@A$$

for example, $h_1 \sim_{\{method\}} h_2$. For `date` defined as a date portion of the `time` attribute, $\sim_{\{\texttt{date}\}}$ equivalence partitions `httpLog` into the two strands $\langle h_1, .., h_7 \rangle$ and $\langle h_8, .., h_{12} \rangle$, reflecting the common practice of "rolling" logs on a daily basis. An alternative view might consider requests from the same user to be equivalent, partitioning the log into three strands involving `frank`, `alice` and `lucy`.

The event equivalence relation $\sim_{\{\texttt{item}\}}$, defined in terms of `item`-equality, groups events together as actions carried out on an order. In this case Figure 6.2 is a set of strands, based on the equivalence relation $\sim_{\{\texttt{item}\}}$, defined over the log scoped to {`user, method, action, item`} attributes: $prtn(\sim_{\{\texttt{item}\}}, \texttt{scopedLog})$. The value of an attribute used in the equivalence relation is underlined.

```
{⟨[frank, PUT, order, 4c4712 ], [lucy, GET, order, 4c4712],
  [lucy, PUT, invoice, 4c4712], [frank, GET, invoice, 4c4712]⟩,
 ⟨[alice, PUT, order, 1d261e], [lucy, GET, order, 1d261e],
  [lucy, PUT, invoice, 1d261e], [alice, GET, invoice, 1d261e]⟩,
 ⟨[frank, PUT, order, 61ec0c], [lucy, GET, order, 61ec0c],
  [lucy, PUT, invoice, 61ec0c], [frank, GET, invoice, 61ec0c]⟩}
```

Figure 6.2: Strands from `httpLog` partitioned by attribute `item`

An alternative view that partitions the log according to the equivalence relation $\sim_{\{\texttt{item,user}\}}$ into strands of operations carried out by a given user on a given item

is depicted in Figure 6.3.

```
{⟨[frank, PUT, order, 4c4712], [frank, GET, invoice, 4c4712]⟩,
 ⟨[frank, PUT, order, 61ec0c], [frank, GET, invoice, 61ec0c]⟩,
 ⟨[alice, PUT, order, 1d261e], [alice, GET, invoice, 1d261e]⟩,
 ⟨[lucy, GET, order, 4c4712], [lucy, PUT, invoice, 4c4712]⟩,
 ⟨[lucy, GET, order, 1d261e], [lucy, PUT, invoice, 1d261e]⟩,
 ⟨[lucy, GET, order, 61ec0c], [lucy, PUT, invoice, 61ec0c]⟩}
```

Figure 6.3: Strands from `httpLog` partitioned by attributes `item` and `user`

6.2.2 Norms for HTTP traces

Trace scoping can be used to construct a definition for trace equivalence whereby, given traces t, s, a set of attributes A and a trace attribute scoping function $@$, then

$$t \approx_A s \Leftrightarrow t@A \equiv s@A$$

The three strands depicted in Figure 6.2, $prtn(\sim_{\{\texttt{item}\}}, \texttt{scopedLog})$ reflect an `item`-centric view of the HTTP log. Within each strand there is a common pattern of behavior, specifically, item ordering actions followed by item invoicing actions. This behavior may be defined in terms of norm

$$prtn(\approx_{\{\texttt{method,action}\}}, prtn(\sim_{\{\texttt{item}\}}, \texttt{scopedLog}))$$

depicted in Figure 6.4, with the attributes used for trace equivalence in bold. Note that the norm contains only a single set of strands, the same as depicted in Figure 6.2. The norm is a collection of strands that have an equivalent behavior pattern (according to $\approx_{\{\texttt{method,action}\}}$), each carried out on an identified target (in this case, `item`). Scoping the strands to the set of attributes used for trace equivalence that is,

$$prtn(\approx_{\{\texttt{method,action}\}}, prtn(\sim_{\{\texttt{item}\}}, \texttt{scopedLog}))@\{\texttt{method,action}\}$$

reduces the norm into the following single strand representing the common sequence of order processing operations.

```
{{⟨[PUT, order], [GET, order], [PUT, invoice], [GET, invoice]⟩}}
```

{{⟨[frank, **PUT**, **order**, 4c4712], [lucy, **GET**, **order**, 4c4712],
[lucy, **PUT**, **invoice**, 4c4712], [frank, **GET**, **invoice**, 4c4712]⟩,
⟨[alice, **PUT**, **order**, 1d261e], [lucy, **GET**, **order**, 1d261e],
[lucy, **PUT**, **invoice**, 1d261e], [alice, **GET**, **invoice**, 1d261e]⟩,
⟨[frank, **PUT**, **order**, 61ec0c], [lucy, **GET**, **order**, 61ec0c],
[lucy, **PUT**, **invoice**, 61ec0c], [frank, **GET**, **invoice**, 61ec0c]⟩}}

Figure 6.4: A behavioral norm from `httpLog` partitioned by attribute `item` for operations {`method, action`}

Different event and trace equivalences result in different norms, reflecting different kinds of patterns of behavior within the system. For example, Figure 6.3 depicts a partition of the log into strands that define the actions of a given `user` on a given `item`. The strand

⟨[frank, PUT, order, 4c4712], [frank, GET, invoice, 4c4712]⟩

from this partition represents `frank` ordering and invoice-processing item `4c4712`. Across these strands is a repeating user-order-invoice behavior norm that can be identified in terms of strands that have equivalent `method` and `action` attributes, that is, by the (strand) trace equivalence relation $\approx_{\{\texttt{method,action}\}}$. In this case, the strand partition of the log (given in Figure 6.1) is partitioned by $prtn(\approx_{\{\texttt{method,action}\}}, prtn(\sim_{\{\texttt{user,item}\}}, \texttt{scopedLog}))$ into a customer norm and a merchant norm, depicted in Figure 6.5. A customer (norm)

{{⟨[frank, **PUT**, **order**, 4c4712], [frank, **GET**, **invoice**, 4c4712]⟩,
⟨[frank, **PUT**, **order**, 61ec0c], [frank, **GET**, **invoice**, 61ec0c]⟩,
⟨[alice, **PUT**, **order**, 1d261e], [alice, **GET**, **invoice**, 1d261e]⟩},
{⟨[lucy, **GET**, **order**, 4c4712], [lucy, **PUT**, **invoice**, 4c4712]⟩,
⟨[lucy, **GET**, **order**, 1d261e], [lucy, **PUT**, **invoice**, 1d261e]⟩,
⟨[lucy, **GET**, **order**, 61ec0c], [lucy, **PUT**, **invoice**, 61ec0c]⟩}}

Figure 6.5: Behavioral norms from `httpLog` partitioned by attributes {`user, item`} and operations {`method, action`}

puts orders and gets invoices while the merchant (norm) gets orders and puts invoices. These norms also suggest user roles, whereby Frank and Alice are customers and Lucy is the merchant. Consequently, scoping the strands to the set of attributes used in the trace equivalence,

$$prtn(\approx_{\{\texttt{user,item}\}}, prtn(\sim_{\{\texttt{item}\}}, \texttt{scopedLog}))@\{\texttt{user, item}\}$$

reduces norms to singleton sets of strands representing the common sequence of order processing operations by users in different roles.

```
{{⟨[PUT, order], [GET, order]⟩},
 {⟨[PUT, invoice], [GET, invoice]⟩}}
```

6.3 N-gram Based Trace Equivalence

The sample strands used to illustrate the behavioral norms in HTTP logs in the previous section were deliberately simple in order to present the basic concepts of the model. Within those strands, the trace equivalence relation is defined as simple equality of traces scoped to certain attributes. This, however, is impractical for larger traces that capture more complex behaviors than just simple static sequences. For example, Figure 6.6 depicts a set of 10 `item`-centric strands from another log fragment of the HTTP application, scoped to {`method, action`} attributes.[1] The strands capture sequences of operations related to the creation

```
PUT cart     PUT cart     PUT cart     PUT cart     PUT cart
POST cart    POST cart    POST cart    POST cart    POST cart
PUT order    POST cart    POST cart    POST cart    POST cart
             PUT order    POST cart    POST cart    POST cart
                          PUT order    POST cart    POST cart
                                       PUT order    POST cart
                                                    PUT order

PUT cart     PUT cart     PUT cart     PUT cart     PUT cart
POST cart    POST cart    POST cart    POST cart    POST cart
DELETE cart  POST cart    POST cart    POST cart    POST cart
             DELETE cart  POST cart    POST cart    POST cart
                          DELETE cart  POST cart    POST cart
                                       DELETE cart  POST cart
                                                    DELETE cart
```

Figure 6.6: Strands from the HTTP log for cart creation and updates

of a shopping cart, updating it (adding and removing items) and proceeding with an order (first 5) or deleting the cart (last 5). Scoping the trace to operation

[1] For improved readability, we omit the set `{}` and event `[]` parentheses.

attributes provides a level of abstraction, and the log might have contained a number of instances of, for example, creating a cart with three items and then putting an order.

However, even such simple activity, captured at the limited scope, may result in a large number of strands that, even though are representing very similar behavior, are different. Partitioning this set of strands using trace equivalence relation defined as trace equality will result in ten distinct norms. It is more practical to use an approximate matching that will allow strands with small structural differences, such as repetition of the same sub-sequence, to be considered equivalent. We decided to approximate strands as collections of n-grams [14], short sub-sequences of a trace. In general, n-grams are sets of sub-sequences of size n of a given sequence. For example, sequences $\langle$`a, b, a, b`$\rangle$, $\langle$`a, b, a, b, a, b`$\rangle$ and $\langle$`a, b, a, b, a, b, a, b`$\rangle$ can be all represented by a set of two 2-grams $\{\langle$`a, b`$\rangle, \langle$`b, a`$\rangle\}$.

Let *ngrams* define a function that maps a trace to corresponding set of n-grams.

$$\begin{array}{|l} ngrams : Trace \times \mathbb{N}_1 \longrightarrow \mathbb{P}\ Trace \\ \hline \forall\, t : Trace; n : \mathbb{N}_1 \bullet ngrams(t, n) = \{ng : Trace \mid ng \text{ in } t \wedge \#ng = n\} \end{array}$$

Note, in Z in denotes a sub-sequence relation.

For example, the set of n-grams for the first strand in Figure 6.6 is

$\{\langle$`[PUT, cat], [POST, cart]`$\rangle, \langle$`[POST, cart], [PUT, cart]`$\rangle\}$,

and the set for the four subsequent strands is

$\{\langle$`[PUT, cat], [POST, cart]`$\rangle, \langle$`[POST, cart], [POST, cart]`$\rangle,$
$\langle$`[POST, cart], [PUT, cart]`$\rangle\}$.

N-grams can be used to construct a definition for trace equivalence, whereby traces with the same sets of n-grams are considered equivalent. Given traces t, s, a set of attributes A attribute scoping function $@$, and n-gram length n

$$t \approx^n_A s \Leftrightarrow ngrams(t@A, n) = ngrams(s@A, n).$$

For example, the equivalence relation $\approx^2_{\{\texttt{method,action}\}}$ partitions ten strands in Figure 6.6 into four norms, including strands 1, 2–5, 6 and 7–10 respectively. Note that the set of n-grams represents an equivalence class for the corresponding set

of strands. Therefore, where no ambiguity arises, when discussing norms created using n-gram equivalence relation, we refer to the set of n-grams as "norm". Section 7.3.1 provides further examples of n-gram norms for traces of application method calls.

6.4 Norm Search

In previous sections we demonstrated that the selection of equivalence relations, and the event attributes that are used to define them, results in different norms. A norm identifies common sequences of *operations* carried out relative to a *target*. Let O and T denote sets of attributes intended to represent the operation and the target of events, respectively. For example, in the event `[frank, PUT, order, 4c4712]` the attributes `{method, action}` are the operation (`[PUT, order]`) and attribute `item` is the target (`4c4712`). Event equivalence relation $\sim_T$ distinguishes targets while strand equivalence relation $\approx_O$ distinguishes operations. For example, in Figure 6.4, order-invoice operations are carried out relative to `item` targets.

Identifying norms in a log involves partitioning that log into strands using an event equivalence relation and then partitioning the strands into norms using a trace equivalence relation. Given operation and target attribute sets O and T, respectively, then $l@(O \cup T)$ gives the view of interest of the log l. In this case, the norms of log l, defined as

$$\mathcal{N}_T^O(l) = prtn(\approx_O, prtn(\sim_T, l))@O$$

are intended to represent patterns of operations on targets. Note that events not in $O \cup T$ are considered superfluous while targets are not explicitly included in the final set of norms since their existence is implicit in the strands they relate to. In order to identify meaningful norms in the log, it is crucial to properly select the attributes to be used in respective equivalence relations. Also, in the case of the n-gram based trace equivalence, it is required to define the length of the n-grams. In the next section, we discuss a systematic technique for identifying the best selection of these parameters, using false-positive and false-negative metrics.

6.5 Attribute Search

In this section we demonstrate a systematic approach for identifying the attributes suitable for the norm model and, consequently, the norms themselves. We are interested in identifying likely target and operation attributes for event and strand equivalence relations that generate norms that provide meaningful characterizations of a system's behavior. Considering a norm defined as a `time` operation on a `method` target, that is,

$$prtn(\approx_{\{\texttt{time}\}}, prtn(\sim_{\{\texttt{method}\}}, \texttt{httpLog}))$$

does not reveal anything interesting about the `httpLog`. However, the norms in Figure 6.5 do reveal potentially interesting customer and merchant related norms.

The intention is to compute these norms $\mathcal{N}_T^O(l)$ on the basis of a single *learning* log l. The effectiveness of using $\mathcal{N}_T^O(l)$ as a representative model of the given system's behavioral norms is determined by comparison with the norms generated by a further *test* log t of valid interactions of the system.

Let $\mathcal{M}_T^O(l,t)$ define an operation that compares the norms of $\mathcal{N}_T^O(l)$ and $\mathcal{N}_T^O(t)$ and returns a measure in [0..1] that indicates their degree of similarity to each other, whereby a higher value indicates a greater degree of similarity. Intuitively, $\mathcal{M}_T^O(l,t)$ gives a measure of the false negatives when one treats $\mathcal{N}_T^O(l)$ as a model of the behavioral norms in the system. The higher numerical value represents a lower measure of false negatives. A measure of the false positives for $\mathcal{N}_T^O(l)$ as a model of behavioral norms is given by $\mathcal{M}_T^O(l,c)$, where c is a *control* log of the system, that is a log with sequence perturbations that are known not to occur in the system. Section 6.5.1 outlines the encoding of $\mathcal{M}_T^O(l,t)$ based on n-grams.

6.5.1 N-gram based norm similarity

As discussed in Section 6.3 n-grams provide an approximate trace matching and can be used to implement the trace equivalence relation ($\approx$). This is generalized to norms, whereby a set of strands (a norm) is encoded as a set of n-grams that provide an (approximate) method to test a strand for membership of a norm. A variation of the Jaccard coefficient [83] is used to provide a measure of the similarity between two sets (norms) of n-grams. Jaccard coefficient is the simplest metric for set similarity defined as the size of the intersection of the

sets divided by the size of the union of the sets. In our experiments focused on identifying attributes we have found the measure provided by Jaccard coefficient sufficient. Future work should consider usage of another set similarity measures and its applicability.

Let $\mathcal{J}(n, m)$ denote this measure between norms n and m:

$$\mathcal{J}(n, m) = \frac{\mid n \cap m \mid}{\mid n \cup m \mid}$$

Given a log l and a test trace t then the average of the best of the Jaccard coefficients from the log norms in $\mathcal{N}_T^O(l)$ to the test norms in $\mathcal{N}_T^O(t)$ defines a similarity measure between the sets of norms is 0. Thus, given operation and target attributes O and T, log and test traces l and t, and an underlying n-gram model then define:

$$\mathcal{M}_T^O(l, t) = \sum_{n \in \mathcal{N}_T^O(l)} \left(\frac{\max_{m \in \mathcal{N}_T^O(t)} \mathcal{J}(n, m)}{\mid \mathcal{N}_T^O(l) \mid} \right)$$

The same calculation is used to define the similarity $\mathcal{M}_T^O(l, c)$ between the log trace t and the control trace c.

Intuitively, $\mathcal{M}_T^O(l, t)$ measures how well the norms in set $\mathcal{N}_T^O(l)$ are covered in set $\mathcal{N}_T^O(t)$. This measure is not a *distance* in the mathematical sense. For example, if $\mathcal{N}_T^O(l)$ and $\mathcal{N}_T^O(t)$ are equal or $\mathcal{N}_T^O(l)$ is a subset of $\mathcal{N}_T^O(t)$ the similarity measure is 1. If $\mathcal{N}_T^O(l)$ contains norms that have no n-grams in common with any norm in $\mathcal{N}_T^O(t)$, the similarity measure is 0. Table 6.1 demonstrates the calculation of model similarity metric. The first part of the table contains norms $\{n_1, n_2, n_3\}$ for the log l, and the second norms $\{m_1, m_2, m_3\}$ the log t. The norms contain n-grams from the set $\{a_1, \ldots, a_5\}$. The third part of the table includes all Jaccard coefficients between norms for the two logs with maximal value highlighted in bold. The similarity between the two sets is an average of the maximal values, that is 0.64.

$\mathcal{N}_T^O(l)$	
n_1	$\{a_1, a_2\}$
n_2	$\{a_1, a_2, a_3, a_5\}$
n_3	$\{a_1, a_5\}$

$\mathcal{N}_T^O(t)$	
m_1	$\{a_1, a_2\}$
m_2	$\{a_1, a_2, a_3, a_4\}$
m_3	$\{a_2, a_3\}$

$\mathcal{J}(n, m)$	m_1	m_2	m_3
n_1	**1.00**	0.50	0.00
n_2	0.50	**0.60**	0.50
n_3	**0.33**	0.20	0.00

Table 6.1: Norm model similarity calculation example

6.5.2 Implementation of attribute search

Given traces l (log), t (test) and c (control), the attribute search finds operation attributes O and target attributes T and an n-gram model ($\approx$) resulting in the best values for $\mathcal{M}_T^O(l,t)$ and $\mathcal{M}_T^O(l,c)$. A prototype of this search has been implemented [20]. Figure 6.7 depicts the pseudo-code of the attribute search algorithm. Given the traces (l,t,c) and the set of event attributes A the algorithm

```
function attributeSearch(l, t, c, A)
    targets = 2^A \ {∅}
    for (T in targets) {
        operations = 2^(A\T) \ {∅}
        for (O in operations) {
            result[O][T] = [M_T^O(l,t), M_T^O(l,c)]
        }
    }
    return result
}
```

Figure 6.7: Pseudo-code of the attribute search algorithm

returns the similarity measures for the pairs of operation and target attributes within A. Note that 2^A denotes the power set of the set of attributes, that is all possible sets of attributes defined by A. The algorithm iterates over possible sets of attributes for the target and operation and calculates the norm similarity metrics for each attribute pair. Note that the attributes selected for the target are not considered for operation. This is because these attributes are used to separate strands from each other, and therefore have a static value within the strand, which makes them unsuitable to be operation attributes. Also, empty sets of attributes are not considered in the search. While the implementation is quite effective for demonstrating this for the moderately-sized logs and attribute sets described in the next section, we have not focused on search efficiency. In principle, the search is exponential in its parameters. Note that other parameters of the model, such as the length of n-gram, are externalized from the algorithm. The algorithm can be used to find the optimal values for those parameters by iterating over the candidate values for the parameters. For example, in order to find the best n-gram length, the attribute search may be repeated for each of the sizes. Also, one could first identify the best operation/target attributes for one n-gram length and then execute the algorithm for a range of n-gram lengths and reduced set of good operation/target attributes.

6.6 Attribute Search Evaluation

Two experiments were carried out in order to evaluate whether the norm model attributes, and corresponding norms, could be discovered from system logs. Section 6.6.1 describes the discovery of norms from the logs of a simulated system. Section 6.6.2 documents norms that were discovered in our study of a complex enterprise-grade application system.

6.6.1 Norms in a simulated system

A system that simulated the execution of the HTTP example in Section 6.2 was developed. It was extended to include additional actions such as `cart`, `payment`, `dispatch` and `return`, and methods such as `POST` and `DELETE`. The fragment of sample execution of the application is depicted in Figure 6.8. The objective

user	**method**	**action**	**item**
userA	PUT	payment	1d261e
userB	DELETE	order	1d261e
userD	GET	dispatch	61ec0c
userF	PUT	return	4c4712
userC	POST	order	1d261e
userD	GET	cart	4c4712
userF	PUT	invoice	61ec0c
userG	POST	payment	1d261e
userB	GET	invoice	4c4712
userE	DELETE	cart	61ec0c

Figure 6.8: A fragment of a log from simulated system

of this experiment was to demonstrate that norms could be discovered in logs of systems that were fabricated deliberately to have repeating patterns within their apparently random-looking behavior. In particular, that the attribute search would find the sets of operation O and target T attributes as expected.

The system was built as a collection of concurrent users, engaging HTTP events. A simple control-flow model was used to specify user behavior scenarios in terms of repeating sequences of events to be carried out across (random) items. Figure 6.9 depicts sample scenario, where each operation representing an operation

executed by a user in a transaction, such as `operation12: [$user1, GET, invoice]`. During the simulation, a set of users is selected at random to execute sequences from the choice defined by the scenario.

```
SCENARIO [
  SEQUENCE {
    operation12
    operation4
    operation31
    operation14
  }
  CHOICE {
    SEQUENCE {
      operation37
      operation17
    }
    SEQUENCE {
      operation37
      operation17
    }
  }
  LOOP {
    operation17
    operation57
    operation10
    operation3
  }
  CHOICE {
    SEQUENCE {
      operation12
      operation9
    }
  }
  SEQUENCE {
    operation11
    operation4
    operation7
    operation24
    operation35
    operation10
    operation68
  }
]
```

Figure 6.9: A sample user behavior scenario for simulated system (simplified)

The simulation comprised of 500 transactions performed by a group of 25 users, each selecting executions at random from the choice of 30 randomly generated scenarios. Running the simulation twice generated different learning l and test t traces, each containing approximately 50,000 events. A similar sized control log c was generated by re-running the simulation, with the different set of scenarios. The search was repeated three times to consider the norm models using n-gram size of 3, 5 and 7.

While the event attributes included `user`, `method`, `action` and `item`, it should be noted that in this experiment no information was provided to the search algorithm that might a priori suggest candidate operation and target attributes. Each event also included a transaction identifier (`trans`) that tied the event to a unique execution sequence within a user behavior scenario.

Table 6.2 provides a selection of operation and target attributes for different n-gram sizes, along with $\mathcal{M}_T^O(l,t)$ (false negative) and $\mathcal{M}_T^O(l,c)$ (false positive) measures based on the simulated log traces l, t and c.

