Imperial College London
Department of Computing

Practical and Efficient Runtime Taint Tracking

Ioannis D. Papagiannis

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

Runtime taint tracking is a technique for controlling data propagation in applications. It is typically used to prevent disclosure of confidential information or to avoid application vulnerabilities. Taint tracking systems intercept application operations at runtime, associate meta-data with the data being processed and inspect the meta-data to detect unauthorised data propagation. To keep meta-data up-to-date, every attempt of the application to access and process data is intercepted. To ensure that all data propagation is monitored, different categories of data (e.g. confidential and public data) are kept isolated.

In practice, the interception of application operations and the isolation of different categories of data are hard to achieve. Existing applications, language interpreters and operating systems need to be re-engineered while keeping meta-data up-to-date incurs significant overhead at runtime. In this thesis we show that runtime taint tracking can be implemented with minimal changes to existing infrastructure and with reduced overhead compared to previous approaches. In other words, we suggest methods to achieve both *practical* and *efficient* runtime taint tracking.

Our key observation is that applications in specific domains are typically implemented in high-level languages and use a subset of the available language features. This facilitates the implementation of a taint tracking system because it needs to support only parts of a programming language and it may leverage features of the execution platform. This thesis explores three different applications domains. We start with event processing applications in Java, for which we introduce a novel solution to achieve isolation and a practical method to declare restrictions about data propagation. We then focus on securing PHP web applications. We show that if taint tracking is restricted to a small part of an application, the runtime overhead is significantly reduced without sacrificing effectiveness. Finally, we target accidental data disclosure in Ruby web applications. Ruby emerges as an ideal choice for a practical taint tracking system because it supports meta-programming facilities that simplify interception and isolation.

To Donatos,
for buying a PC instead of a typewriter
while being *absolutely clueless* about how to use one.

To Panagiota,
for being capable to make me *want*
to stop staring at computer screens.

Στον Δονάτο,
για το οτι αγόρασε υπολογιστή αντί για γραφομηχανή
χωρίς να έχει *απολύτως καμία ιδέα* για το πως θα τον χειριστεί.

Στην Παναγιώτα,
για την ικανότητά της να με κάνει *να θέλω*
να σταματήσω να κοιτάω μία οθόνη υπολογιστή.

Acknowledgements

First of all, I would like to thank my supervisor, Peter Pietzuch, for giving me the unique opportunity to work towards a doctorate degree and for providing his ongoing support over the years. He taught me what is important in research, how a research group is run and some invaluable practical skills. For all that I will always be grateful. I should also thank him for answering even the "stupidest" of my questions, never making me feel uncomfortable.

Second, I am grateful to Matteo Migliavacca, the research associate I collaborated with for the best part of my doctoral studies. Matteo took the loneliness out of research. He showed me how to ask the right questions. Our stimulating in-person meetings and our long chat sessions are amongst my best memories of the last four years.

I should also thank all the people in the SmartFlow project: David Eyers, Petr Hosek, Brian Shand, David Evans and Jean Bacon. They have all contributed to a great collaboration, and I consider myself privileged for having received their feedback on various occasions.

At Imperial College, I have been a member of the Large-Scale Distributed Systems group. Marco, Thom, Andreas, Nick, Raul, Mouna, you have all been an exceptional company at the local pub—even though I can't say the same for myself. I reserve here a great "thank you" to Evangelia Kalyvianaki, for listening to my whining without her complaining. That was quite an accomplishment.

My research was funded by the Engineering and Physical Sciences Research Council (EPSRC) via the EP/F042469 and EP/F044216 research grants.

Finally, I would like to thank my family, Donatos, Athina, Sophia and Panagiota, for their encouragement and motivation. Most importantly, for enabling and allowing me to spend so many hours studying and interacting with computers. This is what *really* made everything possible.

Τέλος, θα ήθελα να ευχαριστήσω την οικογενειά μου, τον Δονάτο, την Αθηνά, την Σοφία και την Παναγιώτα, για την υποστηριξή τους. Κυρίως, θα ήθελα να τους ευχαριστήσω και για το οτι έκαναν εφικτό αλλά και για το οτι μου επέτρεψαν να αφιερώσω τόσο χρόνο στο διάβασμα και στον υπολογιστή. Αυτός ειναι *ο πραγματικός λόγος* που έφτασα ως εδώ.

— London, September 2012

Declaration

This thesis presents my work in the Department of Computing at Imperial College London between October 2008 and September 2012.

Parts of the work were done in collaboration with other researchers.