These results suggest, as expected, that the repeating pattern of `method-action` operations carried out on transactions (`trans`) is a good norm; providing a high

operation O	target T	n	$\mathcal{M}_T^O(l,t)$	$\mathcal{M}_T^O(l,c)$
`trans`	`user`	3	0.14	0.45
`item`	`method`, `user`	7	0.18	0.47
`trans`	`item`	3	0.18	0.46
`method`, `action`	`user`	5	0.28	0.00
`method`, `user`	`trans`	3	0.32	0.01
`method`, `action`	`item`, `user`	3	0.75	0.02
`method`, `action`	`trans`	7	0.78	0.03
`method`, `action`	`trans`	5	0.81	0.00
`method`, `action`	`trans`	3	0.85	0.02

Table 6.2: Some norms in the simulated system

degree of similarity between the learning and test logs (few false negatives) and a low degree of similarity between the learning and control logs (few false positives). While attribute `trans` was intentionally constructed to provide a unique transaction identifier for the user-scenario, it is interesting to note that the search also suggests the pair `{user, item}` as a reasonable set of target attributes. On further investigation it turned out that in the simulation logs, it was more likely that the execution scenarios of different users involved different items, that is, there were relatively few instances of order-invoice transactions involving multiple users. This could be regarded as an unexpected norm that emerged as a consequence of the simulation design.

The problem of finding the best selection of attributes is exponential with respect to the number of attributes in the event. For five attributes, an exhaustive search over space of all possible combinations of attribute for target (2^5), operation (also 2^5) and n-gram sizes (3) would require 3072 algorithm iterations. Thanks to some basic optimizations (such as not considering operation attributes in targets) in the experiment we reduced that number to 510 which took about 30 minutes on a mid range laptop. It should be noted that the experiment was performed using an early implementation of the norm model [20]. The later efficient implementation, as described in Chapter 7, significantly reduced model comparison time, to values in range of 80 milliseconds on the same setup. Using that implementation, the search process could be reduced to under a minute.

6.6.1.1 Norms similarity and aggregation

Encoding the strands with n-grams provides only a small degree of approximation. For example, for 500 execution traces in the experiment, the number of norms (represented as distinct n-gram sets) for operation `{method, action}` and target `trans` is in the range of 400-450. That means that only some of the strands converge to the same n-gram sets.

A further experiment was performed to observe how similar to each other are the norms generated for the simulated system trace. While this experiment is not related to the attribute search, it provides an insight into potential applications of the behavioral norms model. In this experiment we select decreasing thresholds of similarity (from 1 to 0 with step 0.01) and aggregate the norms that are similar (according to Jaccard coefficient) to each other over that threshold. Because norms are sets of n-grams, the aggregation is a union of the sets of n-grams. As a result, we obtain a set of norms aggregated at all 101 similarity levels. Figure 6.10 depicts the pseudo-code of the algorithm used in the experiment.

The experiment was performed for 10 different simulations of the user behavior scenarios in the system. Figure 6.11 plots the numbers of aggregate norms for decreasing thresholds of norm similarity. With a degree of similarity of 1, the norms must exactly match and there is a large number of distinct norms, many with very similar but strictly different behavior. With a degree of similarity of 0, traces of the same events but differently ordered are matched and, as a result, there is effectively one norm that blends all system activities. Note that this level of similarity when applied to an n-gram based model, reduces behavioral norms to a set of n-grams, as described in [17].

Inspecting Figure 6.11 reveals that 30 distinct aggregate norms are identified for a substantial range of the norm similarity threshold values between 0.5 and 0.05. Recall that in configuring the control-flow model, logs were generated by users repeatedly selecting, at random, from a choice of 30 different execution scenarios. On inspection, each norm corresponds to one of the execution scenarios, confirming that the proposed operation O and target T attributes reflect the patterns of behavior that were intended. Naturally, the model with similarity levels in this range is not useful for the purpose anomaly detection, however, this experiment demonstrates a potential additional application of the norm model, that is, identifying underlying patterns of behavior in an application. In the case of the simulated system, analysis of the HTTP log allowed the identification of

```
function aggregate(𝒩) {
    for (sim 1..0 step -0.01) {
        norms[sim] = 𝒩
        repeat:
        for (n in norms[sim]) {
            for (m in norms[sim]) {
                if (n ≠ m ∧ 𝒥(n, m) > sim) {
                    norms[sim] = norms[sim]\{n, m}
                    norms[sim] = norms[sim]∪{n ∪ m}
                    goto repeat
                }
            }
        }
    }
    return norms
}
```

Figure 6.10: Pseudo-code of norm aggregation algorithm

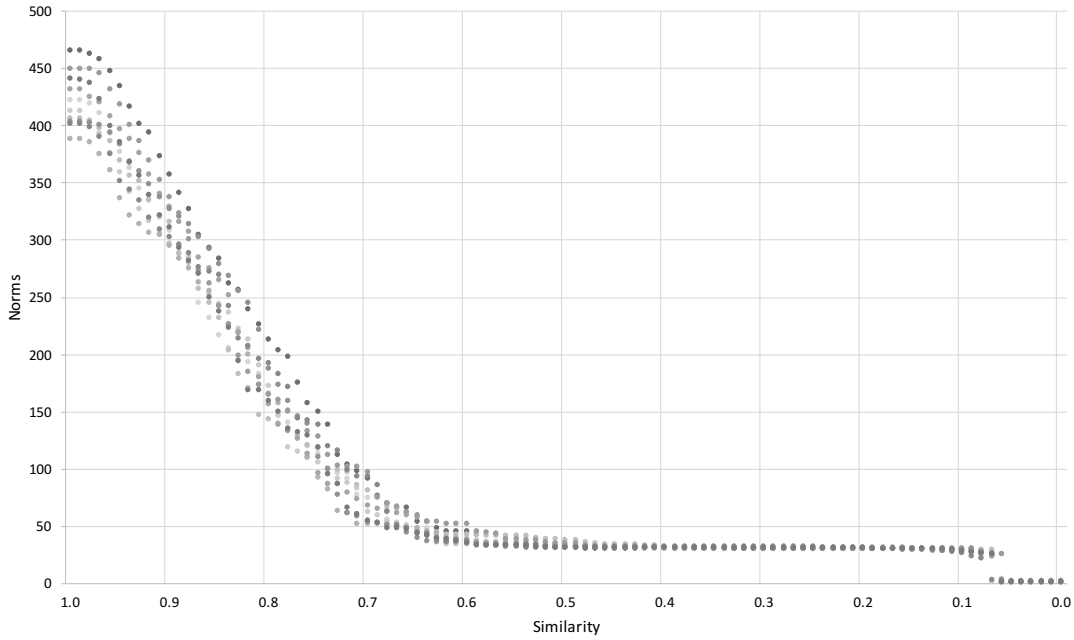


Figure 6.11: Number of aggregate norms for different norm similarity thresholds (simulated system)

the patterns of user behavior.

6.6.1.2 Simulating anomalies

In the following experiment, the system simulating the HTTP application was extended to model an access control mechanism. The access control was implemented by assigning each of the scenarios and each of the users one of five roles: `Customer`, `Merchant`, `Warehouse`, `Support` and `Supervisor`. The simulation has been modified, so that scenarios are only executed by users with a matching role. For illustration, the part of a scenario related to making an order may only be executed by users with the role `Customer`, while the part related to issuing an invoice, by users with the role `Merchant`. Information about user role was included as an additional event attribute and new events contained 6 attributes (`method`, `action`, `user`, `item`, `role`, `trans`). Figure 6.12 depicts a typical trace of the system including roles.

user	**role**	**method**	**action**	**item**
userA	Customer	PUT	payment	1d261e
userB	Customer	DELETE	order	1d261e
userD	Warehouse	GET	dispatch	61ec0c
userF	Customer	PUT	return	4c4712
userC	Merchant	POST	order	1d261e

Figure 6.12: A fragment of a log from a simulated system including roles

The attribute search was performed for the modified system. Optimal values identified by the search are {`role`, `method`, `action`} for the operation and `trans` for the target. It should be noted that, compared with the system without the access control capability, the operation includes `role` in addition to `method` and `action`. Inclusion of the `role` attribute makes the operation more precisely defined (for example, "GET invoice as Customer" compared to just "GET invoice") and produces a more accurate norm model. The norm aggregation, as presented in Figure 6.13 (Access control enabled) shows the result similar to the original system (without access control) and reflects 30 scenarios for a large similarity threshold value range.

In the second part of this experiment, the simulation was modified to model a security flaw. The access control was disabled so any user could run any scenario

regardless of their role. The intention of this change was to simulate an accidental misconfiguration that breaks security control and check how such change will reflect on system's norms. The analysis of the number of aggregate norms for different similarity threshold values was performed on the trace from modified system. Figure 6.13 shows the number of aggregate norms in relation to the similarity threshold value for both executions (with access control enabled and disabled) of the system with the access control capability.

This experiment shows that, at the lower similarity threshold values, the operation of the system with broken access control is reflected using more aggregate norms than the one with the access control functioning properly. Although both systems were executing the same 30 scenarios, they produce traces that are behaviorally different according to identified norm attributes. In the system with enabled access control, different executions of the same scenario produced traces only different because of the control flow. For norm similarity threshold values between 0.5 and 0.05 the executions of the same scenario were considered equivalent. In the system with broken access control, executions of the same scenario, but with different roles, produced traces in which *operations* were also different. Even at low similarity levels such traces could not be considered equivalent and were parts of different norms.

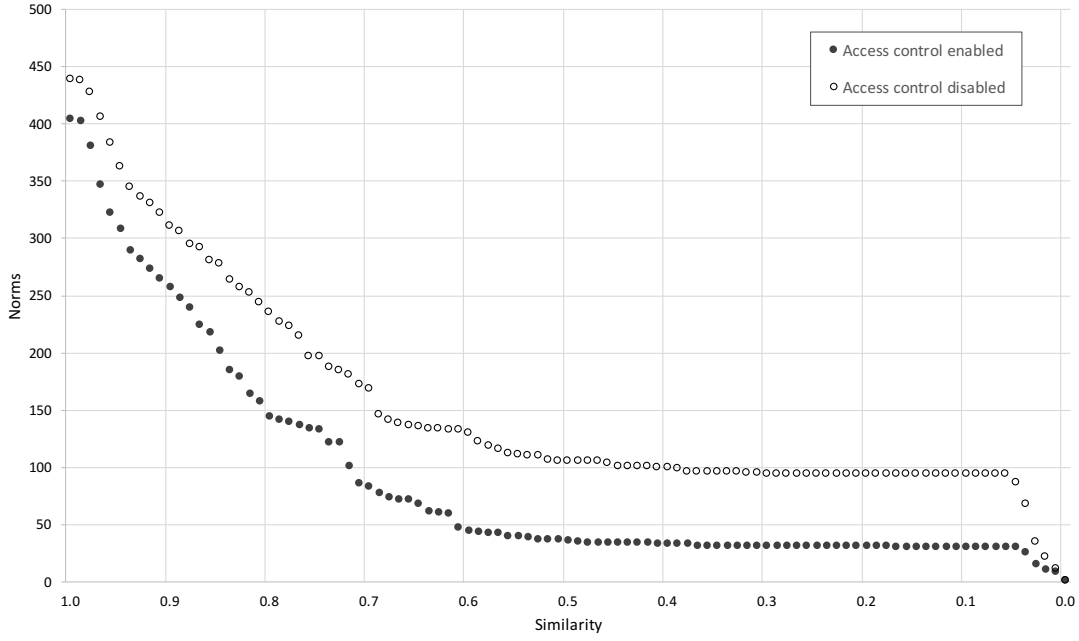


Figure 6.13: Number of aggregate norms for different norm similarity thresholds for two configurations of simulated system with access control

This experiment demonstrates that for certain thresholds of norm similarity the change in the system's behavior may be identified by observing the number of aggregate norms that emerge from that system. This is possible without a priori knowledge about what behavior the individual norms actually represent.

6.6.2 Norms in an enterprise system

The second experiment was based around an enterprise Java-based social software application. This application provides its users with social communication and content management. The objective of this experiment was to investigate the efficacy of the attribute search in an existing large-scale application system.

The application server running the social software application was configured to include a custom Java Security Manager that recorded every permission check as an event. Six attributes of the events logged by the manager were considered: type of permission (attribute `perm`, with values `FilePermission`, `SocketPermission`, etc.); `action` (with values, `open`, `read`, etc.); name of application server's `thread` used to perform the operation; application's `user` on behalf of which the thread executes, and name of the `class` that invoked the code requiring the permission. Time was recorded using reduced precision. A fragment of the trace from the system is depicted in Figure 6.14.

time	class	perm	action	thread	user
[05/Nov/2012:09:11]	UploadAction	File	open	Thread-4	frank@example.com
[05/Nov/2012:09:11]	DownloadAction	Socket	connect	Thread-5	lucy@example.com
[05/Nov/2012:09:11]	UploadAction	File	write	Thread-4	frank@example.com
[05/Nov/2012:09:11]	DownloadHelper	Socket	write	Thread-5	lucy@example.com
[05/Nov/2012:09:11]	DownloadHelper	File	read	Thread-5	lucy@example.com

Figure 6.14: A fragment of a log from enterprise system

The application was invoked 1,500 times via its REST API in order to execute 11 different high-level operations related to file management functionality of the application: get service status, upload new file, update file contents, change file name, download file, delete file, create folder, change folder name, delete folder, add file to folder and remove file from folder. The file and folder names and file contents were selected at random and file size was between 100 bytes to 500 kilobytes. The REST API calls were made concurrently for 10 different application users. This was done twice in order to generate the learning and test traces, each containing about 30,000 events, each representing recorded permission check. The logs from the Java Security Manager were not scoped in any way.

The control trace was created by randomly reordering events from the test trace, in order to use the same events, but in an order different than during normal execution of the application. The attribute search was executed separately for the norm models with n-gram sizes of 3, 5 and 7. Table 6.3 provides a selection of operation and target attributes, along with $\mathcal{M}_T^O(l,t)$ (false negative) and $\mathcal{M}_T^O(l,c)$ (false positive) measures that were computed during the search. The results in Table 6.3 are for an n-gram size of 3, which, when analyzing this system, produced better results, that is lower measures for both false positive (lower numerical value) and false negative (higher numerical value).

operation O	target T	$\mathcal{M}_T^O(l,t)$	$\mathcal{M}_T^O(l,c)$
`perm`	`action`, `thread`, `user`	0.38	0.49
`class`, `user`	`action`, `prtm`, `time`	0.15	0.01
`name`, `perm`	`thread`, `trace`	0.51	0.37
`perm`	`time`, `user`	0.99	0.11
`class`, `perm`	`time`, `user`	0.97	0.07
`class`, `perm`	`thread`, `time`, `user`	0.99	0.08
`action`, `class`	`thread`, `time`, `user`	0.92	0.00
`action`, `class`, `perm`	`thread`, `time`, `user`	0.92	0.00

Table 6.3: Some norms in the enterprise system

The results show that the best candidates for operation are the combinations of `action`, `perm` and `class` attributes. It is interesting to note that the operation with attributes {`action`, `class`} has results comparable to {`action`, `class`, `perm`}. Intuitively, the latter seems to be a better choice as it defines actual operation more precisely (`[SomeClass, open, file]` compared to just `[SomeClass, open]`). However, it is common that every permission has a distinct set of actions specific to that permission. For example, `FilePermission` is defined in terms of `read`, `write`, `execute`, `delete` actions, while `SocketPermission` is defined in terms of `accept`, `connect`, `listen` and `resolve` actions. Thus, for this system, permission name does not provide any additional information to the operation.

Studying the effect that the norms aggregation has on the number of norms identified in $\mathcal{N}_T^O(l)$, for the best O and T, provides some interesting insights. In Figure 6.15, when the threshold of norm similarity varies between, 0.34 and 0.17, then 11 aggregate norms are identified. This is expected, as these 11 aggregate norms effectively correspond with the underlying behavior patterns resulting from

repeatedly invoking the 11 different REST API calls in order to populate the log.

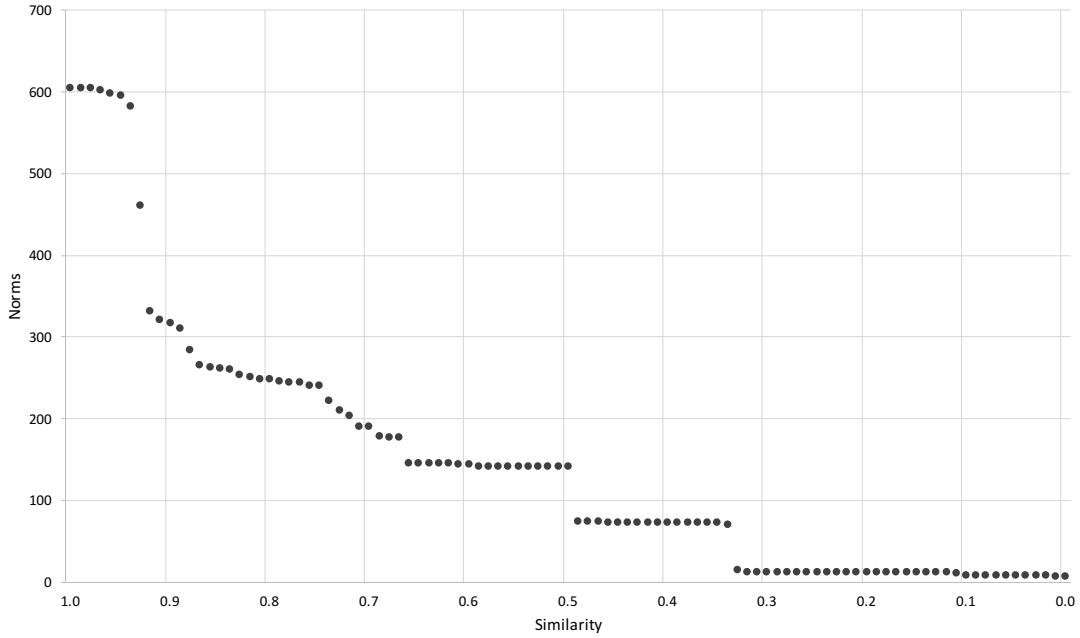


Figure 6.15: Number of aggregate norms for different norm similarity thresholds (enterprise system)

However, Figure 6.15 also points to the presence of other potential patterns of behavior in the system. There are three other regions in the graph that suggest different numbers of aggregate norms. For similarity threshold values: between 0.66 and 0.55 there are 140 norms, between 0.49 and 0.45 there are 68 norms, and there are 7 norms between 0.17 and 0.05. We conjecture that these additional norms are a consequence of the application, or its underlying infrastructure, performing different kinds of internal operations for the same REST API calls. For example, for the same API call the application may, at one time, retrieve an object from remote storage while retrieving it from local cache another time; an application may once have established a connection, while at another time uses a previously established connection from a connection pool. Traces generated by different execution paths are sufficiently different to be result in distinct norms where a high degree of norm similarity is required. This means that it may be possible to use multiple threshold levels in order to simultaneously observe a larger number of more precise norms or a smaller number of more general ones. This experiment was a preliminary attempt to validate that meaningful norms can be identified in a real and complex enterprise system. In building the experiment,

many simplifications had to be made, such as ignoring client-side caching and calling only the REST API rather than accessing application through the web interface as a user.

6.6.3 Discussion

We demonstrated that it is possible to discover correct norm model attributes without any prior knowledge about the system. In addition, aggregating norms allows identifying underlying patterns of behavior of the system characterized by the set of aggregate norms. Such norms may provide an efficient way to compare and evaluate system behavior during its life cycle. For example, in Section 6.6.1.2 we demonstrated how the number of norms increases after an accidental misconfiguration of the system. Future research should consider if observation of the emergent behavior defined by the norms aggregated to a certain similarity threshold could be used as a general technique for monitoring system behavior operation. Norm discovery and the associated parameters such as similarity or false positive and false negative metrics are intended to provide an insight into the system operation, but these values do not correspond to the performance of anomaly detection mechanism built on norms. Similarly, the discovery process is not typically part of the end-to-end operation of anomaly detection, rather a step that allows the identification of suitable parameters for particular application or a platform. A number of possible further applications of behavioral norms is considered in [22].

6.7 Modeling Behavior of Collaborating Systems

In previous sections we have shown how different configuration of the norm model may lead to different views of the system. In this section we show how behavioral norms could be used to model a behavior of multiple systems collaborating together and used to identify anomalies in that behavior.

6.7.1 An online photograph sharing service

Consider an on-line photograph hosting and sharing service. The service allows users to upload and store their photographs, establish a network of friends with whom to share photographs, comment on photographs, and so forth. The service also provides activity tracking of the users and their friends. Users can view the actions they have performed (for example, the photographs they uploaded and when), and limited tracking of the actions of other users (for example, accesses and comments on the photographs they share). For example, Figure 6.16 provides a fragment of a log of such actions that are visible to the user `frank`.

time	user	context	action	id	extra
2013-11-04 16:53:05	frank	self	login	-	-
2013-11-04 16:55:21	frank	self	uploadPhoto	img23	Holidays 2013
2013-11-04 16:57:55	frank	self	uploadPhoto	img24	New bike
2013-11-04 17:01:03	frank	self	share	img23	lucy
2013-11-04 17:04:29	lucy	friend	viewPhoto	img23	-
2013-11-04 17:05:18	frank	self	share	img24	alice
2013-11-04 17:05:19	lucy	friend	comment	img23	I wish, I was there...
2013-11-04 17:21:34	alice	friend	viewPhoto	img24	-
2013-11-04 17:22:01	alice	friend	comment	img24	Nice|

Figure 6.16: A partial log from the photo hosting service

This activity data need not necessarily come from a conventional text log. Actions/events may be presented to the consumer by the provider using a web interface or as a feed in some common format such as RSS or ATOM and we assume that a consumer is able to view the events relevant to its interaction with the provider. Events are comprised of attributes; the events in Figure 6.16 have attributes that provide `time` of event, `user` name, `action` carried out, and whether the action is carried out by the user viewing the log (the `context` value `self`) a `friend` or `other` user, the image `id`, and any `extra` data.

Studying Figure 6.16, we see that `frank` logs-in, uploads two photographs, shares photographs with users `lucy` and `alice` who in turn view and comment. A closer inspection of the log in Figure 6.16 reveals what appears to be two, interleaving, transaction-like patterns of behavior. In the first, `frank` uploads a photo `img23`, shares it with `lucy` who then views and comments. In the second, the same sequence of actions occur in relation to `frank` sharing `img24` with user `alice`. This analysis identifies a simple transaction-style behavior in the log fragment.

⟨`uploadPhoto, sharePhoto, viewPhoto, commentPhoto`⟩

In identifying these transaction style patterns it is important to distinguish the roles that are played by the different event attributes. Intuitively, the attribute value `action` represents the operation being carried out by the event and this operation is effectively parameterized by the image identifier (target attribute `id`). Further study of the log is required to decide whether the `user`, `context` and `extra` attribute values should play a role in this transaction.

6.7.2 Norms in online photograph sharing service

Considering the log fragment in Figure 6.16, the attribute search process may discover a behavioral norm model, and the corresponding norm, depicted as:[2]:

⟨`self.uploadPhoto, self.sharePhoto, friend.viewPhoto, friend.commentPhoto`⟩

with attributes `context` and `action` representing the event operation on a common target attribute `id`. `time` and `extra` are considered to have no discernible effect on behavior. Thus, the log sub-sequence

```
2013-11-04 16:55:21   frank  self    uploadPhoto   img23   Holidays 2013
2013-11-04 17:01:03   frank  self    share         img23   lucy
2013-11-04 17:04:29   lucy   friend  viewPhoto     img23   -
2013-11-04 17:05:19   lucy   friend  comment       img23   I wish, I was there...
```

is a valid instantiation of the above norm, while the sub-sequence

```
2013-11-04 16:55:21   frank  self    uploadPhoto   img23   Holidays 2013
2013-11-04 17:01:03   frank  self    share         img23   lucy
2013-11-04 17:04:29   lucy   friend  viewPhoto     img23   -
2013-11-04 17:05:19   lucy   friend  comment       img24   I wish, I was there...
```

is not a valid instantiation of the norm as it does not involve a common photograph `id`.

Figure 6.17 depicts likely behavioral norms that might be discovered if given a complete provider log for Frank. The first norm describes the behavior that can be observed from Figure 6.16. The other norms represent additional kinds of typical normal behavior, such as Frank viewing photos shared by other users, or connecting with friends.

These discovered norms provide an insight into the behavior of the provider. Frank and his community usage patterns and configuration, such as privacy settings, are reflected in these norms. For example, Frank uses the service's default

[2]For improved readability, we depict norms as representative sequences of operations.

```
⟨self.uploadPhoto, self.sharePhoto, friend.viewPhoto, friend.commentPhoto⟩
⟨friend.uploadPhoto, friend.sharePhoto, self.viewPhoto, self.commentPhoto⟩
⟨friend.uploadPhoto, self.viewPhoto, self.commentPhoto⟩
⟨other.requestConnect, self.acceptConnect⟩
⟨self.requestConnect, other.acceptConnect⟩
```

Figure 6.17: Norms for user's collaboration with photo hosting service provider

privacy policy that considers newly uploaded photos as private. This requires him to explicitly share every photo before it is viewed by other users. Some of Frank's friends have a similar configuration, and this is reflected in the second norm. Other friends configured their account differently to make all of their uploaded photos visible to their friends or public, by default. This behavior is captured in the third norm, which lacks an explicit sharing operation.

6.7.3 Provider anomalies

Assume that Frank's photo hosting service wishes to attract additional traffic and increase the amount of content that is available to their users. To do this, they decide to change their default application behavior. The change is to make all new content visible to the user's friends by default. Users can still configure the policy explicitly in order to override the default behavior. Unaware of the new default setting, Frank continues to use the service and uploads new images. Frank's friends may now see the image instantly, without Frank's explicit action to share. This change is made only to the default behavior of the application. It does not modify application's terms of use nor the privacy policy. Frank still has the right to restrict his content, configure his policy differently, or remove any of his content. While this provider change may be done entirely legally, it has a negative effect on Frank's use of the application. This situation is somewhat similar to the phenomenon of the "dark side of the code" discussed in Chapter 2, though, the security gap exists between provider's behavior expected by the consumer and the actual behavior.

Frank's set of norms may be used to detect this application change. His service provider, after the change, will start generating logs that cannot be matched to the norms in Figure 6.17. This unrecognized activity may be considered an anomaly and alert Frank to investigate the change. Performing norm discovery on the new log can reveal that a new norm has emerged.

```
⟨self.uploadPhoto, friend.viewPhoto, friend.commentPhoto⟩
```

```
PRINT SERVICE (provider=print)                HOSTING SERVICE (provider=host)
time     user  context action      id         time     user  context action        id
19:31:05 frank self    newOrder                                                    
                                              19:31:19 frank prtsvc  listPhotos    
                                              19:31:20 frank prtsvc  getThumbnail img01
                                              19:31:20 frank prtsvc  getThumbnail img02
                                              ...                                 
                                              19:31:21 frank prtsvc  getThumbnail img08
19:33:41 frank self    select      img03                                            
19:33:52 frank self    select      img07                                            
                                              19:34:06 frank prtsvc  getPhoto  img03
                                              19:34:08 frank prtsvc  getPhoto  img07
19:36:02 frank self    submitOrder                                                 
```

Figure 6.18: A log of two collaborating systems

This anomaly is specific to Frank's interaction with the service. For other users, such as those whose photos are intentionally shared with others by default, the change has no impact. For such users, the above norm would already be considered an acceptable norm (based on the analysis of their logs).

6.7.4 Anomalies across multiple collaborating providers

Continuing the example, Frank uses an additional service provider: an on-line photograph printing service. Using this service he can order prints for his photographs on-line and have them delivered to his friends and family. The service is integrated with the photograph hosting provider. This is convenient for Frank as he can give the printing site permission to access his photographs and order prints without the need to re-upload. The access delegation can be done using a standard protocol such as OAuth [84]. In a typical scenario, Frank accesses the printing service, and selects his hosting service as the location of images. The printing service accesses his account and downloads photograph miniatures. Frank selects the photographs that he wants to be printed and the printing service, with its delegated authority from the photograph sharing service, downloads the full size image files for each of them.

The logs (visible to Frank) from both providers for such a scenario are presented at Listing 6.18. Log events now originate from two different service providers and this is distinguished by a new event attribute `provider` in the logs. In addition, events for actions performed on behalf of Frank by the printing service provider have a `context` attribute value `prtsvc` in the hosting provider log. Frank has given the printing service a permission to access his photos. While

short-lived permission delegations are possible in schemes such as OAuth, many providers offer long-lived *offline* permissions, which are often requested by the third-party providers [85], irrespective of the dangers. The expected behavior is that the service will only access the photos when Frank places a print order. Technically however, there is no such restriction and the print service may access the photos at any time. Frank can only trust that this service provider will behave properly. Analyzing the hosting service log in isolation, the following norm may be discovered.

⟨`prtsvc.listPhotos, prtsvc.getThumbnail, prtsvc.getPhoto`⟩

This norm represents a typical way in which a print service accesses user photographs when interacting with the hosting service. With its delegated permission from Frank, the printing service could decide to download all of Frank's photos in the background without interaction with Frank. This activity will generate a log in the hosting service. Based on the behavioral norm above, however, this activity can be regarded as a normal behavior.

Building the behavioral norms from the individual printer service log is insufficient to fully capture the interaction between consumer and the two providers. The norms should be discovered from a single log that aggregates the events from both service providers. In this case, log operations are characterized in terms of three attributes: `provider.context.action` with a sample norm

```
⟨print.self.newOrder, host.prtsvc.listPhotos,
 host.prtsvc.getThumbnail, print.self.select,
 host.prtsvc.getPhoto, print.self.completeOrder⟩
```

This norm captures an aggregated behavior of all of the parties collaborating together. Any activity of printing service unrelated to Frank's print ordering will be considered abnormal, as it will not match the norm.

6.7.5 Discussion

Consumer security is impacted by the provider services with which it directly or indirectly interacts. Individually, providers may have different motivations in providing service and the security mechanisms available to the consumer to control interaction tend to be weak. For example, service providers often provide only coarse-grained access controls to their consumers. When multiple applications

need to collaborate, they may be given more access than is actually required. We argue that anomaly detection style techniques can be used by a consumer to monitor interactions with providers. The challenge is to formulate a sufficiently precise model of normal interaction and we propose that consumers mine their provider logs to build profiles of past, presumably acceptable, behavior.

Conventional anomaly detection is routinely used to help protect a provider from malicious consumers; we have considered using anomaly detection to protect a consumer from multiple, possibly collaborating, providers. A single consumer transaction may span multiple providers interacting with each other and the consumer. Prescribing rules for each of the providers separately is not sufficient. As seen in Section 6.7.4, an anomaly may not manifest itself when only single provider-centric rules are considered. The anomaly may be an acceptable activity from the individual provider, but be unacceptable when considered part of a value chain.

Another difficulty in determining normal interaction is distinguishing acceptable and unacceptable provider interaction. Simply comparing provider behavior against known and precise access control rules is not sufficient. Section 6.7.3 illustrated how provider misbehavior can be subtle and within the boundaries of the contract, but is a deviation from normal/past interactions. There is a number of factors that may contribute to definition of normal, such as functionality provided by application, user specific configuration and her usage patterns. What is acceptable to one consumer may be a risk to another. All of these factors may change over time in unpredictable ways and have unexpected impact on the consumer.

6.8 Conclusion

Behavioral norms provide a generic framework to model repeating patterns of behavior inferred from system logs. The model is based on equivalence relations that partition the log into strands and group strands together into norms. We have presented, that specifying different types of equivalence may result in a different view of system operation and discovering potentially new and unexpected runtime characteristics of the system. In addition, considering the combined view of multiple collaborating systems may allow the discovery of common patterns of their operation and be potentially useful in identifying anomalies. We demonstrated that the norm model configurations can be discovered automatically and

evaluated it experimentally. The experiments suggest that the search mechanism can be quite efficient in identifying log event attributes that could be used to build norm model equivalence relations.

While this dissertation focused mainly on applying behavioral norms for the purpose of anomaly detection, the model may have other security applications. Observing system norms, changes in its number and structure over time may help to identify security problems. Developing theories about the emergent norms and their security applications is a topic for future research. In [22] we discuss how norms can be interpreted in the context of various security properties of the system.

Chapter 7

Runtime Verification of Java Applications

7.1 Introduction

In previous chapters, we described the model of behavioral norms and some of its interpretations. This chapter includes an in-depth discussion on how the model could be applied to application behavior at the level of Java method calls and used to detect anomalies that arise from vulnerabilities caused by the dark side of the code. Based on the discussion in previous chapters, we introduce an interpretation of scope and trace equivalence in the context of Java applications. We also introduce a runtime verification algorithm and discuss how it can be applied to identifying and preventing exploitation of the vulnerabilities during application execution. The operation of the algorithm is demonstrated through various types of anomalous execution traces. Finally, we present the strategy for algorithm implementation and integration of runtime verification with Java applications. Without loss of generality, the discussion in this chapter is focused around, and illustrated with, the Java microblog application introduced in Chapter 2.

7.2 Scope

In Section 2.4 on page 20, we presented two traces of application execution. The first trace captured the execution of the microblog application in response

to posting a message. The second trace resulted from an attack exploiting a public getter vulnerability that allowed the change of the current user recorded in the application's session. Figure 7.1 depicts both traces using `Caller: Class.method` format. As the message contains no URLs, the portion of logic

```
ParamsInt: PostAction.setText
Dispatcher: PostAction.execute
PostAction: PostAction.getUser
PostAction: Message.setAuthor
PostAction: TextUtils.getUrls
PostAction: DAO.add
PostAction: PostAction.return
```

```
ParamsInt: PostAction.setText
ParamsInt: PostAction.getUser
ParamsInt: User.setId
Dispatcher: PostAction.execute
PostAction: PostAction.getUser
PostAction: Message.setAuthor
PostAction: TextUtils.getUrls
PostAction: DAO.add
PostAction: PostAction.return
```

Figure 7.1: A trace fragment of message posting and user impersonation

related to creating website snapshot is not executed. The listing does not include every Java method invocation. The full trace would typically contain many thousands of events for the action execution, including the methods that belong to the application, dependent libraries, application server and Java standard library. Instead, the trace is scoped to calls of methods in classes from the application's Java package, that is `net.micro` and calls made by classes in that package. For example, the call `Dispatcher: PostAction.setText`, made by Struts to set the message text in the application's action class, is in scope because the `PostAction` class is in the application's package. The call, `PostAction: TextUtils.getUrls`, by the application to `textutils` library to identify URLs, is in scope because the caller class is in the package. Many other calls meet both criteria. In addition, only certain attributes (`caller`, `class` and `method`) are included in the trace. Other values, such as the full stack trace or heap status, while available at the method execution time, were considered out of scope.

Selecting the correct scope is essential in order to identify unexpected code execution. For example, the events related to the execution that results from exploiting the public getter vulnerability (discussed in Section 2.3.2, page 18) are included in the trace presented at the right-hand side of Figure 7.1. It includes two additional events, that are missing from the trace on the left-hand side.

However, the second vulnerability of the microblog application, that is, accessing

arbitrary files for URLs included in a message text(Section 2.3.1.3, page 17), can not be observed through traces at this scope. For example, the trace for posting a message with a single `http` link and a single `file` link are indistinguishable at the scope that only includes application methods. A different selection of the scope makes this vulnerability more apparent. For example, the scope that includes methods of the WebUtils library and all Java permission checks is shown in Figure 7.2. At this scope, the traces reflect the difference in two execution paths of the application, that is, opening an HTTP URL and a file URL.

```
WebUtils: URL.init
WebUtils: URL.openConnection
WebUtils: URLConnection.getInputStream
HttpURLConnection: URL.GET
HttpURLConnection: Socket.connect
HttpURLConnection: Socket.resolve
HttpURLConnection: Socket.connect
WebUtils: RenderEngine.render
WebUtils: WebUtils.return
```

```
WebUtils: URL.init
WebUtils: URL.openConnection
WebUtils: URLConnection.getInputStream
FileURLConnection: File.read


WebUtils: RenderEngine.render
WebUtils: WebUtils.return
```

Figure 7.2: Traces for posting message with `http` and `file` URLs, lower abstraction scope

7.3 Modeling Trace Equivalence

Figure 7.3 depicts sample strands from the execution of `PostAction` code. The strands originate from the log of method calls, partitioned to separate calls that are executed for different application's transactions that process HTTP requests. The partitioning is done with the event equivalence relation based on the common transaction identifier of HTTP requests. We will focus on the strands scoped to `class` and `method` attributes. Trace 1 captures the posting of a message without a URL in its text, Traces 2, 3 and 4 posting of messages that include one, two and three URL links respectively. Traces 5-8 capture posting of messages when snapshots for some of the URLs can not be taken, for example due to an incorrect URL or the web server being not accessible at the time. These strands capture just a small portion of possible strands within the scope, and many more execution paths are possible with different numbers of URL links, various error conditions, and so forth. Because the strands are scoped to the class/method name only, they represent a number of instances that differ by attributes which are not in the scope. For example, there could have been a number of strands

for posting a message without a URL in the log of system execution, but there is only a single sequence that represents them in the selected scope.

Trace 1	**Trace 2**	**Trace 3**	**Trace 4**
PostAction.setText	PostAction.setText	PostAction.setText	PostAction.setText
PostAction.execute	PostAction.execute	PostAction.execute	PostAction.execute
PostAction.getUser	PostAction.getUser	PostAction.getUser	PostAction.getUser
Message.setAuthor	Message.setAuthor	Message.setAuthor	Message.setAuthor
TextUtils.getUrls	TextUtils.getUrls	TextUtils.getUrls	TextUtils.getUrls
DAO.add	Message.addLink	Message.addLink	Message.addLink
PostAction.return	WebUtils.snapshot	WebUtils.snapshot	WebUtils.snapshot
	Message.addImage	Message.addImage	Message.addImage
	DAO.add	Message.addLink	Message.addLink
	PostAction.return	WebUtils.snapshot	WebUtils.snapshot
		Message.addImage	Message.addImage
		DAO.add	Message.addLink
		PostAction.return	WebUtils.snapshot
			Message.addImage
			DAO.add
			PostAction.return

Trace 5	**Trace 6**	**Trace 7**	**Trace 8**
PostAction.setText	PostAction.setText	PostAction.setText	PostAction.setText
PostAction.execute	PostAction.execute	PostAction.execute	PostAction.execute
PostAction.getUser	PostAction.getUser	PostAction.getUser	PostAction.getUser
Message.setAuthor	Message.setAuthor	Message.setAuthor	Message.setAuthor
TextUtils.getUrls	TextUtils.getUrls	TextUtils.getUrls	TextUtils.getUrls
Message.addLink	Message.addLink	Message.addLink	Message.addLink
WebUtils.snapshot	WebUtils.snapshot	WebUtils.snapshot	WebUtils.snapshot
Message.addImage	Message.addLink	Message.addLink	Message.addLink
Message.addLink	WebUtils.snapshot	WebUtils.snapshot	WebUtils.snapshot
WebUtils.snapshot	Message.addImage	DAO.add	Message.addLink
DAO.add	DAO.add	PostAction.return	WebUtils.snapshot
PostAction.return	PostAction.return		DAO.add
			PostAction.return

Figure 7.3: Sample strands of microblog application

7.3.1 Approximating norms with n-grams

A *norm* is defined in Section 5.5, page 52, as a set of equivalent traces. It is practical to use an approximate matching that will allow strands with small structural differences, such as repetition of the same sub-sequence, to be considered equivalent. Section 6.3 discusses n-gram interpretation of the trace equivalence relation. A set of n-grams can act as a norm and represent a number of equivalent strands. Figure 7.4 presents the set of n-grams[1] corresponding with traces from Figure 7.3.

[1]For improved readability, we omit the set { } and sequence ⟨⟩ parentheses that would normally be used around sets and n-grams .

Set 1 (Trace 1)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, DAO.add
DAO.add, PostAction.return
```

Set 2 (Trace 2)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, Message.addLink
Message.addLink, WebUtils.snapshot
WebUtils.snapshot, Message.addImage
Message.addImage, DAO.add
DAO.add, PostAction.return
```

Set 3 (Trace 3 & 4)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, Message.addLink
Message.addLink, WebUtils.snapshot
WebUtils.snapshot, Message.addImage
Message.addImage, Message.addLink
Message.addImage, DAO.add
DAO.add, PostAction.return
```

Set 4 (Trace 5)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, Message.addLink
Message.addLink, WebUtils.snapshot
WebUtils.snapshot, Message.addImage
Message.addImage, Message.addLink
WebUtils.snapshot, DAO.add
DAO.add, PostAction.return
```

Set 5 (Trace 6)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, Message.addLink
Message.addLink, WebUtils.snapshot
WebUtils.snapshot, Message.addImage
Message.addImage, DAO.add
DAO.add, PostAction.return
```

Set 6 (Trace 7 & 8)
```
PostAction.setText, PostAction.execute
PostAction.execute, PostAction.getUser
PostAction.getUser, Message.setAuthor
Message.setAuthor, TextUtils.getUrls
TextUtils.getUrls, Message.addLink
Message.addLink, WebUtils.snapshot
WebUtils.snapshot, DAO.add
DAO.add, PostAction.return
```

Figure 7.4: Sample bi-gram sets for strands in Figure 7.3

7.3.2 Groups and arrangements

Software execution strands typically consist of a number of common subsequences that appear in a number of strands. For example, the strands of `PostAction` execution often include the same opening and closing sequence of events. Consequently, the n-gram sets built from the strands contain a number of common n-grams which appear in many sets. When comparing sets in Figure 7.4, it could be observed that they mostly include common subsets of n-grams and differ only by a few n-grams. In this section we describe a different representation of sets of n-grams that reduces the redundancy caused by common repeating groups of n-grams.

A *group* defined over a collection of sets is a common subset of elements that always appear together in the sets in a collection. For example, in the collection of sets

$$\{a, b, c\}, \{a, b\}, \{a, b, c, d\}, \{e, f\}, \{e, f, g\},$$

the following groups can be defined

$$\{a, b\}, \{e, f\}, \{c\}, \{d\}, \{g\}.$$

The group $\{a, b\}$ means that every set containing element a will also contain element b, and vice versa.

Sets of n-grams built from strands often contain distinct subsets of n-grams that always appear together and could be represented as groups. Groups may be a part of a longer, continuous sequence of operations, such as ⟨`TextUtils.getUrls, Message.addLink`⟩ and ⟨`Message.addLink, WebUtils.snapshot`⟩. Also, some n-grams may represent sequences that always occur in the strand, for example a common beginning (⟨`PostAction.setText, PostAction.execute`⟩) and ending (⟨`DAO.add, PostAction.return`⟩). Table 7.1 depicts groups for the n-gram sets from Figure 7.4. Note that if some n-gram does not always appear with some other n-grams it becomes a singleton group.

Grouping n-grams allows for more compact representation of n-gram sets as *arrangements* of groups. For example, six n-gram sets from Figure 7.4 may be represented as the six-group arrangements in Table 7.2.

7.4 Runtime Verification

The model of groups and arrangements is used to create a reference profile of application behavior. That profile is used to verify the application activity at runtime. As described in Section 7.3, the strands used to build the profile originate from the log of method calls, partitioned in order to separate calls that are executed for different application's transactions that process HTTP requests. Therefore, the runtime verification mechanism should be able to distinguish application operation in different transactions. Verifying that a complete strand matches a reference profile created using a model of groups and arrangements is

Group	n-grams
Group 1	{⟨PostAction.setText, PostAction.execute⟩, ⟨PostAction.execute, PostAction.getUser⟩, ⟨PostAction.getUser, Message.setAuthor⟩, ⟨Message.setAuthor, TextUtils.getUrls⟩, ⟨DAO.add, PostAction.return⟩}
Group 2	{⟨TextUtils.getUrls, DAO.add⟩}
Group 3	{⟨TextUtils.getUrls, Message.addLink⟩, ⟨Message.addLink, WebUtils.snapshot⟩}
Group 4	{⟨WebUtils.snapshot, Message.addImage⟩}
Group 5	{⟨WebUtils.snapshot, Message.addLink⟩}
Group 6	{⟨WebUtils.snapshot, DAO.add⟩}
Group 7	{⟨Message.addImage, Message.addLink⟩}
Group 8	{⟨Message.addImage, Dao.add⟩}

Table 7.1: N-gram groups

Arrangement	Groups	Group	Arrangements
Arrangement 1	$\{1, 2\}$	Group 1	$\{1, 2, 3, 4, 5, 6\}$
Arrangement 2	$\{1, 3, 4, 8\}$	Group 1	$\{1, 2, 3, 4, 5, 6\}$
Arrangement 3	$\{1, 3, 4, 7, 8\}$	Group 3	$\{2, 3, 4, 5, 6\}$
Arrangement 4	$\{1, 3, 4, 6, 7\}$	Group 4	$\{2, 3, 4, 5\}$
Arrangement 5	$\{1, 3, 4, 5, 6\}$	Group 5	$\{5, 6\}$
Arrangement 6	$\{1, 3, 5, 6\}$	Group 6	$\{4, 5, 6\}$
		Group 7	$\{3, 4\}$
		Group 8	$\{2, 3\}$

Table 7.2: N-gram group arrangements

trivial. It requires computing n-grams for the strand and checking if the same set is defined by any of the arrangements in the profile. However, it can not be used for verifying software execution at runtime. The runtime verification should detect anomalies for every observed event, that is, in most cases where only some portion of an execution trace is available. In this section, we discuss the algorithm that allows online anomaly detection against reference profiles generated according to the behavioral norm model.

7.4.1 Verification algorithm

Given a sequence of past events in the transaction that lead to the current state of the transaction, the algorithm in Figure 7.5 checks whether processing the next event in this current state can result in anomalies. It relies on the reference profile, generated according to groups and arrangements model defined in terms of three variables: a set of `profile.events`, a mapping between n-grams and corresponding groups (`profile.groups`), and a mapping between groups and arrangements in which they appear (`profile.arrangements`). The current state of a transaction identified by `id` is captured using three variables: `state[id].ngram`, which captures recent events, an initially empty set of `state[id].groups` already observed in the transaction and, a set of possible `state[id].arrangements` for the transaction, initially equal to `profile.arrangements`.