- **Chapter 3.** The motivation for the work as well as the design of the DEFC model were the result of my interaction with Matteo Migliavacca. The implementation effort for achieving isolation between processing units was led by Matteo Migliavacca and Brian Shand.
- **Chapter 4.** The semantics of DPL and the translation process to DEFC were a collaborative effort between myself and Matteo Migliavacca. Brian Shand contributed the policy requirements for the healthcare scenario.
- **Chapter 6.** The overall design of SafeWeb and of the MDT portal application was the result of discussions between myself, Matteo Migliavacca, Petr Hosek and Brian Shand. The implementation effort for the event processing back-end was led by Petr Hosek, and I led the implementation effort for the taint tracking library using Ruby's meta-programming facilities.

I declare that the work presented in this thesis is my own, except where acknowledged above.

Publications

[MPE+10b] Matteo Migliavacca, Ioannis Papagiannis, David Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. DEFCon: High-Performance Event Processing with Information Security. In *Annual Technical Conference (ATC)*, Boston, MA, 2010. USENIX.

[MPE+10a] Matteo Migliavacca, Ioannis Papagiannis, David Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Distributed Middleware Enforcement of Event Flow Security Policy. In *Middleware*, Bangalore, India, 2010. ACM/IFIP/USENIX.

[PMP11] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[HMP+11] Petr Hosek, Matteo Migliavacca, Ioannis Papagiannis, David Eyers, David Evans, Brian Shand, Jean Bacon, and Peter Pietzuch. SafeWeb: A Middleware for Securing Ruby-based Web Applications. In *Middleware*, Lisbon, Portugal, 2011. ACM/IFIP/USENIX.

Contents

1 Introduction **23**

1.1 Runtime taint tracking . . . 24
1.2 Application scenarios . . . 24
1.2.1 Tracking data flow in event processing . . . 24
1.2.2 Preventing injection attacks in web applications . . . 25
1.2.3 Guaranteeing patient data confidentiality in the NHS . . . 25
1.3 Problem statement . . . 26
1.4 Research contributions . . . 27
1.5 Dissertation outline . . . 29

2 Background **31**

2.1 Controlling data flow in applications . . . 31
2.1.1 Injection vulnerabilities in web applications . . . 32
2.1.2 Security requirements in event processing . . . 36
2.2 Static methods for data flow analysis and enforcement . . . 39
2.2.1 Taint analysis . . . 40
2.2.2 Symbolic execution . . . 45
2.2.3 Security-typed languages . . . 48
2.3 Runtime taint tracking . . . 51
2.3.1 A model for runtime taint tracking . . . 51
2.3.2 Covert channels . . . 54
2.3.3 Runtime taint tracking systems . . . 57
2.4 Summary . . . 76

3 Taint Tracking for High-Performance Event Processing **79**

3.1 Requirements . . . 80
3.1.1 Performance requirements (latency and throughput) . . . 81
3.1.2 Security requirements (threat model) . . . 82
3.2 Decentralised Event Flow Control . . . 83
3.2.1 Anatomy of an event . . . 84
3.2.2 Security labels . . . 84
3.2.3 Tag and delegation privileges . . . 86
3.2.4 Input/output labels . . . 87

3.2.5 Dynamic privilege propagation . . . 88
3.2.6 Partial event processing . . . 89
3.3 Unit isolation in Java . . . 89
3.3.1 Previous proposals for isolation in Java . . . 91
3.3.2 Isolation methodology in DEFCon . . . 92
3.3.3 Restricting channels due to static fields and native methods . . . 93
3.3.4 Restricting synchronisation channels . . . 97
3.4 Implementation of DEFCon . . . 98
3.4.1 API for event generation and dispatch . . . 100
3.4.2 Event and label matching . . . 102
3.4.3 Managing unit life-cycle . . . 104
3.5 Evaluation . . . 105
3.5.1 Financial trading scenario . . . 105
3.5.2 Experimental results . . . 109
3.5.3 Discussion . . . 113
3.6 Summary . . . 114

4 Event Flow Security Policy 115
4.1 Requirements . . . 115
4.2 The DEFCon Policy Language . . . 118
4.2.1 Event flow constraints . . . 118
4.2.2 Vertical flow separation . . . 120
4.2.3 Horizontal flow separation . . . 121
4.2.4 Parameterisation . . . 122
4.2.5 Event processing contexts . . . 123
4.3 Policy enforcement . . . 124
4.3.1 Distributed policy management . . . 125
4.3.2 Translation to DEFC . . . 127
4.3.3 Inter-engine communication . . . 128
4.4 Evaluation . . . 129
4.4.1 Healthcare case study . . . 129
4.4.2 Policy enforcement with DEFC . . . 131
4.5 Summary . . . 132