```
global variables: profile, state
function check(event) {
  id = transId(event)
  if (event ∉ profile.events) return false // unknown event
  push(state[id].ngram, event)
  if (state[id].ngram[0] == null) return null // too short
  group = profile.groups[ngram]
  if (group == ∅) return false // unkown n-gram
  if (group ∈ state[id].groups) return true // already known group
  state[id].groups = state[id].groups ∪ {group} // record group
  state[id].arrangements =
    (state[id].arrangements ∩ profile.arrangements[group])
  return (state[id].arrangements ≠ ∅) // is arrangement acceptable?
}
```

Figure 7.5: Pseudo-code of runtime verification algorithm

Operator `[]` references mapping element by index and function `push` appends an element to the end of an array, removing elements from the beginning, if necessary. The `transId` function returns a unique identifier of the transaction that the event belongs to. It is the same identifier that was used to partition application log into strands that were used to generate the reference profile. This, for example, could be the value of certain event attributes.

The algorithm first checks if the event is contained in the profile. Then, provided that enough events were captured, it checks whether there is a matching n-gram for the event. It then checks if the group for the n-gram was already observed in the current transaction and, if it was, it returns a positive result. If the group was not yet observed, the algorithm checks whether the arrangements for the group intersect with possible arrangements for the groups observed so far for the transaction. This means that for every event, the algorithm performs only very simple operations such adding elements to sets, set intersection and set membership check. The only state variables that have to be kept for the transaction are: an array of `n` last events for the n-gram, a growing set of observed groups and a shrinking set of possible arrangements. The implementation of the algorithm in Java uses integers to represent the events and a hash table to map the events to the corresponding integer. Also, groups and arrangements, are implemented as collections of integers, so all set operations performed by the algorithm are made using simple integer comparisons.

7.4.2 Algorithm examples and discussion

Table 7.3 depicts the state in subsequent invocations of the runtime verification algorithm for an anomaly-free application execution. The columns represent: event position in a trace, event processed by the algorithm and fields of the `state[id]` variable after processing of each event. The numbers in `groups` and `arrangements` columns refer the identifiers of groups and arrangements in tables 7.1 and 7.2.

For the first event, the algorithm verifies that the event is included in the profile, appends it to the `ngram` array and returns, as the n-gram is not yet complete. For the second event, the `ngram` state variable becomes fully set and the algorithm identifies the group for the n-gram (1, underlined) and corresponding arrangements. The state at this point means that one group (1) was observed and possible arrangements include {1, 2, 3, 4, 5, 6}. The value of the `groups`

#	event	ngram	groups	arrangements
1	PostAction.setText	⟨null, PostAction.setText⟩	{}	{1, 2, 3, 4, 5, 6}
2	PostAction.execute	⟨PostAction.setText, PostAction.execute⟩	{1}	{1, 2, 3, 4, 5, 6}
3	PostAction.getUser	⟨PostAction.execute, PostAction.getUser⟩	{1}	{1, 2, 3, 4, 5, 6}
4	Message.setAuthor	⟨PostAction.getUser, Message.setAuthor⟩	{1}	{1, 2, 3, 4, 5, 6}
5	TextUtils.getUrls	⟨Message.setAuthor, TextUtils.getUrls⟩	{1}	{1, 2, 3, 4, 5, 6}
6	Message.addLink	⟨TextUtils.getUrls, Message.addLink⟩	{1, 3}	{2, 3, 4, 5, 6}
7	WebUtils.snapshot	⟨Message.addLink, WebUtils.snapshot⟩	{1, 3}	{2, 3, 4, 5, 6}
8	Message.addImage	⟨WebUtils.snapshot, Message.addImage⟩	{1, 3, 4}	{2, 3, 4, 5}
9	DAO.add	⟨Message.addImage, DAO.add⟩	{1, 3, 4, 8}	{2, 3}
10	PostAction.return	⟨DAO.add, PostAction.return⟩	{1, 3, 4, 8}	{2, 3}

Table 7.3: Runtime verification: anomaly-free sequence

remains unchanged until the event number 6 when a new group (3, underlined) is identified. The `arrangements` variable is changed to reflect that the group is included in a different set of arrangements than in the current state and becomes an intersection of both sets. Similar operations continue with more groups being added and, consequently, fewer possible arrangements left in the state until the end of execution of the strand.

Table 7.4 depicts an execution with where an unknown method is executed. This may, for example, result from exploiting an unprotected setter method in the common `APIAction` class. This case is not covered in our discussion in Chapter 2; vulnerabilities that may arise from method getters and setters are discussed in detail in Chapter 9. The event `APIAction.setUser` is not included in `profile.events` and the algorithm returns `false` indicating an anomaly. Note that depending on implementation of the runtime verification mechanism, the execution of the current thread may be interrupted at this point, that is before invoking the anomalous method call. To make the discussion more comprehensive, we assume that execution is not interrupted and the thread continues to execute. It is worth noting that the value of the `state.ngram` variable

#	event	state: ngram	groups	arrangements
1	PostAction.setText	⟨null, PostAction.setText⟩	{}	{1, 2, 3, 4, 5, 6}
2	**APIAction.setUser**	⟨null, PostAction.setText⟩	{}	{1, 2, 3, 4, 5, 6}
3	PostAction.execute	⟨PostAction.setText, PostAction.execute⟩	{}	{1, 2, 3, 4, 5, 6}
4	PostAction.getUser	⟨PostAction.execute, PostAction.getUser⟩	{1}	{1, 2, 3, 4, 5, 6}
5	Message.setAuthor	⟨PostAction.getUser, Message.setAuthor⟩	{1}	{1, 2, 3, 4, 5, 6}
6	TextUtils.getUrls	⟨Message.setAuthor, TextUtils.getUrls⟩	{1}	{1, 2, 3, 4, 5, 6}
7	DAO.add	⟨TextUtils.getUrls, DAO.add⟩	{1, 2}	{1}
8	PostAction.return	⟨DAO.add, PostAction.return⟩	{1, 2}	{1}

Table 7.4: Runtime verification: anomalous sequence with an unknown event

does not change, as the algorithm returns immediately after recognizing an unknown event. This is, however, an arbitrary implementation decision. The algorithm may be implemented differently in order to append the anomalous event to the `state.ngram` variable. In this case, the anomaly for event 2 would propagate to the event 3, which would also be considered anomalous as the n-gram ⟨`PostAction.setUser, PostAction.execute`⟩ is not included in the profile.

Table 7.5 depicts an execution that results from the attack discussed above in Section 7.2, in which application session is modified due to the exposure of the public getter `PostAction.getUser`. The attack involves executing application meth-

#	event	ngram	groups	arrangements
1	`PostAction.setText`	⟨`null, PostAction.setText`⟩	{}	{1, 2, 3, 4, 5, 6}
2	`PostAction.getUser`	⟨**`PostAction.setText, PostAction.getUser`**⟩	{}	{1, 2, 3, 4, 5, 6}
3	**`User.setId`**	⟨`PostAction.getUser, PostAction.getUser`⟩	{}	{1, 2, 3, 4, 5, 6}
4	`PostAction.execute`	⟨**`PostAction.getUser, PostAction.execute`**⟩	{}	{1, 2, 3, 4, 5, 6}
5	`PostAction.getUser`	⟨`PostAction.execute, PostAction.getUser`⟩	{1}	{1, 2, 3, 4, 5, 6}
6	`Message.setAuthor`	⟨`PostAction.getUser, Message.setAuthor`⟩	{1}	{1, 2, 3, 4, 5, 6}
7	`TextUtils.getUrls`	⟨`Message.setAuthor, TextUtils.getUrls`⟩	{1}	{1, 2, 3, 4, 5, 6}
8	`DAO.add`	⟨`TextUtils.getUrls, DAO.add`⟩	{1, 2}	{1}
9	`PostAction.return`	⟨`DAO.add, PostAction.return`⟩	{1, 2}	{1}

Table 7.5: Runtime verification: anomalous sequence with an unknown n-gram

ods `PostAction.getUser` and `User.setId`. The first method is included in the profile so, it passes the first check made by the algorithm. However, because it is executed directly after `PostAction.setText` and the corresponding n-gram is not part of the profile, the algorithm flags this execution as an anomaly. Subsequently, if we assume that the execution is not interrupted, the execution of `User.setId` is considered anomalous due to an unknown event.

In the examples discussed in this chapter, the scope includes only `class` and `method` attributes. In practice, however, it would be more appropriate to include the `caller` attribute, as it is done during experiments discussed in Chapter 8. At this scope, the profile would include the `[PostAction: PostAction.getUser]` event, as this is the way the method is called. During the attack, however, the event at position 2 would be [**`ParamsInt:`** `PostAction.getUser]` because the method is called by the Struts parameters interceptor. In this case, the method execution would be still considered anomalous but for a different reason, that is due to an unknown event. Note that, the simple check on whether an event is contained in the profile, be-

fore validating the n-gram, may be quite effective in identifying anomalies at this level of abstraction. This is different from most techniques discussed in Chapter 3 which focused on common, general purpose operating system calls, such as `open` or `read`. At that level of abstraction, it is not practical to verify system calls against the set of system calls that are included in the trace as, most likely, this would be all common calls made by every application. Also, the inclusion of the caller class makes the model much more resistant to mimicry attacks. Typically, the attacker will be constrained to inject method calls using some entry point. The mechanism of injection is reflected in the trace, making it harder to craft any malicious sequence. For example, while Struts `ParametersInterceptor` allows calling public getter methods, each such call is reflected in the trace with the interceptor as a caller class. But, if the profile was built using the scope including the caller class, it would prescribe that the only method that can be invoked by the interceptor is `PostAction.setText`.

Table 7.1 depicts an execution that results in an illegal arrangement. The anomaly is observed at event 11. The n-gram for this event is part of Group 5, which is included in arrangements $\{5, 6\}$. However, based on the thread execution up to this event, the only legal arrangements are $\{3, 4\}$. As there are no common arrangements between the two sets, the algorithm considers the event anomalous due to an impossible arrangement. Note that the sequence represents a normal execution of an application so this case represents a false positive. The profile we use in the example is built using only 8 sample traces so it is expected that it does not cover the entire normal behavior of the application.

#	event	ngram	groups	arrangements
1	PostAction.setText	⟨null, PostAction.setText⟩	{}	{1, 2, 3, 4, 5, 6}
2	PostAction.execute	⟨PostAction.setText, PostAction.execute⟩	{1}	{1, 2, 3, 4, 5, 6}
3	PostAction.getUser	⟨PostAction.execute, PostAction.getUser⟩	{1}	{1, 2, 3, 4, 5, 6}
4	Message.setAuthor	⟨PostAction.getUser, Message.setAuthor⟩	{1}	{1, 2, 3, 4, 5, 6}
5	TextUtils.getUrls	⟨Message.setAuthor, TextUtils.getUrls⟩	{1}	{1, 2, 3, 4, 5, 6}
6	Message.addLink	⟨TextUtils.getUrls, Message.addLink⟩	{1, 3}	{2, 3, 4, 5, 6}
7	WebUtils.snapshot	⟨Message.addLink, WebUtils.snapshot⟩	{1, 3}	{2, 3, 4, 5, 6}
8	Message.addImage	⟨WebUtils.snapshot, Message.addImage⟩	{1, 3, 4}	{2, 3, 4, 5}
9	Message.addLink	⟨Message.addImage, Message.addLink⟩	{1, 3, 4, 7}	{3, 4}
10	WebUtils.snapshot	⟨Message.addLink, WebUtils.snapshot⟩	{1, 3, 4, 7}	{3, 4}
11	Message.addLink	⟨WebUtils.snapshot, Message.addLink⟩	{1, 3, 4, 5, 7}	**{}**
12	WebUtils.snapshot	⟨Message.addLink, WebUtils.snapshot⟩	{1, 3}	**{}**
13	DAO.add	⟨TextUtils.getUrls, DAO.add⟩	{1, 2}	**{}**
14	PostAction.return	⟨DAO.add, PostAction.return⟩	{1, 3, 4, 8}	**{}**

Table 7.6: Runtime verification: anomalous sequence with an illegal arrangement

7.5 Anomaly Manager

While traditional logging may be useful for retrospective detection of anomalous behavior, integrating runtime verification/anomaly detection with the application provides the ability to interrupt application execution before the anomalous operations are executed. The Security Manager is a natural integration point for monitoring and controlling Java application execution. For example, as discussed in Section 7.2, an attack exploiting the URL handling vulnerability of microblog application could be identified by observing the permission checks.

It is limited, however, to the operations which result in permission checks, such as input/output operations or security related activity, such as access to cryptographic keys. In particular, it would not be possible to identify the public getter vulnerability as the relevant activity is not in the scope of the Java Security Manager. An arbitrary off-the-shelf component might not have been programmed with its own permissions. In this case, it is insufficient to rely on the Security Manager, and an additional integration point is required. It could be implemented using Java Aspects that allow intercepting invocation of methods and adding a custom code to execute before or after each call.

We have developed an Anomaly Manager that supports behavioral norm-based anomaly detection for Java applications. It is a runtime library that implements the Java Security Manager interface that traps permission checks from which it can generate a baseline/behavior reference profile, or it can monitor application execution for compliance with a previously generated behavior profile. We have also implemented a Java Aspect that intercepts method calls in the scope of interest. The basic structure of the aspect for the microblog application in the scope discussed in this chapter is depicted in Figure 7.6. The `@Before` annotation prescribes that the aspect intercepts calls to methods of classes in `net.micro`

```
1 @Aspect public class BehaviorAspect {
2   @Before("call(* net.micro..*()) || within(net.micro..*)")
3   public void intercept(JoinPoint context) {
4     Permission perm = new ExecPermission(context);
5     SecurityManager.getSecurityManager().checkPermission(perm);
6   }
7 }
```

Figure 7.6: Java Aspect for the microblog application calls

package, or any call made by classes within that package. The Aspect invokes the `intercept` method, providing the execution context that includes information such as caller, class and method. A special type of permission is created using that context information and the Security Manager is called to check it. This, in turn, activates the anomaly manager that verifies the method call in the context of the profile and the current state. In this arrangement, the manager may interrupt the execution of the current thread (for example, by throwing an exception) if an anomalous call is detected and prevent its execution.

The overall structure of the integration is depicted in Figure 7.7.

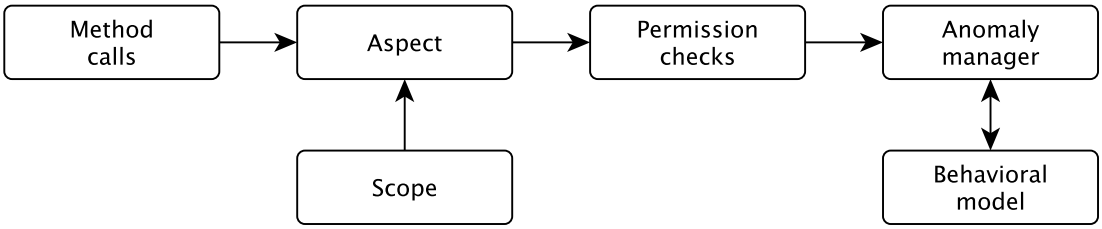


Figure 7.7: Java Anomaly Manager integration

7.6 Discussion

The n-gram interpretation of the behavioral norms model could be viewed as similar to the n-gram based models focused on detecting anomalous system calls, such as `stide` [14, 17], discussed in Chapter 3. Both approaches use n-grams to implement approximate sequence matching but there are a number of differences. The primary difference is that behavioral norms use separate sets to capture behaviorally different transactions, which makes the model more precise when compared to `sdite` in which all application activity is blended into a single set of n-grams. The increased precision may not be evident in the context of the simple examples used in this chapter that contained traces of a single API call/Struts Action. However, the full profile of the microblog application would be built using traces from a number of different API calls, such as searching, viewing and deleting messages. For illustration, the behavioral norms profile could be reduced to the profile presented in [17] by creating a union of all n-gram sets. Also, including context attributes, such as the caller class, significantly increases the precision of the model.

Systems such as `stide` often use a threshold value, such as the number of mismatches against the profile observed before the trace is considered anomalous. This may not be practical at the method call abstraction level since one or two injected method calls may be enough to successfully exploit a vulnerability. This means that the baseline behavior for the norms-based system must be very comprehensive in order reduce the number of false positives. Also, it appears that the increased precision of the model and usage of application-specific method call names, rather than general-purpose system calls, makes it possible to use much shorter n-grams and consequently reduce the size of the profile.

7.7 Conclusion

The model of behavioral norms provides a generic framework for capturing a transaction-like behavior. In this chapter we presented an interpretation of the model focused on Java method calls. The scoping of method calls may be done using various attributes of the method execution context. We focused on Java packages, caller, class and method. The trace equivalence can be implemented using an approximate sequence matching using n-grams. This allows reducing the profile size and cover typical control flow structures such as conditions and loops. Encoding the profile using the groups and arrangements representation allows the further reduction of its size and optimization of the profile for anomaly detection.

The n-gram interpretation focuses only on the attributes that represent an operation and its context, such as `class` and `caller`. The operation attributes repeat in the same form in every strand. However, the log also includes other attributes for which the value may be different in different strands, but remains unchanged within the strand. For example, in the normal operation of `PostAction`, the value returned by method `User.getUser` (such as `frank`) is always the same as the argument of the `Message.setAuthor` method. Further research should consider how such correlations of attributes can be included in the n-gram based model.

We presented an anomaly detection algorithm that allows the verification of the application execution at runtime for every individual method call within scope. We also discussed the strategy for integrating the anomaly detection with the application using the Anomaly Manager. Using Java Security manager and Aspects allows the enablement of the runtime verification in existing applications without any code changes in the application itself.

Chapter 8

Experimental Evaluation

8.1 Introduction

Our hypothesis is that unknown security vulnerabilities in software components can be identified as runtime anomalies arising from unexpected execution paths. In this chapter, we evaluate the hypothesis experimentally. In previous chapters, we focused on the microblog system and corresponding vulnerabilities at the scope of the application code. While this provided an easy to follow example for our discussion it is not appropriate for the experimental evaluation. Testing this hypothesis using a catalog of vulnerabilities hand-crafted for the purpose, may provide insight, but their design can be contrived/"cherry-picked" and is not an effective evaluation of whether the approach would work "in the wild".

We therefore decided to test the hypothesis against a well established and popular enterprise-scale software component that has a history of security vulnerabilities. The experiment still uses the microblog application, but at a different scope. Rather than analyzing the behavior of the application itself, we focused on one of the libraries it uses, that is Apache Struts. Struts is a popular and mature Model-View-Controller used by a number of enterprise applications [86].

The prior research on the subject of detecting anomalous software behavior resulting from exploiting vulnerabilities [3, 14, 17, 18] also based their experiments on the actual vulnerabilities in real-life applications. However, often just a few selected vulnerabilities were considered. For example, the anomaly detection system presented in [17] is evaluated using three vulnerabilities in `sendmail`, one in `lpr` and one in `wu-ftpd`. As we are interested in verifying how effective anomaly

detection is in identifying software vulnerabilities in practice, we decided to consider a complete set of vulnerabilities reported over a long time frame. In the following we considered the vulnerability history of twenty-six versions of Apache Struts over a five-year period, starting with version 2.3.1, released in December 2011, to version 2.3.24.1 released in May 2015. In particular, the objective is to test whether vulnerabilities reported against earlier versions of software can be identified as anomalies, while those same anomalies are not reported against later versions of the software in which the corresponding vulnerability has been remedied.

In the following sections, we describe how the experiment was set up and performed to systematically establish baseline behavior of the application, generate the profile and test it against the vulnerabilities. We also present the results of the experiment with the analysis and discussion.

8.2 Experiment Setup

Figure 8.1 outlines the key elements of the experimental setup. In order to evalu-

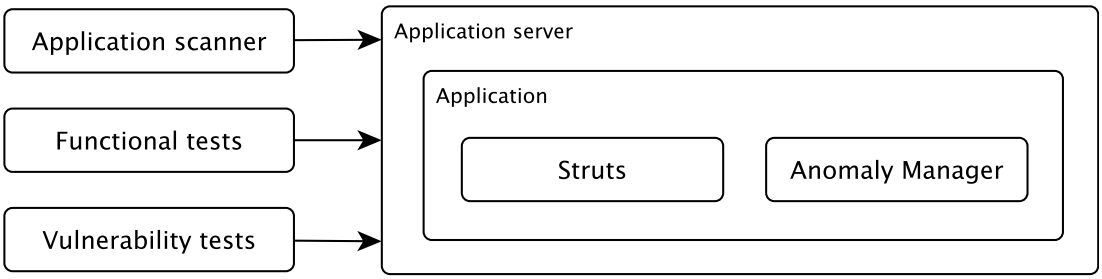


Figure 8.1: Experiment setup

ate Struts behavior in its typical environment, we developed a small Struts-based web application based on the microblog program described in Chapter 2. The application makes conventional use of Struts, with a standard configuration.

The application system is built automatically and the experiment is carried out separately for each version of Struts. Experiment characteristics, such as execution times and sizes, were comparable for the different versions of Struts. Experiments were orchestrated by Apache Maven and each iteration for a different Struts version comprised of two phases. In the first phase, a trace of the application system's execution is generated and from which the behavioral profile

is built for the given version of Struts. In the second phase, the effectiveness of anomaly checking based on the generated behavioral profile is verified.

8.3 Building Behavioral Profiles

A commercial application security scanner was run against the microblog web application. The scanner configuration was standard and not tailored in any particular way for Struts. The same scanner configuration was repeatedly used against each deployment of the application with a different version of Struts. The scanner operates in two phases. First, it crawls the application by accessing every URL, following every link to discover its structure and functionality. This phase is assisted by a set of simple automated functional tests that cover various basic features of the application such as posting a message, which the scanner intercepts to learn the initial set of URLs. This is a standard practice to improve the automatic discovery of the scanner [87]. In the second phase, the scanner interacts with the microblog web application, black-box testing an extensive collection of known vulnerabilities and misconfigurations. This scanning is considered to represent the kind of application interaction that is expected by the developer. The Java Anomaly Manager was deployed with the application and used to build/check the behavior profile in each experiment.

When describing the anomaly detection in Chapter 7, we focused on the application behavior from the perspective of application's own method calls. In our experiment, we consider the behavior of Struts, in terms of how it is used by the application. Therefore, the Anomaly Manager's Aspect was configured to intercept all method calls within the Struts packages, that is `org.apache.struts2.*`., corresponding to 473 distinct methods that are used in the context of the microblog web application. Also, 143 different caller classes are included in the profile. A single scan experiment resulted in around 9000 HTTP requests to the microblog web application URLs with a range of inputs, taking 8 minutes to complete. In monitoring the execution of the application during this scan, the Anomaly Manager generates a 237-megabyte trace containing 2.76 million Struts events, and a set of behavioral norms generated within approximately a further 5 seconds, running on a mid-range computer[1].

The behavior profile was built using request identifier attribute for the target

[1]Intel i5-4300U 1.90GHz, 8GB RAM

and permission/class name and action/method for the operation. Also, as initial experiments indicated no difference in the results with n-gram sizes up to 7 we have selected size of 3 as the most memory efficient. Despite the size of the trace, the resulting profile is quite compact. All of the captured Struts activity is abstracted to 65 behavioral norms taking 40 kilobytes of memory.

Figure 8.2 plots an example of the number of distinct norms generated, against the number of HTTP requests made by the scanner to the application deployed with Struts version 2.3.1. From the start of the scan, as the number of HTTP requests increase, the number of norms resulting from the requests rapidly increase initially, and then appear to stabilize. This graph suggests that our scan size of 8963 HTTP requests is adequate, after which no new norms are identified. Similar results were achieved for the other versions of Struts.

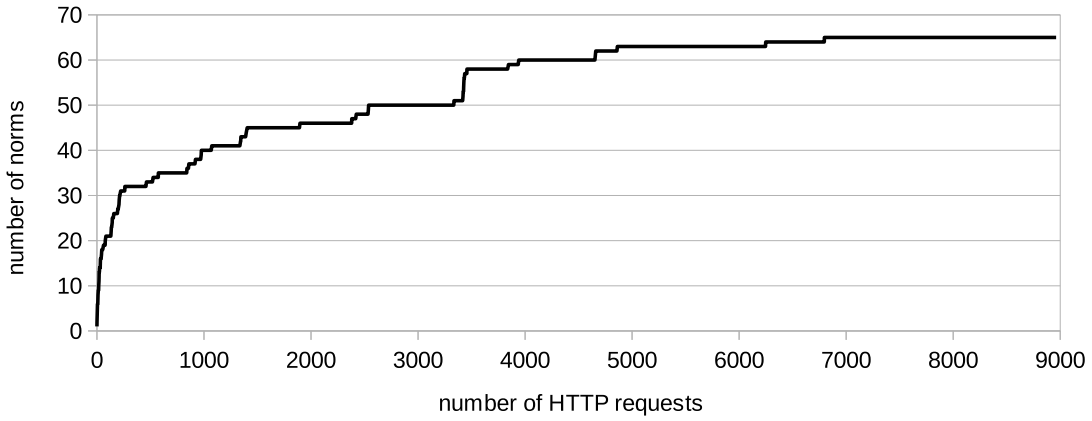


Figure 8.2: Growth in behavioral norms

Having generated a behavioral norm profile for a given version of Struts and included it with the Anomaly Manager in the application deployment, the second phase of the experiment involves testing the effectiveness of using the behavioral model to detect vulnerabilities for that version of Struts.

8.4 Vulnerability Tests

At the time of the experiment and based on the Common Vulnerabilities and Exposures (CVE) entries in the National Vulnerability Database, there were nineteen different vulnerabilities known to the general public for the twenty-six versions of Struts under study. For each vulnerability, the CVE advisory was studied alongside the vulnerable Struts code and the remediated version of the code, and

an attack vector exploiting the vulnerability was developed. Note that we do not include the vulnerability CVE-2013-6348 that applies to a development-time tool Configuration Browser Plugin [88]. This vulnerability is incorrectly attributed to Struts library in CVE records, but it is not included Struts security bulletin. Also, some vulnerabilities were reported in the time-frame of our experiment, but for earlier versions of Struts. Of the nineteen vulnerabilities, attacks for the eighteen listed in Table 8.1 were developed; we could not find enough information to reproduce the vulnerability identified in CVE-2012-4386. For each vulnerability, we implemented an automated test case which attempts to exploit the vulnerability and verify that the exploitation was successful. Appendix A provides definitions of all the test cases.

For example, CVE-2013-2115 is a vulnerability that allows a remote attacker to execute arbitrary OGNL code [36], discussed in Chapter 9, code via a crafted request. It affects Struts JSP tags for rendering URLs. In order to render a URL for a search action, including the current page's parameters, the developer implements the following code.

```
<s:url action="SearchAction" includeParams="all">
```

The tag is evaluated to `/api/search?name=Frank`. Using the tag is convenient and relieves the developer from having to manually map actions to URLs and passing parameters, thus further separating application logic from low-level details. However, the code for processing the tag suffers from a security vulnerability. An attacker may add a request parameter by including OGNL code and that code will be evaluated when processing the tag. For example, the official security advisory for this vulnerability (CVE-2013-2115) describes that an attacker may append

```
x=$[@java.lang.Runtime@getRuntime().exec('cmd')]
```

to the page parameters. The OGNL code, enclosed in `${}` is evaluated and custom Java code is executed by the application. However, the attack results in activity caused not only by an unexpected Struts execution path, but also by the injected code involved in creating a process and accessing a file binary. Rather than injecting malicious code that results in significantly different behavior, each test case attempts to inject benign code that simply sets a variable that can be subsequently checked to determine the success of the attack. Note, setting a variable is not in the scope of Struts methods, or the activity captured by AspectJ and does not get flagged as an anomaly in itself. The objective of each

test is to generate the minimal behavior needed to explore the path of the attack, but it does not engage in subsequent behavior that might be easily recognized as anomalous in its own right. In this way, the test case is intended to represent a worst-case scenario for anomaly detection.

In many cases developing the vulnerability test cases was not trivial. A number of the vulnerabilities are not clearly described in their respective advisories and sometimes the details are intentionally undisclosed. For example, CVE-2013-4310 (discussed below) is described as allowing "to bypass security constraints under certain conditions". It further explains that "more details will available later on when the patch will be widely adopted". Similarly, CVE-2015-1831 "allows remote attackers to 'compromise internal state of an application' via unspecified vectors". In order to prepare test cases for these vulnerabilities, it was necessary to understand their nature through an analysis of source code changes of Struts and experimentation.

8.4.1 Vulnerability test results

Table 8.1 contains[2] the outcome of testing each of the eighteen reported vulnerabilities against each of the twenty-six versions of Struts. Each table cell contains three outcomes. The first outcome indicates whether it was reported that the particular version of Struts was indeed affected (+), or not (−) by the vulnerability. The second outcome specifies whether the execution of the attack test case for that vulnerability was successful (+), or not (−). The third outcome specifies whether the Anomaly Manager detected anomalous behavior during execution of the test case (+), or not (−). For example, the outcome +++ means that the version had the reported vulnerability, that the attack test case successfully executed and that anomalies were detected (true positive).

Considering CVE-2013-2115, the URL tag vulnerability described above, we see from Table 8.1, that the attack test case successfully exploited this vulnerability, and was detected as an anomaly, for all versions 2.3.1 – 2.3.14.1 publicly announced to be vulnerable (+++). A closer examination of the following execution trace fragment, generated from the attack test case, identifies the anomaly as OGNL code used in rendering the URL.

[2]Presented as a table, keeping in mind Edward R. Tufte's (2004) observation that *"small non-comparative highly labeled data sets usually belong in tables"*.

CVE ID	2.3.1	2.3.4	2.3.14	2.3.14.1	2.3.14.2	2.3.15	2.3.15.1	2.3.16	2.3.16.1	2.3.16.2	2.3.16.3	2.3.20	2.3.24	2.3.24.1
CVE-2015-5209	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−−
CVE-2015-1831	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	+++	−−−	−−−
CVE-2014-7809	++−	++−	++−	++−	++−	++−	++−	++−	++−	++−	++−	−−−	−−−	−−−
CVE-2014-0116	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−+	−−+	−−+	−−+
CVE-2014-0113	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−+	−−+	−−+	−−+	−−+
CVE-2014-0112	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−
CVE-2014-0094	+++	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-4316	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-4310	++−	++−	++−	++−	++−	++−	++−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-2251	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-2248	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-2135	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-2134	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-2115	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-1966	+++	+++	+++	−++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2013-1965	+++	+++	+++	−++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2012-4387	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
CVE-2012-0393	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−

vulnerable attack successful anomalies not vulnerable attack failed no anomalies

Note that some outcomes were identical for different versions of Struts, and in the interest of space: version 2.3.4 covers 2.3.1.1, 2.3.1.2 and 2.3.3; 2.3.14 covers 2.3.4.1, 2.3.7, 2.3.8 and 2.3.12; 2.3.15 covers 2.3.14.3, and 2.3.16 covers 2.3.15.2 and 2.3.15.3; 2.3.24 covers 2.3.20.1.

Table 8.1: Attack outcomes on different versions of Struts

```
DefaultUrlHelper DefaultUrlHelper.translateAndEncode
DefaultUrlHelper DefaultUrlHelper.translateVariable
OgnlInvoke invoke
ServletUrlRenderer ComponentUrlProvider.getAnchor
ServletUrlRenderer ComponentUrlProvider.isPutInContext
```

Examining execution traces may help identify that part of the code that is responsible for the vulnerability. Indeed, a study of subsequent versions of the Struts source code that repair the vulnerability reveals that the issue was attributed to `translateVariable` that invoked OGNL processing. In the repaired code, the method was removed as unnecessary. Consequently, `translateAndEncode` method (which called `translateVariable`) was renamed to simply `encode`.

Carrying out the same test case on subsequent non-vulnerable versions 2.3.14.2 – 2.3.24.1 results in an unsuccessful attack and no anomalous behavior (−−−), which is as expected. The corresponding trace fragment for those versions shows that no OGNL code execution—the root cause of the vulnerability in previous versions—was observed in the context of URL rendering:

```
DefaultUrlHelper DefaultUrlHelper.encode
ServletUrlRenderer ComponentUrlProvider.getAnchor
ServletUrlRenderer ComponentUrlProvider.isPutInContext
```

Some tests executed with unanticipated outcomes. A surprising result −++, indicates a successful attack test case, with the anomaly detected (true positive), for Struts version 2.3.14.1 for which no vulnerability was reported in CVE-2013-1965 and CVE-2013-1966. A closer examination of these two vulnerabilities confirms that, contrary to publicly available information, version 2.3.14.1 is indeed vulnerable. In addition, the release notes of version 2.3.14.2 refer to incomplete fixes for the previous security vulnerability.

8.4.2 False negatives

In most cases, Table 8.1 reports that the anomaly detection identified successful attacks on vulnerable versions, while attempted/unsuccessful attacks on non-vulnerable versions were not identified as anomalous. The table, however, reports two vulnerabilities with a false negative result (++−), that is, a successful attack on a vulnerable version for which anomalous behavior was not observed.

One false negative arises from CVE-2014-7809: a Cross-Site Request Forgery vulnerability caused by a predictable token generated using a weak generator. It allows an attacker, knowing a previous value of a token used for CSRF prevention, to predict the value of the next token and to use it to perform a CSRF attack against an authenticated user. Although the attack is caused by a simple coding error, we argue that it does not arise from an unexpected path of code execution. A request with the token generated by the attacker results in exactly the same behavior as a legitimate request made by the user. As such, it can not be detected as an anomaly in the execution path.

Another false negative arises from CVE-2013-4310, which reports an ability to bypass security constraints. Our investigation discovered that this vulnerability is only applicable to applications that have a somewhat unusual security mechanism. As mentioned in Chapter 2, actions in Struts are the basic unit of an application's business logic and are normally mapped to specific URL paths (such as `/api/post`). Struts offers an alternative addressing through URL parameters with `action` prefix, such as `/api/other?action:post`. The vulnerability describes a scenario when a security control, implemented outside struts, based on specific URL pattern is bypassed using the alternative addressing. It could be argued that this scenario does not really describe a Struts vulnerability, but rather a faulty security control that a documented feature of Struts allows to facilitate. In our experiment, the attack exploiting CVE-2013-4310 was undetected because the microblog application included use of `action:` prefixes. Addressing actions though parameters was considered a normal behavior of the application and, when executed during the attack, did not cause anomalous behavior.

8.4.3 False positives

In two instances, the test cases result in false positives, that is, anomalous behavior is detected for an unsuccessful attempt to exploit non-vulnerable version ($--+$). This outcome is observed for CVE-2014-0114 and CVE-2014-0116 in versions where these vulnerabilities are repaired. They are variants of the problem discussed in Chapter 2 where the internal state of an application can be modified through a chain of getters and setters, and in this case, though crafted cookies. A study of the attack test cases reveals that the anomalous behavior is related to the special treatment that Struts gives to particular cookie names. The original vulnerabilities were repaired by adding a blacklist of disallowed cookie names (such as starting with `class`). Thus, processing a normal cookie results

in a behavior that is different to processing a cookie with a blacklisted name. However, in its standard configuration, the application scanner does not generate a request involving a Struts blacklisted cookie and, therefore, the generated profile of expected behavior does not include an execution path corresponding to the security processing of a blacklisted cookie. Thus, the test case, while not an attack, is flagged as an anomaly.

One might be tempted to argue that if the application scanner was explicitly configured to generate/use blacklisted cookie names in its requests, then we might no longer have a false positive. This corresponds to making adjustments to anomaly detection based on what is known about vulnerabilities. While this is perfectly normal in practice, it is contrary to the purpose of our experiment which was to evaluate the ability to identify unknown vulnerabilities/zero-day attacks. Overall, this part of the experiment indicates that all vulnerabilities that could be attributed to unintended code execution paths in Struts were successfully detected. It also shows that, with one exception of blacklisted cookies, in non-vulnerable versions, where a malicious request is properly handled by the application no anomalous behavior is reported.

8.4.4 Results interpretation

Many past studies of anomaly detection techniques [3, 14] provide summary metrics intended to represent the efficacy of the technique. For example, based on our experiment, one could have calculated how many vulnerabilities were detected in comparison with the total number of vulnerabilities; or how many false positives were observed. However, we argue that the extensive analysis of the results in the previous sections provides better means to understand the efficacy of the method and its applicability in real life systems.

We believe that although the experiment was comprehensive in comparison with prior work, providing summary metrics would be inappropriate. The values would be affected by several factors that are independent of the performance of our runtime verification technique. For instance, the experiment was based only on discovered and publicly known vulnerabilities. Any change in the set of vulnerabilities used in the experiment would have an effect on the final metric. For example, if the CSRF vulnerability CVE-2014-7809 (false negative), was not discovered, or if we used a different period, the metric would be significantly different. Also, the metric would have been different if CVE-2013-4310, a questionable vulnerability

discussed in Section 8.4.2, was not attributed to Struts. On the other hand, as we discuss in Section 8.6.2 and in more detail in Chapter 9, some CVE records cover instances of incomplete remediation of previously reported vulnerabilities. This means that what could be considered a single vulnerability, is represented in the results multiple times just because it happened to be recorded multiple times in CVE. Chapter 9 includes further discussion on the problems of using quantitative metrics in these kinds of studies and the benefits of carrying out a more qualitative style of analysis of the vulnerabilities.

8.5 False Positive Tests

On generating an expected model of behavior (for a given version of Struts) we check that the set of norms is sufficiently complete by engaging a further standard application scan and confirm that no anomalies are identified. This confirms that normal, expected behavior is properly recognized by the Anomaly Manager. However, for the purposes of the evaluation, we are interested in confirming that the anomaly detection can also discriminate between attacking behavior that exploits a vulnerability versus other behavior that executes code in the region of the vulnerability, but does not actually exploit the vulnerability. To this end, a suite of functional tests were developed to check this ability to discriminate.

For example, the URL tag vulnerability CVE-2013-2115 described in Section 8.4 involves passing a crafted value through a URL parameter. The test calls an application using an additional parameter but without OGNL code. Some vulnerabilities require more advanced test cases. For example, the attack test case for CVE-2013-2251 involves passing a crafted string through a Struts-specific request parameter (`action:`), allowing indirect action addressing. We developed two further functional test cases that check this particular functionality. The first test `ActionPrefix` uses the indirect addressing, but with a correct action name, and checks that the requested action was called. The second test `WrongActionPrefix` uses an incorrect action but checks whether the application replied with an expected and corresponding error. The purpose of these tests is to check whether the anomaly detection actually reacts to a genuine vulnerable path of execution or whether the path from the related valid functionality is flagged as an anomaly. By exercising the functionality normally (even if in an error scenario) and during the attack we can distinguish between these two cases.

Overall, we developed nineteen test cases that explore non-attacking behavior in

the region of the eighteen vulnerabilities, described in Appendix B. The outcome-score of each test is similar to the vulnerability tests and represented using two values. The first outcome value reports whether the test was successful, that is whether the tested functionality worked correctly. Note that because we have also tested an application's response to an incorrect request, a successful outcome may mean that the application correctly responded to an incorrect value, for example by presenting an error page. The second outcome value reports whether an anomaly was detected during execution of the test. Most of the test outcomes indicate a successful test with no anomalies. However, in two cases the outcome indicates a test failure with no anomalies observed. These are the test cases for indirect action addressing using `action:` prefix. The test fails for all versions from 2.3.15.2. This is because, as a response to CVE-2013-4310, this functionality was disabled.

8.6 Discussion

8.6.1 Anomaly prevention in practice

A further experiment was carried out to confirm that detected attacks could also be prevented by the Anomaly Manager. The results match those reported in Table 8.1 and we found that the attack could be interrupted before the anomalous operation occurs. We did not observe any interruption during application scanning (expected behavior) nor during the functional test run.

The execution is interrupted by throwing a security exception by the Anomaly Manager, which is consistent with the default Java Security Manager. The exception does not need to be explicitly handled by the application and, by default it only interrupts the execution. As with any other security manager, an application may attempt to catch the exception and attempt to recover or clean up. For example, it may roll back a transaction or log the anomaly with some application-specific information such as a current user or source IP address.

Anomaly detection/prevention adds a performance overhead that should be considered. Verifying an activity against the model consists of simple operations, such as hash table lookups and integer set operations. In order to integrate anomaly detection with the application, we used a Java Security Manager and Aspect Oriented Programming, as discussed in Chapter 7. These tools are rou-

tinely used for implementing security mechanisms [89] and their performance impact has been investigated [90, 91]. During the Struts experiment, the average time to process an HTTP request from the scanner to the application took 4950μs without instrumentation. With the Anomaly Manger enabled for runtime verification, the average time increased by 3.85% to 5140μs. However, the increase depends on how much of application activity is covered by runtime verification: in the experiment this was limited to the Struts library. The evaluation has been carried out using Java and instrumentation tools, such as AspectJ, designed for that platform. For other platforms, different techniques of integration should be considered. The memory requirements for the runtime verification are minimal. As described in Chapter 7, the state has to be kept for every active transaction (a worker thread in the case of Java web applications). The state contains three sets of integers: an n-gram (integer identifiers of n recent events), set of observed groups identifiers and set of possible arrangement identifiers.

In the experiment, we relied on an automated scanner in order to learn the application's expected behavior. The purpose of using this type of tool was to ensure that application is explored in an unbiased, repeatable way. However, we do not consider application security scanning as the only possible way to identify baseline behavior. Automated functional or integration test could be also a good way to exercise expected behavior of an application and can be combined with the scanner. Also, enterprise application testing tools include more advanced testing techniques, such as glass-box scanning [92], that combines dynamic testing with the source code analysis. Note that the application scanner may trigger an unexpected behavior, which if unattended to, becomes part of the norms used in runtime verification. However, this phenomenon was not observed during our experiments. Furthermore, we assume that any unexpected behavior identified during scanning would likely represent some form of an attack known/tested by the scanner, which the developer would remediate by modifying the source code. An alternative approach is automated test case generation based on simulated user behavior [54]. Applicability and efficacy of various automated methods to acquire normal behavior baseline is a subject for future research.

8.6.2 Additional insights

Part of our experimental setup involved inspecting the code bases of different versions of Struts in order to identify the vulnerable code and implement the attack tests. In carrying out this detailed code-level review we observed a num-

ber of programming phenomena across the different versions. In particular, the phenomena that some generic functionality of Struts allows for a specific execution scenario that compromised security. The otherwise harmless features, such as addressing actions, setting their parameters and evaluating expressions, when used in a particular way allowed unintended operations. For example, CVE-2014-0094, CVE-2014-0112, CVE-2014-0113, CVE-2014-0116 and CVE-2015-1831 exploit the feature that allows setting action properties through HTTP request parameters/cookies and accessing sensitive objects, such as a session or class loader.

Overall, we observed that the majority of the programming issues relate to rather simple programming errors. In particular, while a general functionality was implemented, a specific, unexpected execution pattern was not considered nor handled by the code. In some cases, such as accessing the class loader via a parameter (CVE-2014-0094) we observed that developers were surely aware that accessing properties through parameters can be a security risk [24]. Some specific parameters, such as session object, were blacklisted but others that are less obvious, such as class, were not. In other cases, when a vulnerability was identified in one part of the framework it was not immediately correlated to another part that was also vulnerable [24]. For example, the fix for CVE-2014-0094 addressed class loader manipulation through request parameters but did not provide the fix for the same attack using cookies. Our analysis of Struts vulnerabilities seems to confirm other studies [24, 26], discussed in Chapter 9, that indicate that developers tend to repeat security errors even when they are aware of a particular vulnerability.

8.7 Conclusion

Performing a comprehensive and controlled evaluation of anomaly detection with the objective of identifying contemporary software vulnerabilities is a difficult task. There is a major effort required to acquire information about the vulnerabilities and develop corresponding test cases. Also, the results should be interpreted very carefully, as there could be a number of external factors that could potentially affect the results. Often, there is only a limited number of vulnerabilities that are available for each software component and they appear for different, often non-compatible versions. Some components, even though critical for the security of the system, can not be tested directly or in isolation because,

like Struts, their conventional operation interleaves with other components.

In the case of the microblog application, our experiments demonstrate that it is possible to learn a sufficiently rich model of the application's expected use of Struts such that it can be used to detect anomalies in its subsequent use of Struts. Indeed, results indicate that all seventeen execution path-related vulnerabilities identified over twenty-six versions of Struts over five years can be effectively identified as anomalies. We limited our experiments to Struts operation in the context of the particular application.

The deeper analysis of Struts vulnerabilities suggests that they indeed may be resulting from the developer's oversights caused by the dark side of the code, discussed in Chapter 2. The vulnerabilities in the scope of our experiment typically appear in the context of other components such as the OGNL language or Java class loader. In the next chapter, we discuss the results of in-depth analysis of Struts vulnerabilities over a period of 12 years and attempt to discover what could be their root causes.

Chapter 9

Security Vulnerabilities

9.1 Dark Side of the Code Revisited

In Chapter 2, we introduced the phenomenon of the dark side of the code. Using an example, we argued that software development at a high level of abstraction, using components that abstract low-level details may lead to security vulnerabilities. These vulnerabilities result from unexpected behavior that arises from component interoperation. We also argued that this type of vulnerabilities are particularly difficult to identify in the development process. However, Chapter 2 did not present any evidence that the dark side of the code problem actually happens in software developed and deployed in real-world settings. Also, when analyzing results of the experiment in the previous chapter we identified that some vulnerabilities in Apache Struts could indeed be caused by integrating components without a proper understanding of the consequences it has on the overall application operation. We have also noticed that some vulnerabilities, once identified, may re-appear in future versions of the software either due to incomplete fixes or continued misunderstanding of the underlying problem, which could likely be caused by the security gap. In this chapter, we investigate how common is this phenomenon in software "in the wild".

Research aimed at gaining insights into the appearance of vulnerabilities in software has tended towards quantitative studies. Metrics such as a number of bugs over time, their rate of appearance, type and severity, can be gathered and statistically analyzed [93, 94]. Such measurements may point to interesting trends, however, they rely on their hypothesis and the efficacy of the underlying data

which, for example, may include attributes such as CVSS scores and lines of code. Furthermore, it cannot help one understand why particular security weaknesses persist over time and cannot be properly addressed, nor what causes the implementation of a weak security mechanism. While a quantitative study can provide a basis for supporting a hypothesis, where a hypothesis does not form the basis of the research question an exploratory approach is informative.

In this chapter, we take an exploratory approach to gaining insights into the reasons why developers introduce software vulnerabilities. Informed by qualitative research techniques, we carried out a systematic study of the evolution of security vulnerabilities over a long period of time with a view to discovering security-relevant phenomena that emerge. We continue to focus on Apache Struts, and in order to provide a sufficiently in-depth analysis, we consider a particular Struts functionality. We provide the background necessary to understand the technical account given on the evolution of the control discusses a number of phenomena that emerge during the analysis of this control.

9.2 Methodology

While in Chapter 8 Struts was used mainly because of its popularity and a number of vulnerabilities in a short period of time, our rationale in selecting it for this analysis are different. Struts is a mature and widely used package that has been developed according to best practices, both in terms of code implementation as well as development life cycle, with a documented policy and change management process. The security processes surrounding Struts are transparent and include documented processes for reporting vulnerabilities and publishing security advisories. We focused our attention on the functionality of one particular Struts security control that has had a series of reported security vulnerabilities and has evolved over time. The chosen control is sufficiently critical to ensure both internal and public interest in identifying security problems, and that reported issues are treated seriously by the development team.

We performed a systematic analysis of the Struts source code published over twelve years from 2004 to 2015. This was done in a qualitative style, whereby the objective was to identify security-related phenomena, or patterns, that emerge from the activity of making code revisions. The analysis focused on the code changes that arose as a consequence of, and/or were the cause of, the security advisories over that period. In particular, these were re-

lated to a security control that is responsible for preventing the injection of malicious code into the framework via the parameters of web page requests. This security control checks parameter values passed to the Struts `ParametersInterceptor` and `CookieInterceptor`, preventing their misuse. Note that the `ParametersInterceptor` functionality originated in the XWork project and was later merged into Struts; the XWork source was subject of the analysis during this initial period.

We reviewed: the security advisories/vulnerability publications; code-updates (security-related or otherwise); related discussions on the development mailing list, and other publications often contributed by the vulnerability reporters who sometimes provided additional technical details. Often, only partial details of an attack were published and, in all cases, it was possible to re-construct/implement the attacking code by reverse-engineering the code changes and published information. This led to the analysis of over 300 security-relevant code changes over the evolution of Struts.

In carrying out this analysis we identified the code changes that had an impact on security, either fixing a known vulnerability or introducing a security issue. For ease of exposition, the analysis is summarized in terms of aggregate changes over releases that culminate in a published security-related release. In this way, we believe that our inferences about the developer's intentions are more reliable than those based on (possibly incomplete) changes made in between security releases. While the observation of changes in-between the releases may provide an insight into security mechanism evolution, it was not clear whether the changes at these stages could be considered complete. As a result, we discovered 20 key security-related changes, that are summarized in Table 9.1.

During our investigation, we identified elements of the security mechanism and mapped the changes into the corresponding categories, given by the right-hand columns in the table. Each row of the table covers a key change to the security mechanism including the time of the change and the status of the security mechanism after the change. The identified changes often take a simplified form, of a regular expression or an acronym, representing the essence of the change. For simplicity, the significant element of the change is highlighted using **bold** text. Note that the categories discovered during the analysis are not related to the actual structure of the code, as the corresponding security mechanisms were routinely moved/refactored within the source code, given different names and so forth. Section 9.4 provides the detailed discussion about the changes and their

interpretation.

Throughout this process we strove to make observations about vulnerabilities, repairs and coding activities, based solely on the evidence in this corpus.

9.3 Struts Operation

This section provides an overview of those parts of Struts that are required to understand our analysis of the security control used in the study. In order to provide a sufficiently detailed discussion in this chapter we present further technical details of Struts, using an example. One of the features of Struts is the ability to easily separate the business logic from the operational details related to processing HTTP requests. For example, consider a sample piece of code of a web application responsible for handling a request to add an application user by an administrator, presented in Figure 9.1. The listing shows three parts of the application: class `User` encapsulates the details of an application user; class `AddUser`, implemented as Struts *action*, provides the logic for adding a user to the system, and the JSP provides the fragment of the *view* (a page is presented when the action is complete).

Note that `AddUser` does not contain any web-specific logic, which is handled by Struts. It defines getter and setter methods (`getNewUser` and `setNewUser`, respectively) for retrieving and setting the user object, and a `setSession` method (required by `SessionAware`) interface for Struts to set the Session object. For example, the client may send a request such as:

```
http://app/user/add?newUser.name=john&newUser.role=support
```

in order to add new support user. The request is received by Struts which, based on its configuration, decides whether it should be processed by the `AddUser` action. It instantiates the class and then (as it implements `SessionAware` interface) calls `setSession` method with a session map for the current user. Struts then parses the parameters, creates a new User object using the values from the request, and provides it to the action using the `setNewUser` method. Subsequently, the `execute` method executes, taking advantage of all the properties that have been set. After the user is added, the response is rendered and, during this process, the JSP code also refers to the action properties such as a newly added user and session.

```
public class User {
    private String name;
    private String role;
    ... // getters and setters for fields

    public boolean isAdmin() {
        return (role.equals("admin"));
    }
}

public class AddUser extends ActionSupport implements SessionAware {
    private User newUser;

    public User getNewUser() {
        return newUser;
    }

    public void setNewUser(User user) {
        this.newUser = user;
    }

    public void setSession(Map session) { // for SessionAware
        this.session = session;
    }

    public String execute() throws Exception {
        if (session.get("user").isAdmin()) {
            DAO.add(newUser);
            return SUCCESS;
        }
    }
}

<s:property value="#session['user'].name"/> added user <s:property
value="newUser.name"/> with role <s:property value="newUser.role"/>
```

Figure 9.1: A sample MVC Struts application code

9.3.1 OGNL

Struts uses Object-Graph Navigation Language (OGNL) [36], an expression language used primarily to get and set properties of Java objects. OGNL expressions are evaluated against a collection of objects called *context*. One of the objects, called *root*, is distinguished as the default root of the object graph. When processing a request, Struts sets the current action object (for example `AddUser`) as the root. In the example in Section 9.3, the expression `newUser.name` is used both in request parameters and also in the JSP file. The values are being accessed through public getters and setters. For example, the OGNL `newUser.name` to get the name of an object is equivalent to Java code `action.getNewUser().getName()`. Similarly, using OGNL to set a value expressed as `newUser.name` to `alice` is equivalent to Java code `action.getNewUser().setName("alice")`.

Other Struts-specific objects, such as session, request and application configuration, are also included in the OGNL context. For example, the expression `#session['user'].name` can be used to access a `name` property of an object that exists in the session map under index `'user'`. Finally, the context contains a number of variables that control the OGNL processor's behavior, such as rules for accessing classes depending on its type, access restrictions and caching. Figure 9.2 provides an overview of context structure at the time of execution of `AddUser` action.

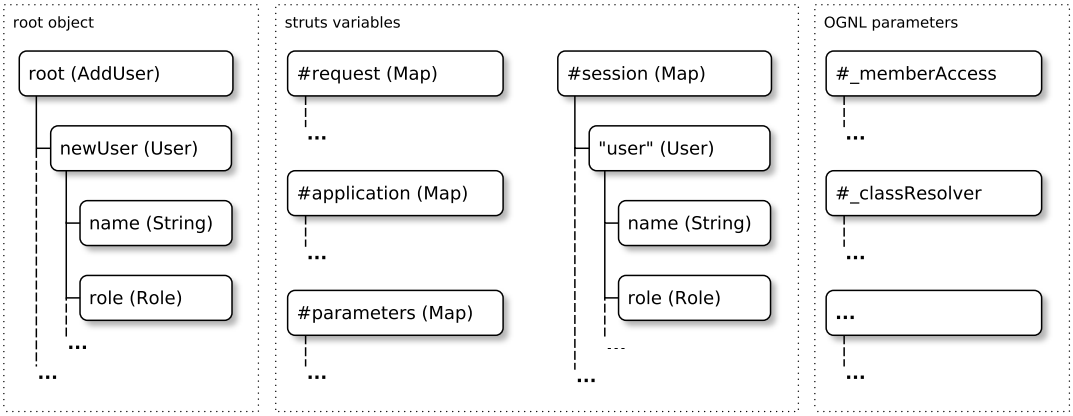


Figure 9.2: OGNL context in the example application

9.3.2 Struts Interceptors

Struts *interceptors*, upon which our study is based, parse request parameters and set corresponding values in action objects. Interceptors in Struts are responsible for handling common tasks before/after the action is executed. Typical tasks performed by interceptors is handling HTTP requests (such as request parameters or cookies), input validation, access control, caching and so forth.

Processing of the request parameters is performed by `ParametersInterceptor`. Our study started on the code published at the beginning of 2004, with the release of the XWork 1.0 library in which the interceptor was first implemented before being merged into Struts. Since then, the functional requirements of the parameter interceptor have not changed. It iterates over each parameter and sets the action values using parameter name as OGNL expression to identify the object, and parameter value as the value to set. Starting from June 2007, `CookieInterceptor` sets action properties based on HTTP cookies.

9.4 Tracing the Evolution of a Security Control

The parameters and cookie interceptors allow clients to provide custom OGNL expressions that are evaluated by Struts. OGNL expressions can result in the execution of custom code which accesses program variables. From its first release, this functionality has been considered a security threat and, a mitigating security control has always formed a part of its implementation. In this section, we systematically trace the evolution of this control over a 12 year period: 2004–2015. The results are summarized in Table 9.1 and described in detail in the remainder of the section.

In January 2004 the interceptor included just one security measure: disabling Java method execution (ME) through OGNL. By default, OGNL allows Java methods to be called in a manner similar to field access. For example, an attempt to set a value of a property specified by expression`map[method()]` results in the invocation of `method` against the root object in order to get the value to be used as `map` key. Disabling the method execution is implemented using a custom OGNL method accessor and controlled by `context['xwork.MethodAccessor.denyMethodExecution']` variable.

date	accepted parameters	accepted cookies	excluded patterns	OGNL
Jan 2004	[empty]	n/a	n/a	**ME**
Dec 2004	excluded: {'=', ',' ,'#'}	n/a	n/a	ME
Feb 2007	excluded: {'=', ',' ,'#', ':'}	n/a	**dojo\..***	ME
Jun 2007	excluded: {'=', ',' ,'#', ':'}	[empty]	dojo\..*	ME
Jul 2008	excluded: {'=', ',' ,'#', ':'}	[empty]	dojo\..*	ME, **SM**
Jul 2008	ex: {'=', ',' ,'#', ':', **\u0023**}	[empty]	dojo\..*	ME, SM
Oct 2008	**[\p{Graph}&&[^,#:=]]***	[empty]	dojo\..*	ME, SM, **SC**
Aug 2010	**[a-zA-Z0-9\.\]\[\(\)_'\s]+**	[empty]	dojo\..*,^struts\..*	ME, SM, SC
Dec 2011	[a-zA-Z0-9\.\]\[\(\)_'\s]+	**[a-zA-Z0-9\.\]\[_'\s]+**	dojo\..*,^struts\..*	ME, SM, SC
Dec 2011	**[a-zA-Z0-9\.\]\[\(\)_']+**	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*	ME, SM, SC
Jan 2012	**\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))***	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*	ME, SM, SC, **EE**
Apr 2012	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..***,^session\..*,^request\..*,^application\..* ^servlet(Request|Response)\..*,parameters\...***	ME, SM, SC, EE
Aug 2012	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))* **[100]**	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*, ^servlet(Request|Response)\..*,parameters\...*	ME, SM, SC, EE
Mar 2014	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**^class\..***,^dojo\..*,^struts\..*,^session\..*, ^request\..*,^application\..*,^servlet(Rrequest| Response)\..*,^parameters\..*,^action:.*,^method:.*	ME, SM, SC, EE
Apr 2014	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**(.*\.|^\.*|\[('|"))(c|C)lass(\.|('|"))]|\[).***,^dojo\..*, ^struts\..*,^session\..*,^request\..*,^application\..*, ^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
Apr 2014	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**default:** (.*\.|^\.*|\[('|"))class(\.|('|"))]|\[).*; **params:** ^dojo\..*, ^struts\..*,^session\..*,^request\..*,^application\..*, ^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
May 2014	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['\w+'\])|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	(.*\.|^\.*|\[('|"))class(\.|('|"))]|\[).*,^dojo\..*,^struts\..*, ^session\..*,^request\..*,^application\..*, ^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
Dec 2014	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['(\w|[\u4e00-\u9fa5])+'\])|(\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	**config/params:** ^action:.*,^method:.* (priority) **default:** (.*\.|^\.*|\[('|"))\bclass(\.|('|"))]|\[).*,(^|.*#),dojo(\.|\[).*, (^|.*#)struts(\.|\[).*,(^|.*#)session(\.|\[).*,(^|.*#)request(\.|\[).*, (^|.*#)application(\.|\[).*,(^|.*#)servlet(Request|Response) (\.|\[).*,(^|.*#)parameters(\.|\[).*,(^|.*#)context(\.|\[).*, (^|.*#)_memberAccess(\.|\[).*	ME, SM, SC, EE, **EC**
May 2015	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['(\w|[\u4e00-\u9fa5])+'\])|(\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	(^|.*#)(dojo|struts|session|request|application| servlet(Request|Response)|parameters|context| _memberAccess)(\.|\[).*'", ^(action|method):.*	ME, SM, SC, EE, EC
Sep 2015	\w+((\.\w+)|(\[\d+\])|(\(\d+\))|(\['(\w|[\u4e00-\u9fa5])+'\])|(\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	**(^|\%\{)((#?)(top(\.|\['|\[")|\[\d\]\.)?)**(dojo|struts|session| request|response|application|servlet(Request|Response| Context)|parameters|context|_memberAccess)(\.|\[).*, ^(action|method):.*	ME, SM, SC, EE, EC

Table 9.1: Security mechanism evolution: 2004–2015

9.4.1 Tampering with OGNL

The first vulnerability that was identified since the initial release relates to overwriting context variables via parameters. For example, a parameter and also an OGNL expression `#session['user'].role` may be used to set the role of the current user stored in the session. A more advanced expression may set a number of properties at once while still retaining original behavior, that is, setting the `newUser` parameter: `#session['user'].role=admin,#testMode=true,newUser.name'`.

In December 2004 the problem was fixed by modifying the interceptor to check if the name is acceptable, by verifying it against a blacklist of characters, using the following condition.

```
if (name.indexOf('=') != -1 || name.indexOf(',') != -1 ||
    name.indexOf('#') != -1)
```

In July 2008 it was reported that the fix was incomplete, because the # character can be encoded using its Unicode `\u0023` replacement, for example, by using a parameter `\u0023session['user'].role=admin`. This problem was fixed by adding string `\u0023` to the blacklist. Note that the other two already blacklisted characters, which could also be represented using the Unicode string, were not included in the Unicode form. Shortly after, the fix was further modified in a twofold way. First, the code was modified and the check routine replaced with a regular expression: `[\p{Graph}&&[^,#:=]]*`. Note that the Unicode replacements for the characters were no longer excluded. In addition, the interceptor was modified to run OGNL operations against a separate temporary instance of the context object (SC), without Struts-specific variables such as `session` preventing their manipulation.

At the same time, a new problem was discovered. OGNL allows accessing static fields in Java objects using the `@class@field` notation. For example, expression `@java.lang.System@exit(0).foo` can be used to call static `exit()` method, causing the JVM to exit. Static methods provided by the Java standard library can be used to perform a number of operations, including executing custom commands. This problem was fixed by adding an option to disable static method access (SM) in OGNL and disabling it by default.

A vulnerability in this security mechanism was found in July 2010. It took advantage of the fact that context variables were still accessible through the Unicode "trick" *and* that the OGNL-specific context variables controlling access

to method execution, were available without restriction on the temporary context.

This allows a modification of the OGNL runtime configuration, allowing method execution and eventually custom method execution. An example sequence of OGNL expressions for the attack could be the following [95].

```
#_memberAccess['allowStaticMethodAccess'] = true
#foo = new java.lang.Boolean("false")
#context['xwork.MethodAccessor.denyMethodExecution'] = #foo
#rt = @java.lang.Runtime@getRuntime()
#rt.exec('mkdir /tmp/PWNED')
```

Such a sequence could be encoded in a parameter, bypassing the blacklist, by using Unicode replacements for the `#=,` characters. This vulnerability resulted in a change in the regular expression to a stricter whitelist of characters: `[a-zA-Z0-9\.\]\[\(\)_'\s]+` effectively disallowing usage of the Unicode replacements.

A year after this change, there was a report that while the restriction worked for regular methods it did not apply to public constructors. While execution of methods by OGNL was disabled through custom accessors, the logic did not cover constructor invocation. Some constructors may be useful to an attacker, such as `FileWriter` constructor creating or overwriting a file (`new java.io.FileWriter('filename')`). Rather than disabling constructor invocation in the existing custom accessor, the issue was fixed by disallowing a white space character, essential for constructor syntax, in the parameter name. As a result, the regular expression was modified to `[a-zA-Z0-9\.\]\[\(\)_']+`.

A few weeks later, in December 2011 the new way of bypassing the restriction was discovered [96]. It took advantage of OGNL's ability to evaluate the content of variables that already exist in the context. The attack requires setting two parameters. The first parameter uses an acceptable name but has a value containing the OGNL expression, and the second refers to the first, for instance, as an array key. For example, an attacker may first set the value of existing parameter to an expression, `newUser.name=OGNL code` and then evaluate the parameter value by referencing its name, `z[(newUser.name)(0)]=0`. When the second parameter is evaluated, OGNL will attempt to establish an index of `z` property, and evaluate the expression stored in `newUser.name`. In effect, the vulnerability allows character-based restrictions on parameter names to be bypassed. This, in turn, enables access to context variables that control method

execution restrictions and lead to executing custom code. The vulnerability was fixed by modifying the regular expression for an acceptable name, yet again. This time, it matched characters such as `[]` or `()` only in a specific context so as to disallow expressions that may evaluate other variables. An additional logic to control the expression evaluation (EE) was added to OGNL customization code.

The last vulnerability related to tampering with OGNL using parameter names was reported and fixed in August 2012. As parsing OGNL parameters requires significant processing effort it is attractive as a target for denial of service attacks. Requests with particularly long/complex OGNL expressions can be used to exhaust system resources. The problem was fixed by limiting the size of parameter names to 100 characters.

9.4.2 Accessing properties

Another set of security problems related to processing request parameters relate to the ability to access properties of the root (action) object. In OGNL, access to the properties is controlled by the method or field access modifier in Java. For example, the `newUser` property is accessible because of the public getter `getNewUser`. If the method was defined as private or protected then the access would not be possible. The relationship between the ability to access and actual method access may not be clear or always intended. It may happen that the developer's code already has an object that would perfectly suit use within an action, but it includes a public method that should not be exposed. For example, a different implementation of the example application Section 9.3 may use the `User` object but does not intend to allow the user to set the `role` parameter. In February 2007 a configuration parameter named `excludeParams` was provided in order to allow developers to prevent access to some properties. The parameter can be set to a comma-delimited set of regular expressions defining patterns for parameter names that should be ignored by the interceptor. Initially, the parameter was set by default to `^dojo\..*` and shortly after also `^struts\..*`.

Struts uses a dependency injection software pattern. In particular, it allows action classes to acquire certain common runtime information by implementing specific interfaces such as `SessionAware` or `RequestAware`. The example application in Section 9.3 implements `SessionAware` interface and corresponding `setSession` method. This instructs Struts to call it with the current server's session before action execution. Note that implementing a corresponding public

getter (not required by the interface) could open up a session for manipulation using request parameter, such as `session['user'].role`. If the application implements such getter, it is expected to restrict access to the parameter in the configuration.

Between 2007 and 2011 various reporters pointed out that implementing a setter, as required by Struts interfaces may also allow for manipulation. While the user may not directly access session attributes due to the lack of a getter, it may override a session object provided to the action. For example, a parameter `session.user=a` results in creating a new Map with the `user` key. The use of this vulnerability is rather limited [97] but, in certain cases, it may allow unintended manipulation of application internals. In April 2012, a fix was eventually implemented by including a number of common parameter names such as `session` to the `excludeParams` list. The problem was not completely solved as it only protects a few commonly used properties and the override mechanism is still available.

A more significant problem was discovered in March 2014. Every Java object contains a `getClass` method that returns a Java class for that object. The returned `Class` object contains a number of getters and setters, in particular `getClassLoader`, which returns an instance of the current class loader. Access to this object allows manipulation of the application server's internal state and allows for custom code execution [98]. The first attempt to fix the vulnerability was to add `^class\..` pattern to `excludeParams`. Within a few days, a number of vulnerabilities related to an incomplete fix were reported. One was the pattern matches `class` string at the beginning of the parameter name (`^`), but the `class` property does not necessarily have to be accessed through the root. As all Java objects contain `getClass` method, the class loader manipulation can be done through any of them, for example `newUser.class.classLoader...` Another reported vulnerability related to the fact that OGNL allows specifying parameters in upper-case form, such as `Class`, which are not matched by the regular expression. An improvement was published on the Struts web page as a hotfix, in the form of suggested custom configuration of the interceptor `(.*\.|^|.*|\[('|"))(c|C)lass(\.|('|")]|\[).*`. It must be noted that, while an upper-case version was considered only for `class`, but not for `session`, `request` and others that were previously excluded. Eventually, the code performing the regular expression matching was modified to ignore case, and the expression was simplified.

The series of fixes related to the `class` attribute resulted in a number of rather ad-hoc code changes that were rationalized in December 2014. The default set of excluded patterns was moved from the configuration file directly to the utility class used for pattern matching. Two, security unrelated, patterns were kept in the configuration file. However, the code was implemented in a way that configuration parameters overwrote the default set, effectively removing all security related excluded patterns. This problem was fixed by moving the two patterns to the code and leaving the configuration empty.

The last vulnerability relates to a special variable called `top`, implemented for the Struts-specific handling of OGNL, allowing access to the root object. Effectively, this variable allowed excluded patterns of parameters to be bypassed by allowing variables to be addressed in a way that does not match the regular expressions. As a result, the `top` parameter was added to the list of excluded patterns.

Finally, a more comprehensive fix was implemented. In addition to a regular expression specifying parameter names, a custom OGNL property accessor provided by Struts was modified to exclude classes (EC) by their types and package names. For example, any attempt to access an object of a type `java.lang.Class`, or any class in `javax` package (specific to J2EE objects such as session), will be rejected. This mechanism does not have the weaknesses of the string matching approach that was repeatedly bypassed, as the verification of the object type is done at OGNL accessor level, regardless of how the expression was constructed. However, the list of excluded classes and package names is rather arbitrary.

9.4.3 CookieInterceptor

Since June 2007, Struts includes the `CookieInterceptor` with functionality similar to `ParametersInterceptor` but applicable to HTTP cookies. The interceptor iterates over the cookies sent with the request and sets the value indicated by the OGNL expression provided by the cookie name. In the interceptor configuration, the developer may set the parameters/cookie names to processed or specify to accept all cookies.

Our analysis revealed that in several instances, problems that were applicable to both interceptors were fixed only for the `ParametersInterceptor`. At the time of the first release of `CookieInterceptor`, the developers were aware of OGNL tampering issues and the rudimentary protection was already implemented for parameters, as presented in Table 9.1. However, it took over four years and

an external reporter to implement the whitelist of accepted characters, similar to that for parameters.

Additionally, issues related to accessing parameters, described in Section 9.4.2 were not considered for cookies for quite some time. Until April 2014 there was no restriction as to what properties can be accessed with cookies and the `excludeParams` configuration directive was applicable only to parameters. In particular, the initial fix for the critical class loader manipulation issue was also only applied to parameters. Only after the problem was explicitly reported was the problem fixed, though only for the `class` property and not for the `session`, `request`, and so forth.

9.5 Analysis of Security Control Evolution

As we traced the evolution of the security control, as outlined in the previous section, we observed a number of repeating phenomena related to introduction and persistence of vulnerabilities, and inhibitors to the proper implementation of the security control.

9.5.1 The dark side of the code

One challenge is the difficulty of properly understanding every aspect of an application's operation. Modern software development is built layer upon layer of components, each encapsulating lower level detail. However, security issues often relate to low-level details that are not always accessible to the developer. As a result, programmers rarely understand all the operational details of the entire stack. This problem, referred to as the "dark side of the code" [23], discussed Chapter 2, can be viewed as a gap between the possible operation of the application as perceived by the developer and the actual operation of which the software is capable. While this dissertation and [23] argue, in principle, for the existence of the dark side of the code, our study observed this phenomenon occurring in a number of vulnerabilities and confirm its existence in practice.

When the Struts interceptor developers designed the initial set of forbidden characters, they did not consider their Unicode alternatives that were later used to bypass the blacklist based security control. The OGNL library allows the use of such replacements, however, there is inadequate information concerning the

scope in which the characters can be used. In its coverage of string literals, the official OGNL documentation makes a vague mention of escape characters. In addition, the information provided about escaping characters is done in the context of string delimiters such as `"` and `'` and could be easily interpreted as applying only to them. The same issue applies to OGNL's ability to address properties using upper case characters.

Similarly, that the `getClass` method, implemented by the JVM and existing in every Java object, may be used to perform an attack might not have been expected by the Struts developer. The complexity of this attack confirms that an in-depth understanding of the Java internals, as well as the class loader specific to the application server, is required in order to develop an attack vector. Additionally, the developers might not have expected that access to public constructors, exploited later as file overwrite attack, could be harmful. As it is the best practice in object oriented programming to not implement constructors that cause any side effects, it may be difficult to appreciate that Java standard library includes one that allows writing a file.

9.5.1.1 Report bias

A dark side can also exist when it comes to both documenting and/or interpreting vulnerability reports; the extent of the security problem may not be fully appreciated in its reporting. Security vulnerabilities are often identified by security researchers who are external to the development team. Usually, the issue is reported with a detailed description of the problem, example attack vector, and so forth. Upon receiving the information about the problem, the developers may follow a detailed report in isolation, as the prescription for the vulnerability's remedy. However, often the reporter may not have a complete understanding of the application and their report may be incomplete; or they may limit their focus to a representative example. The vulnerability, however, may have a broader scope than that identified by the report or there may be further attack vectors related to the same root cause of the issue.

In Struts, the sequence of fixes related to the `class` property exemplifies this phenomenon. Each time, the remediation was shaped by the way the issue was reported. This is exemplified by the usage of the `class` parameter at the beginning of an expression (while it can be used for any object) and failing to provide the protection for `CookieInterceptor`.

Similarly, an issue related to the exposure of constructors when using OGNL was reported as a problem that led to the overwriting of custom files. Although this was only one example of the attack vector, this is how the vulnerability was described in Struts official advisory, despite the problem having a broader scope. In reality, a number of other actions are possible, provided the availability of a suitable public constructor in the class path [99].

9.5.1.2 Security metric bias

During the analysis, the vulnerabilities were compared to the published official security advisories. We noticed that in many cases, the CVSS score did not properly represent the problem. This can be attributed to an incomplete understanding of the problem when the report was published. The CVSS documentation acknowledges that the characteristics of a vulnerability can change over time; the *temporal* metrics, used to calculate the temporal score include properties such as exploitability or remediation level. However, it is the base metrics, such as Confidentiality/Integrity Impact that often change as the problem is better understood.

For example, CVE-2008-6504 describing the Unicode vector to bypass the blacklist of characters has a CVSS score of 5.0. The impact metrics for Confidentiality and Integrity are, None and Partial, respectively. Another occurrence of the same problem, that resulted from an incomplete fix due to adding a temporary context object, published in CVE-2010-1870 has the same score. This is, however, not consistent with the actual impact of the vulnerability. Access to context variables effectively allows execution of custom Java code, system commands, and more. It is likely that the team was not aware of the impact when the first advisory was published; in the second case, however, the official advisory points out the variables used to control method execution. The last vulnerability CVE-2012-0392 reported for this problem, relating to evaluating OGNL expressions using two parameters, has correct Confidentiality/Integrity Impact metrics of *Complete* and the overall score of 9.3. Even though, in hindsight, it is clear that the impact of all three issues was the same, the published information is still incorrect, something that can only be revealed by detailed analysis.

Thus, CVSS values can be biased by the understanding of the problem at the time of advisory publication. Therefore, and irrespective of the objectivity of the measure, it may not be appropriate to use CVSS in a temporal context: using it

to compare the (in)security of an application may lead to incorrect conclusions. The extent to which this may influence the results of past studies is a subject for further investigation.

9.5.2 Developer's blind spots

Anticipating security problems requires a cognitive effort and often is a distraction from the main objective of the developer. Oliveira et al. [26] show that developers often fail to correlate security problems to their workload even if they are aware of the problem in general. Oliveira's experimental hypothesis was that vulnerabilities can be blind spots in developer's heuristic-based decision-making processes: while a programmer focuses on implementing code to meet functional demands, which is cognitively demanding, they tend to assume common, but not edge, cases. Supporting the hypothesis, the study [26] found that 53% of its participants knew about a particular coding vulnerability, but they did not correlate it with an experimental programming activity assigned to them unless it was explicitly highlighted.

Our analysis confirms the existence of this phenomenon in a mature product and experienced team. Even where developers are expected to be aware of the security problems (as they encountered them in the past), they may fail to consider them. When the cookie interceptor was implemented, the developers were aware of possible issues related to evaluating OGNL expressions without restrictions. Some of the restrictions were already implemented for the parameters interceptor. Yet, for three years the corresponding protection was not considered for cookies. Similarly, the access to Struts-specific `top` object, that allowed bypassing the excluded parameters list was well understood by the team. The `top` object facilitates the extensions to OGNL provided in Struts and, as such, it is described in the documentation. However, for almost four years when various parameters were excluded for security reasons, it was not considered in the regular expressions.

Overlooking the unrestricted access to public constructors could also be partially attributed the problem of developer's blind spots. While, at first, the developers might not have been aware of the potential security exposure it introduces (dark side), it was no longer the case after July 2010, when usage of the constructor was highlighted to the team in the context of another reported vulnerability. Also, the example exploit, included in the advisory published on the Struts website, explicitly used the `Boolean` class constructor. While invoking the constructor

was not a primary objective of the attack, it was used to facilitate it. The usage of a constructor was not considered when preparing the fix for the previous issue, even though it could have been easily included.

9.5.3 Opportunistic fix

When developing the fix for a security problem, developers may prefer an implementation that fits their existing code. While the fix related to the root cause of the problem may be more suitable and more comprehensive, developers tend to develop fixes that are more convenient to implement and that do not cause disruption to the existing code structure.

Prevention against modification of context variables or execution of custom code was first implemented using simple pattern matching of OGNL expressions, rather than by limiting OGNL's capability to perform these operations, which followed later. While we do not know why exactly this approach was taken, our analysis of the source code from 2004 shows that, in the code structure of the time, a more comprehensive solution required major changes in a number of helper classes and, perhaps, in the OGNL itself. Over time, and in response to numerous issues, a more comprehensive solution was implemented at lower level of abstraction, explicitly interacting with OGNL processor.

Similarly, prevention against class loader manipulation was first implemented by adding a pattern to the list of excluded parameters, which was already in place. Only after a series of vulnerabilities arising primarily from the gaps in the regular expressions were reported, a more comprehensive fix, which involved specification of excluded classes and packages, was implemented. However, at the time of writing the defense against constructor execution still relies solely on the regular expression matching, specifically the lack of the white space among allowed characters. As the regular expressions were often bypassed in the past, it may be more suitable to include protection against constructor execution at the OGNL level.

9.5.3.1 Compatibility problems

Sometimes, fixes to vulnerabilities are sub-optimal in the interest of compatibility with older versions and existing consumer workloads. Software consumers may rely on a particular functionality that was subsequently identified as a source of

the security problems. Thus, there must be a trade-off between a comprehensive fix that breaks consumer's code and a weaker fix that may not fully address the problem or contribute to security vulnerabilities in the future, for example by the introduction of the dark side of the code.

In Struts, many vulnerabilities were related to execution of static methods. As many applications take advantage of this functionality, it prevented the Struts team disabling it completely, which would be a preferred fix. Instead, the static method execution was controlled through configuration. As a result, the property controlling method execution disablement became a frequent target for other vulnerabilities and allowed escalating any context manipulation issue to remote code execution. Completely turning off this functionality is being planned since 2014 and developers are warned that it should be considered obsolete. Similarly, the plan to remove the `top` object was recently announced.

9.5.4 Counter-intuitive mechanism

Some fixes can mean that security controls in a software component can become difficult to understand or counter-intuitive. While the component may not, strictly speaking, contain a vulnerability, systems using it may introduce their own vulnerabilities, due to incorrect usage of the security controls.

The problems of developers not properly understanding the relationship between method access and property exposure were discussed in Section 9.4.2. Anecdotally, many application developers are not aware of the problems arising from implementing public getters/setters for sensitive objects, or are unaware of exposing them through inheriting a class that contains such methods. The Struts team also fall victim to that problem with the `class` property, which has not been considered for over 10 years.

Initially, the `excludeParams` configuration property was not implemented as a security mechanism. It was intended to make the interceptor ignore some URL parameters that are used by other layers of the application. For example, a popular JavaScript framework dojo often uses `dojo.preventCache` parameter for its own purpose. Such parameters will not have the matching properties in the action classes, and attempting to process them through the `ParametersInterceptor` results in errors/exceptions, hence they are easy to spot and include in the configuration. Some patterns (such as `^dojo\..`) were set in the default Struts configuration file shipped within the Struts JAR file.

At that time, Developers could specify other, application-specific parameters in their application's configuration file. It was not a concern that the developer would overwrite the default pattern by setting their set of specific patterns, as the application developers are normally aware what non-action parameters their application uses, and would include a full list of those parameters in their configuration file.

Then, gradually, in order to remediate reported vulnerabilities, security related parameters were added to the default configuration and the `excludedParams` setting become a part of Struts security mechanism. After that, in order to maintain the security, the application developer has to find the current set of patterns from the default configuration file and include it when specifying patterns in the application-specific file, as otherwise they would overwrite the pre-defined security-related patterns. In addition, each time the application upgrades Struts to a new version, the process has to be repeated as the new version may contain new patterns. At one point the Struts team itself accidentally became a victim of this process. In version 2.3.20, released in December 2014, the code responsible for applying the patterns has been modified. Most of the patterns had been moved to a separate class handling pattern matching for both (parameter and cookie) interceptors. The two remaining patterns were kept in the default configuration file. This change overwrote the patterns for security-related properties (such as `class`) contained in the struts code, by those in the default configuration file. As a result, the release 2.3.20 had effectively re-enabled all previously fixed attack vectors related to property access. The eventual fix to the problem was moving the two patterns from the configuration to the code. The fix did not address the problem of overriding security-related excluded patterns by the configuration, but simply meant that the default configuration was empty. Now, in order to set their own excluded patterns, the application developer has to obtain the current version of the default patterns from the Struts Java code and append their own patterns.

9.5.4.1 Assumptions about consumers

A factor that contributes to the implementation of a counter-intuitive security mechanism is incorrect assumptions about the consumers' understanding of security mechanisms. The developers may not be aware that a typical consumer does not understand all subtleties of the security framework. In analyzing the discussion on the Struts issue tracking system, we noticed that some of the ini-

tial reports on security problems were dismissed, due to the existence of technical means to counter default insecure behavior by specific configuration or customization. Some of these issues were eventually admitted as vulnerabilities and the default behavior was changed.

Struts developers might not have realized how counter-intuitive the management of the `excludeParams` property was until it impacted on themselves. In fact, at the time of publishing the fix for the accidental overwriting (described in the previous section), one of the developers opened an issue in the Struts tracking system to make the patterns additive.

Another example of this problem is the exposure of J2EE objects, such as session or request through public getters and setters, required by the dependency injection mechanism, as described in Section 9.4.2. It was assumed that application developers would implement their own protection, such as custom parameter checks. However, it is unlikely that a casual Struts consumer will be aware of such an option or the need to use it. Many application developers are not even aware that the implementation of the getter to match a setter is not required by Struts interfaces; it is a common Java practice to implement matching getters, but clearly poor Struts practice as it may expose sensitive properties such as `session`. In November 2016, a query [98] to a popular GitHub repository shows 8,337 instances where a class implementing `SessionAware` interface also implements a public `getSession` method. Eventually, access to these objects was recently disabled at OGNL level, regardless of getter/setter access modifiers or their inclusion in the excluded patterns.

9.5.5 Evolution of phenomena

During our analysis, we noticed that the above-mentioned phenomena tend to appear in the order given in Figure 9.3: they evolve as the developer's knowledge about the security problem and understanding of the issue and the consequences of fix increases. At first, the developer may not be aware, or only partially aware, of a potential security problem. This may be caused by an incomplete understanding of the full operation of the application or relying on an incomplete advisory by the third-party. As they become more aware, developers may fail to remediate the problem fully due to the blind spots in their programming process. Also, the fix may be applied only to some scenarios or to some parts of the system but not the other. Later, the developed fix may not be comprehensive

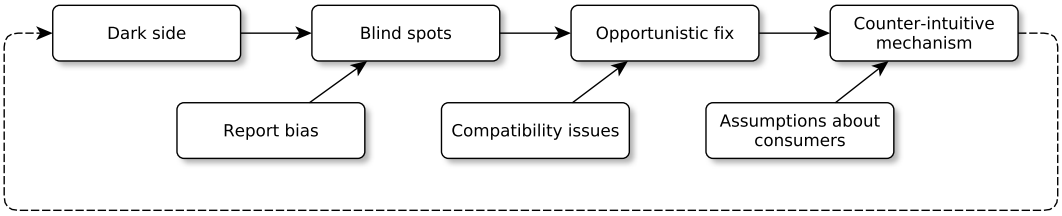


Figure 9.3: Phenomena life cycle

or not fix the root cause of the problem. This may be a result of an attempt to implement the fix with the least possible effort, or the technical constraints, such as compatibility with previous versions. Finally, the resulting security mechanism may be counter-intuitive, resulting in its incorrect usage by the consumers. Note that the end of the sequence at one level of abstraction may become a starting point for the security problem at a higher level of abstraction, where the counter-intuitive mechanism of a framework or library contributes to the problem with comprehending system's low-level details (dark side of the code) of the consuming application.

9.6 Conclusion

A systematic analysis of the Struts parameter and cookie interceptor controls over 12 years was carried out. A number of phenomena emerged in the evolution of the control, and these provide insights into why insufficient controls were implemented. In addition, we observed that the phenomena have their own life cycle as developers' understanding of security issues increase.

In particular, the "dark side of the code", discussed in Chapter 2, stands out as the primary cause of the vulnerabilities, with other, such as developers blind spots or opportunistic approach to issue fixing, emerging from it. The majority of the identified issues are related to interoperation between Struts and OGNL. At first, it was considered an abstraction that allowed an easy enablement of rich parameters within Struts without considering any details of OGNL language or the supporting library. However, as subsequent vulnerabilities were identified, that abstraction was gradually broken. Eventually, Struts included a significant amount of customization to the OGNL processor and controls that often interleave significantly with its operation. In fact, it were these changes that have proven to be most effective and properly fixed the vulnerabilities.

Chapter 10

Conclusion

Code reuse, typically in the form of software libraries and, recently, cloud services is a foundation of contemporary software development. It allows developers to dedicate their time and energy to core business logic of the application, instead of considering all the common and routine low-level details. The increased development velocity and the reduced cost come at a price of losing control of application operation, and enabling unexpected behaviors that undermine its intended security controls. These unexpected behaviors, which we define as the dark side of the code, often result in security vulnerabilities that are not visible at the high abstraction level that developers work on, and are therefore very difficult to detect using typical security practices. It is a kind of paradox that developing at a high level of abstraction involves encapsulating low-level details, but ensuring the security of an application involves breaking this abstraction and consideration of those details.

Anomaly detection was used in the past to identify attacks exploiting software vulnerabilities, but most of the work considered simple software and a small number of vulnerabilities. Our hypothesis is that anomaly detection techniques could be used to detect instances of the unexpected behavior and help with the identification of underlying vulnerabilities. The goal was to investigate how practical it is to apply anomaly detection to large contemporary software in real world development and deployment scenarios. This necessitated identifying the key challenges of this approach. Focusing on a common scenario of Java web applications, we considered problems of selecting an appropriate abstraction level and scope in which application behavior should be monitored, acquiring the baseline behavior and finding a suitable model to abstract it.

Analysis of sample applications and vulnerabilities led to the observation that monitoring application method call sequences provides the view of application activity most suitable for anomaly detection. Because the system operation at this level has often a form of distinct transactions, we developed a general model of behavioral norms to capture this type of behavior. Inspired by previous work in the area, we provided an interpretation of the model based on n-grams and implemented a proof of concept runtime verification component that allows easy integration with existing applications.

Evaluating the proposed mechanism was a challenging task. As our focus was not only on the efficacy of the anomaly detection technique but also its applicability as a practical solution for modern large-scale software, we had to perform a sufficiently comprehensive experiment. We validated the runtime verification system against the complete set of nineteen vulnerabilities reported over a five-year period for Apache Struts. Preparing the experiment involved examination of these vulnerabilities, which often required in-depth analysis of the source code changes in order to develop corresponding test cases. The outcome of the experiment suggests that the anomaly detection can, in principle, be a useful technique for discovering the unexpected application behavior resulting from attacks exploiting software errors.

The analysis of the results led to a number of insights into the nature and root causes of the programming errors. In order to better understand these problems, we increased the time frame of our investigation to twelve years and analyzed the evolution of one part of the Struts framework in more detail. We identified that the accidental misuse of software components due to an incorrect and insufficient understanding of their operation was the primary reason for developers to introduce security vulnerabilities. We also discovered a range of related phenomena.

Below we summarize results, contributions, limitations and future research areas for the three main themes of this dissertation.

The dark side of the code The first contribution is the discovery of the phenomenon of "dark side of the code" (Chapter 2), the unexpected behavior of the application resulting from accidental misuse of software components. The study, which covered twelve years of evolution of security controls and vulnerabilities in Apache Struts (Chapter 9), confirmed the existence of the dark side "in the wild". It also led to the discovery and confirmation of other phenomena that contribute to security weaknesses in the software and hinder the remediation.

This phenomena, while at a high level related to general area of developer's flaws in software engineering provide a new, security-specific interpretation of those problems point out to new root causes of their appearance. This analysis also demonstrated the importance of qualitative studies of software vulnerabilities as well as the effort that is required to perform them.

Recognizing the dark side of the code as a separate problem of secure software engineering can lead to better understanding of the mechanisms that cause developers to introduce security vulnerabilities in component-based software. This may allow improving software development and deployment processes by providing specific solutions that can reduce the security gap. In the future research, we plan to develop a set of practices for safe component development and composition that can help to avoid accidental component misuse and corresponding security vulnerabilities.

Model of behavioral norms The second contribution is the definition of the formal model of behavioral norms (Chapter 5). It is a general purpose framework for inferring transaction-like behavioral patterns from the system logs. We have demonstrated that behavioral norms could be used in model various types of system traces and events, such as application method calls, HTTP logs, system permission checks and API calls between separate collaborating web services (Chapter 6). We have also proposed the technique of automatic discovery of model parameters and evaluated it using simulated and a real-world systems. Finally, we explored other applications of the behavioral norms model, in particular the discovery of the underlying patterns of system and user behavior by behavioral norms aggregation (Chapter 6).

We have presented a simple interpretation of the model using sets of n-grams. This implementation was sufficient to demonstrate the application of the model for detecting anomalies in software, in the scope of our experiments. The future research should evaluate the suitability of other anomaly detection techniques such as sequences of variable length [55, 56], finite state machines [56] and neural networks [57]. These alternative data models may be used to improve the performance and expressiveness of the model and also provide new properties that can be used to analyze system structure and operation through the analysis of the norms.

Anomaly detection in real-world, contemporary software The third contribution is the analysis of challenges of applying anomaly detection to large-scale contemporary software systems (Chapter 4). We identified the key problems that are typically not considered in the previous research, such as the selection of abstraction level and scope and acquiring baseline system behavior in a way that could be applicable to modern software engineering practices.

Informed by these problems, we proposed a mechanism and prototype implementation of the runtime verification for Java (Chapter 7). It should be straightforward to adapt this mechanism to similar platforms, such as .NET. However, other very different platforms, such as Node.js, with dynamically typed language and extensive use of concurrency become increasingly popular in the enterprise configurations. It remains an open question how difficult it could be to adopt our techniques to such environments. The usage of Aspects allowed easy integration with Java applications but it may result in unacceptable performance degradation. Future research should consider alternative ways to enable runtime verification such as direct integration with Java Virtual Machine. Also, we discussed a number of potential ways to generate the baseline behavior. Using an application scanner combined with automated functional tests provided satisfactory coverage in the scope of our experiments. Future research should examine efficacy of application scanning in context larger applications running in production systems and applicability of other automated methods of collecting a baseline behavior such as glass-box testing or automatic test generation. This may lead to improved coverage and development of more advanced methodologies suitable for software in production configurations.

The runtime verification mechanism was experimentally evaluated in the context of nineteen vulnerabilities reported for Apache Struts versions over a five year period. Experiment results confirm that the anomaly detection can be applied to complex contemporary software components and provided further insight into the practical challenges of this task. It is, to the best of our knowledge, the first analysis of the efficacy of anomaly detection in the context of the full set of security vulnerabilities reported over a long time period for a real-world software system.

While the experiment was comprehensive when compared to the other work on the subject [14, 15, 17–19], it considered only a single, albeit complex, software component. Future research should consider if similar results could be achieved for other software. Also, it remains an open question what would be the perfor-

mance of anomaly detection if it was applied to multiple components simultaneously. Future research should consider different strategies, such as combining the behavior of components or monitoring its operation in isolation, and their impact anomaly detection efficacy, model size and performance overhead. This will provide a further insight into applicability of anomaly detection to software at scale and may lead to development of more advanced model interpretations or integration mechanisms.

Appendix A

Vulnerability Test Cases

A.1 Test cases structure and setup

All test cases are based on making an HTTP request to an application and validating the result. Test cases were designed to reduce the amount of operations caused by exploitation to the necessary minimum. For that reason, the test cases do not perform any useful attacks. In order to achieve that the application included a number of elements that made the attacks easier to perform and verify. For example, the action classes for the application include a public `test()` method, that, if set, adds the additional `exploited=true` value to the application response structure. If set, that value is reflected in the standard JSON response generated by application APIs.

Note that additional code added to the application in order simplify the test cases do not affect Struts operation that is the subject of the experiment. The tests would be possible without introducing the additional code, just at higher cost and potentially involving more operations, often from Struts itself. Therefore, the additional code allows the testing of the worst case scenarios for runtime verification. Other, specific additions or setup steps are discussed in context of specific test cases.

In general the test cases are defined in perform following steps:

- test setup, such as setting a specific variable or clearing state (optional),
- HTTP request to perform an attack,
- verification whether the attack was successful, and

- optional cleanup.

The steps such as setup or cleanup were usually done by making a call to a separate J2EE `service` servlet operating outside of Struts, and not covered in the scope of runtime verification. The servlet performs operations such as checking variable value or setting up a class loader. It is also used for cleanup operations, such as resetting value to its default state. Those cleanup operations are not included in below test case definitions.

A.2 Test cases

CVE-2012-0393: Unrestricted access to public constructors

Setup For this test we implemented a `TestClass` with a public constructor. The class includes a static field `constructorCalled` that is set on constructor invocation and publically accessible.

Request `/api/post?text=x[new tests.TestClass()]`

Check Variable `TestClass.constructorCalled` is set.

To improve the readability the request paths and parameters in the text are encoded according to RFC 3986. For example, in this test case, the actual request URI is `/api/add?text=x%5bnew%20tests.TestClass()%5d`.

CVE-2012-4387: No restriction for OGNL parameter length

Request `/api/add?text=a&⟨500-char-parameter-name⟩=z`

Check Request fails due to invalid parameter name. In fixed versions the parameter is ignored and request is successful.

CVE-2013-1965: Improper handling of parameters during a redirect

Setup An application included a parameter that, when set, makes struts action to complete with a 'redirect' return value. `struts.xml` application configuration specifies that for that return value an HTTP redirect should be sent.

Request `/api/post?text=${test()}&redirect=true`

Check JSON response includes `exploited: true`.

CVE-2013-1966: Improper OGNL parameter handling in '`url`' tag

Setup Application HTML includes a tag ⟨`s:url action="SearchAction" includeParams="all"`⟩. Application includes additional `testSetStatus()` method that sets application `status` variable that is rendered in JSP view inside a META tag to `exploited`.

Request `/ui/main?z=${testSetStatus()}`

Check HTML response includes ⟨`meta name='status' content='exploited'`⟩ string.

CVE-2013-2115: incomplete fix for CVE-2013-1966

Setup As in CVE-2013-1966

Request `/ui/main?z=1${testSetStatus()}`

Check As in CVE-2013-1966

CVE-2013-2134: Improper handling of OGNL in request wildcard matching

Setup The application configuration includes a wildcard match for unknonw API names to render standard JSON response indicating an error

Request `/api/${test()}`

Check JSON response includes `exploited: true`

CVE-2013-2135: Double-evaluation of OGNL parameters

Setup The application `struts.xml` configuration includes a directive to pass the value of `text` variable to the JSP view for JSON processing.

Request `/api/post?text=${%{test()}}`

Check JSON response includes `exploited: true`

CVE-2013-2248: Unrestricted redirect to user-specified location

Request `/api/post?redirect:http://example.net/`

Check Response includes 302 code and `Location` header with value `http://example.net/`.

CVE-2013-2251: Improper handling of OGNL in the parameter prefixes

Request `/api/post?action:${test()}`

Check JSON response includes `exploited: true`

CVE-2013-4310: Bypassing external access controls using '`action:`' prefix

Setup Discussed in detail in Section 8.4.2. Application includes a J2EE filter to prevent access to `/api/private` API implementing an action that returns `private: true` in its JSON response.

Request `/api/post?action:private`

Check JSON response includes `private: true`

CVE-2013-4316: Unrestricted Dynamic Method Invocation enabled by default

Request `/api/post!test`

Check JSON response includes `exploited: true`

CVE-2014-0094: Unrestricted access to '`class`' property

Setup A custom implementation extending Tomcat Java Class Loader used by the application, which includes an `exploited` field with associated getter and setter.

Request `/api/post?text=a&class.classLoader.exploited=1`

Check The class loader `exploited` field is set to 1.

CVE-2014-0112: Incomplete fix for CVE-2014-0094

Setup As in CVE-2014-0094.

Request `/api/post?text=a&Class.classLoader.exploited=1`

Check As in CVE-2014-0094.

CVE-2014-0113: Access to '`class`' parameter using cookies

Setup As in CVE-2014-0094.

Request `/api/post?text=a` including an HTTP header `Cookie: class.classLoader.exploited=1`.

Check As in CVE-2014-0094.

CVE-2014-0116: Access to context variables using cookies

Setup Application includes a custom implementation of `HTTPServletResponse` including a static `exploited` field.

Request `/api/post?text=a` including an HTTP header `Cookie: response.exploited=1`

Check `HTTPServletResponse.exploited` is set to 1.

CVE-2014-7809: Predictable CSRF token

Setup A specific action is configured in to be protected using Struts CSRF protection. The attack normally involves calculating CSRF token using a previous value. However, in this test case tha actual value is obtained using `service` servlet. The fact that the value is not actually calculated does not affect the outcome of the test.

Request Call to the action including the token.

Check Request not rejected by CSRF protection.

CVE-2015-1831: Parameter restrictions accidentally disabled in the default configuration

Setup As in CVE-2014-0116.

Request `/api/post?text=a&response.exploited=1`

Check As in CVE-2014-0116.

CVE-2015-5209: Parameter restrictions bypassed using '`top`' parameter

Request `/api/post?text=a&top.session[user].id=frank`

Check Application session's `user` variable's `id` property is `frank`.

Appendix B

False Positive Test Cases

Table B.1 includes the results of false positive tests. These test cases are imple-

Test case	2.3.1	2.3.4	2.3.14	2.3.14.1	2.3.14.2	2.3.15	2.3.15.1	2.3.16	2.3.16.1	2.3.16.2	2.3.16.3	2.3.20	2.3.24	2.3.24.1
UnkwnonAction	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
PostAction	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
ViewAcion	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
NoParameters	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
MissingParameter	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
MissingParameter2	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
WrongParameter	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
WrongParameter2	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
ActionPrefix	+−	+−	+−	+−	+−	+−	+−	−−	−−	−−	−−	−−	−−	−−
WrongActionPrefix	+−	+−	+−	+−	+−	+−	+−	−−	−−	−−	−−	−−	−−	−−
ValidRedirect	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
GoodCookie	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
BadCookieName	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
BadCookieValue	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
NoURLBadCookie	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
PrivateAction	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
UIAction	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
ExtraUrlParameter	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−
UIWrongPath	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−	+−

Note that some outcomes were identical for different versions of Struts, and in the interest of space: version 2.3.4 covers 2.3.1.1, 2.3.1.2 and 2.3.3; 2.3.14 covers 2.3.4.1, 2.3.7, 2.3.8 and 2.3.12; 2.3.15 covers 2.3.14.3, and 2.3.16 covers 2.3.15.2 and 2.3.15.3; 2.3.24 covers 2.3.20.1.

Table B.1: False positive test cases outcomes on different versions of Struts

mented to exercise Struts in operations similar to the ones used by vulnerabilities, such as making a request with a parameter that does not exist, or incorrect action name, but without actually exploiting the vulnerability. Each outcome specifies whether the test was successful and if any anomalies were detected during the test. For example outcome +− indicates successful test with no anomalies detected.

The `UnknownAction`, `PostAction` and `ViewAction` test cases make HTTP requests with URLs that map to no action in the application, `PostAction` and `ViewAction` respectively. The `NoParameters` test case makes `PostAction` request with no parameters, `MissingParameter` makes a request with missing required parameter and `MissingParameter2` with missing parameter but with the other valid parameter present. Similarly, `WrongParameter` makes a request with one incorrect parameter name and `WrongParameter2` with one parameter incorrect and the other correct. `ActionPrefix` test case makes a request with `action:` prefix, as discussed in Section 8.4.2, with correct action name, and `WrongActionPrefix` with an incorrect one. The `ValidRedirect` test case exercises additional action in the application that results in an (intended) redirect to a third-party website. The three Cookie-related test cases exercise usage of cookies with a correct name and cookies with incorrect (not whitelisted) name, and value. The `PrivateAction` is the test case specific to CVE-2013-4310, making an authenticated call (allowed by J2EE filter) to the private API. `UIAction` makes a casual usage of the UI, `UIWrongPath` the UI action with incorrect path and `ExtraURLParameter` includes additional parameter to the UI request, relevant to CVE-2013-1966.

The outcome of each test case is specific to the action. Often it is just verifying that the action succeeded or an expected error was returned.

Index

behavioral norms model, 47

dark side of the code, **8**, 126

Event, 47
event equivalence
    attribute based $\sim_A$, 56
    generic $\sim$, 50
event scope
    attribute scoping @, 55
    generic, 48

Jaccard coefficient $\mathcal{J}(n, m)$, 62

microblog application, 9

n-grams, 34, **60**
Norm, 52
norm set similarity $\mathcal{M}_T^O(l, t)$, 62
norms for log $\mathcal{N}_T^O(l)$, 61

Strand, 50
Struts
    interceptors, 119
      cookie, 125
      parameters, 119
    OGNL, 118
    operation, 10, 116

Trace, 48
trace equivalence
    attribute-based $\approx_A$, 57
    generic $\approx$, 52

References

[1] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Comput. Netw.*, 31(9):805–822, April 1999.

[2] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Department of Computer Engineering, Chalmers University of Technology, 2000.

[3] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[4] D. Gollmann. Software security – the dangers of abstraction. In *The Future of Identity in the Information Society*, volume 298 of *IFIP Advances in Information and Communication Technology*, pages 1–12. Springer Berlin Heidelberg, 2009.

[5] H. H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 1(4):83–86, July 2003.

[6] OWASP Foundation. OWASP Top 10 2013, 2013. `https://www.owasp.org/index.php/Top_10_2013`.

[7] SANS Institute. *CWE/SANS TOP 25 Most Dangerous Software Errors*, 3.0 edition, 2011.

[8] T. McLean. Critical vulnerabilities in JSON Web Token libraries. Blog post `https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html`, accessed 2 December 2016, 2015.

[9] P. Abate, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228 – 2240, 2012. Automated Software Evolution.

[10] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 2004.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[12] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 251–261, New York, NY, USA, 2003. ACM.

[13] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *IEEE Symposium on Security and Privacy*, pages 280–289, 1989.

[14] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.

[15] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 418–430, Washington, DC, USA, 2008. IEEE Computer Society.

[16] P. Raman. Jaspin: Javascript based anomaly detection of cross-site scripting attacks. Master's thesis, Carleton University, 2008.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, August 1998.

[18] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. *On the Detection of Anomalous System Call Arguments*, pages 326–343. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[19] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.

[20] O. Pieczul and S. N. Foley. Discovering emergent norms in security logs. In *2013 IEEE Conference on Communications and Network Security (CNS - SafeConfig)*, pages 438–445, 2013.

[21] O. Pieczul and S. N. Foley. Collaborating as normal: Detecting systemic anomalies in your partner. In *Security Protocols XXII: 22nd International*

Workshop, Cambridge, UK, March 19-21, 2014, Revised Selected Papers, pages 18–27. Springer International Publishing, 2014.

[22] O. Pieczul, S. N. Foley, and V. M. Rooney. I'm OK, You're OK, the System's OK: Normative security for systems. In *Proceedings of the 2014 Workshop on New Security Paradigms*, NSPW '14, pages 95–104, New York, NY, USA, 2014. ACM.

[23] O. Pieczul and S. N. Foley. The dark side of the code. In *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, pages 1–11. Springer International Publishing, 2015.

[24] O. Pieczul and S. N. Foley. The evolution of a security control. In *Security Protocols XXIV: 24th International Workshop, Brno, Czech Republic, Revised Selected Papers*. Springer International Publishing, 2016. (to appear).

[25] O. Pieczul and S. N. Foley. Runtime detection of zero-day vulnerability exploits in contemporary software systems. In *Data and Applications Security and Privacy XXX: 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*. Springer International Publishing, 2016.

[26] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 296–305, New York, NY, USA, 2014. ACM.

[27] G. J. Holzmann. Code inflation. *IEEE Software*, 32(2):10–13, Mar 2015.

[28] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[29] S. Foley. A non-functional approach to system integrity. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan 2003.

[30] P. Ryan. Mathematical models of computer security. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 1–62. Springer Berlin / Heidelberg, 2001.

[31] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich. Report on the NSF workshop on formal methods for security. Available at `https://arxiv.org/abs/1608.00678`., August 2016.

[32] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*, pages 118–127, 2001.

[33] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In *ACM CCS 07*, 2007.

[34] D. Davis. Compliance defects in public-key cryptography. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, SSYM'96, pages 17–17, Berkeley, CA, USA, 1996. USENIX Association.

[35] Carnegie Mellon University. *CERT Secure Coding Standards – VOID 2 MET21-J. Do not invoke equals() or hashCode() on URLs.* `https://www.securecoding.cert.org/confluence/x/5wHEAw`.

[36] D. Davidson. Ognl language guide, 2004.

[37] D. E. Denning. An intrusion-detection model. In *IEEE Symposium on Security and Privacy*, pages 118–133, 1986.

[38] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16 – 24, 2013.

[39] H. Debar, M. Dacier, and A. Wespi. A revised taxonomy for intrusion-detection systems. *Annales Des Télécommunications*, 55(7):361–378, 2000.

[40] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 59–66, 1988.

[41] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[42] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A highly-scalable software-based intrusion detection system. In

Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pages 317–328, New York, NY, USA, 2012. ACM.

[43] Snort. Snort subscriber rule set categories. Online `https://www.snort.org/rules_explanation`, retrieved 10 Nov 2016.

[44] E. Albin and N. C. Rowe. A realistic experimental comparison of the suricata and snort intrusion-detection systems. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 122–127, March 2012.

[45] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[46] H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *IEEE Symposium on Security and Privacy*, pages 316–326, 1991.

[47] P. Helman and G. E. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Trans. Software Eng.*, 19(9):886–901, 1993.

[48] D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Detecting unusual program behavior using the statistical component of the next-generation intrusion detection expert system (nides). Technical report, SRI International. Computer Science Laboratory, 1995.

[49] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. Statistical techniques for online anomaly detection in data centers. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 385–392, May 2011.

[50] H. S. Teng, K. Chen, and S. C. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 278–284, May 1990.

[51] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.

[52] Z. Yu, J. J. P. Tsai, and T. Weigert. An automatically tuning intrusion detection system. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):373–384, April 2007.

[53] H. Inoue and S. Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the 2002 Workshop on New Security Paradigms*, NSPW '02, pages 52–60, New York, NY, USA, 2002. ACM.

[54] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[55] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, RAID '00, pages 110–129, London, UK, UK, 2000. Springer-Verlag.

[56] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the 2000 Workshop on New Security Paradigms*, NSPW '00, pages 101–110, New York, NY, USA, 2000. ACM.

[57] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st Conference on Workshop on Intrusion Detection and Network Monitoring - Volume 1*, ID'99, pages 6–6, Berkeley, CA, USA, 1999. USENIX Association.

[58] W. Lee, S. J. Stolfo, and P. K. Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection, 1997.

[59] K. M. C. Tan and R. A. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 188–201, 2002.

[60] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, September 2004.

[61] W. van der Aalst, K. M. van Hee, J. M. E. M. van der Werf, and M. Verdonk. Auditing 2.0: Using process mining to support tomorrow's auditor. *IEEE Computer*, 43(3):90–93, 2010.

[62] W. M. P. van der Aalst and A. K. A. de Medeiros. Process mining and security: Detecting anomalous process executions and checking process conformance. *Electron. Notes Theor. Comput. Sci.*, 121:3–21, February 2005.

[63] R. Accorsi and T. Stocker. On the exploitation of process mining for security audits: The conformance checking case. In *Proceedings of the 27th Annual*

ACM Symposium on Applied Computing, SAC '12, pages 1709–1716, New York, NY, USA, 2012. ACM.

[64] F. Bezerra and J. Wainer. Algorithms for anomaly detection of traces in logs of process aware information systems. *Inf. Syst.*, 38(1):33–44, March 2013.

[65] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 24(5):823–839, May 2012.

[66] H. Inoue and A. Somayaji. Lookahead pairs and full sequences: A tale of two anomaly detection methods. In *in: Proceedings of the 2nd Annual Symposium on Information Assurance (Academic track of the 10th NYS Cyber Security Conference*, pages 9–19, 2007.

[67] X. Shu, D. Yao, and N. Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 401–413, New York, NY, USA, 2015. ACM.

[68] M. Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

[69] P. Helman and J. Bhangoo. A statistically based system for prioritizing information exploration under uncertainty. *Trans. Sys. Man Cyber. Part A*, 27(4):449–466, July 1997.

[70] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. *Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits*, pages 54–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[71] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[72] J. E. Tapiador and J. A. Clark. Masquerade mimicry attack detection: A randomised approach. *Computers & Security*, 30(5):297 – 310, 2011. Advances in network and system security.

[73] H. G. Kayacik and A. N. Zincir-Heywood. On the contribution of preamble to information hiding in mimicry attacks. In *Advanced Information Net-*

working and Applications Workshops, 2007, AINAW '07. 21st International Conference on, volume 1, pages 632–638, May 2007.

[74] P. Li, H. Park, D. Gao, and J. Fu. Bridging the gap between data-flow and control-flow analysis for anomaly detection. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 392–401, Washington, DC, USA, 2008. IEEE Computer Society.

[75] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[76] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. Javamop: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1427–1430. IEEE Press, 2012.

[77] G. Creech and J. Hu. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Trans. Comp.*, 2014.

[78] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Trans. Dependable Secur. Comput.*, 2010.

[79] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[80] J. M. Spivey. *The Z notation - a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[81] J. M. Spivey. *The fuzz manual*. The Spivey Partnership, 1992.

[82] A. Luotonen. The common logfile format. `http://www.w3.org/pub/WWW/Daemon/User/Config/Logging.html`, 1995.

[83] M. R. Anderberg. *Cluster analysis for applications*. Probability and mathematical statistics. Academic Press, 1973.

[84] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[85] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[86] Securityfocus. Apache Struts CVE-2016-1182 Security Bypass Vulnerability. `http://www.securityfocus.com/bid/91067`, accessed 27 November 2016, 2016. List of affected products.

[87] IBM. *IBM Security AppScan Enterprise Manual – Recording QA automation test scripts with the Manual Explorer tool*, 9.0.3 edition, 2015.

[88] Apache Struts. *Config Browser Plugin*. Online, `https://struts.apache.org/docs/config-browser-plugin.html`.

[89] B. De Win, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In B. De Decker, F. Piessens, J. Smits, and E. Van Herreweghen, editors, *Advances in Network and Distributed Systems Security*, volume 78 of *IFIP International Federation for Information Processing*, pages 125–138. Springer US, 2002.

[90] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising aspectj. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 117–128, New York, NY, USA, 2005. ACM.

[91] A. Herzog and N. Shahmehri. Performance of the Java security manager. *Computers & Security*, 24(3):192–207, 2005.

[92] IBM. *IBM Security AppScan Enterprise Manual – Glass box scanning: testing application code during a scan*, 9.0.3 edition, 2015.

[93] F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: A study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, pages 195–208, Berlin, Heidelberg, 2011. Springer-Verlag.

[94] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis. Dismal code: Studying the evolution of security bugs. In *Proceedings of the LASER 2013 (LASER 2013)*, pages 37–48, Arlington, VA, 2013. USENIX.

[95] M. Kydyraliev. CVE-2010-1870: Struts2/XWork remote command execution. o0o Security Team blog, 2010. online; accessed 2016-01-21, `http://blog.o0o.nu/2010/07/cve-2010-1870-struts2xwork-remote.html`.

[96] M. Kydyraliev. CVE-2011-3923: Yet another Struts2 Remote Code Execution. o0o Security Team blog, 2011. online; accessed 2016-01-21, `http://blog.o0o.nu/2012/01/cve-2011-3923-yet-another-struts2.html`.

[97] J. Long. Struts 2 Session Tampering via SessionAware/RequestAware WW-3631. Code Secure blog, 2011. online; accessed 2016-01-21, `http://codesecure.blogspot.ca/2011/12/struts-2-session-tampering-via.html`.

[98] Z. Ashraf. Analysis of recent struts vulnerabilities in parameters and cookie interceptors, their impact and exploitation. IBM Security Intelligence portal, 2014. online; accessed 2015-05-21.

[99] J. Dahse. Multiple vulnerabilities in Apache Struts2 and property oriented programming with Java, 2011. online; accessed 2016-01-21.