5 Partial Taint Tracking for Protection against Injection Attacks 135
5.1 The case for partial taint tracking . . . 136
5.2 Overview . . . 136
5.3 Taint meta-data representation . . . 138
5.3.1 Taint categories . . . 138
5.3.2 Storing taint meta-data . . . 140
5.4 Taint tracking transformations . . . 141
5.4.1 Data entry points . . . 142

5.4.2 Taint propagation . . . 142
5.4.3 Guarded sinks . . . 144
5.5 Partial taint tracking . . . 144
5.5.1 Detecting vulnerabilities in presence of non-tracking code . . . 145
5.5.2 Compatibility transformations . . . 146
5.6 Evaluation . . . 149
5.6.1 Securing Wordpress . . . 149
5.6.2 Security . . . 150
5.6.3 Performance . . . 152
5.6.4 Discussion . . . 154
5.7 Summary . . . 155

6 Protecting Data Confidentiality in Ruby-based Web Applications 157
6.1 Requirements . . . 158
6.1.1 The case for a data quality assurance application in ECRIC . . . 158
6.1.2 Practical runtime taint tracking for web applications . . . 160
6.2 End-to-end tracking of confidential data . . . 161
6.3 Event processing back-end . . . 163
6.3.1 Taint tracking policy . . . 163
6.3.2 Event dispatch . . . 164
6.3.3 Controlling unit execution . . . 165
6.4 Web front-end . . . 167
6.5 Evaluation . . . 169
6.5.1 Implementation of the MDT portal application . . . 170
6.5.2 Security properties . . . 171
6.5.3 Performance overhead . . . 174
6.6 Summary . . . 175

7 Conclusion 177
7.1 Thesis summary . . . 178
7.2 Lessons learned . . . 180
7.3 Limitations in runtime taint tracking system design . . . 181
7.4 Future work . . . 184

Bibliography 187

List of Figures

2.1 Historical percentage of injection vulnerabilities in the CVE database 33
2.2 A multi-domain event processing scenario in healthcare 37
2.3 Exploration of the program from Listing 2.2 using symbolic execution 46
2.4 Overview of a generic runtime taint tracking system 53
2.5 Exploiting implicit taint meta-data changes to leak confidential information. . 55

3.1 An example of an event processing system for stock trading 80
3.2 DEFCon as a runtime taint tracking system 83
3.3 A DEFCon event with multiple named parts 85
3.4 Unit labels and privileges in DEFC 88
3.5 Isolation enforcement between units in DEFCon 93
3.6 Overview of the DEFCon architecture 99
3.7 Label-aware publish/subscribe matching in DEFCon's Event Dispatcher . . 103
3.8 A stock trading platform in DEFCon 107
3.9 Maximum supported event throughput in DEFCon 110
3.10 Maximum supported event throughput in Marketcetera 110
3.11 Event processing latency in DEFCon 111
3.12 Breakdown of event processing latency in Marketcetera 111
3.13 Amount of used memory in DEFCon 112

4.1 Specifying event flow policy within a bank 116
4.2 Vertical and horizontal flow separation in DPL 121
4.3 Overview of the DEFCon architecture with DPL support 125
4.4 A multi-domain healthcare scenario for the evaluation of DPL 129

5.1 Overview of partial taint tracking using PHP Aspis 137
5.2 PHP Aspis as a runtime taint tracking system 138
5.3 Example of an XSS vulnerability manifesting in non-tracking code 145

6.1 Overview of the SafeWeb architecture 162
6.2 SafeWeb as a runtime taint tracking system 163
6.3 Isolation of units and callbacks performed by the event processing engine . . 166
6.4 Variable-level taint tracking granularity in the SafeWeb web front-end . . . 168
6.5 Deployment of the MDT portal application at ECRIC 170
6.6 Processing latency within the MDT portal application 174

List of Tables

2.1 Classification of vulnerabilities for Wordpress and Drupal 34
2.2 Overview of design choices from taint tracking systems 54

3.1 An overview of the DEFCon API available to event processing units 101

5.1 Excerpt of the definition of an XSS taint category in PHP Aspis 139
5.2 Representation of taint meta-data for a single taint category in PHP Aspis . 140
5.3 Augmenting values with taint meta-data in PHP Aspis 141
5.4 Example source code transformations in PHP Aspis 142
5.5 Injection vulnerabilities in Wordpress plugins 151
5.6 Performance overhead of PHP Aspis in terms of page generation time 153

List of Listings

2.1 An example of string analysis in a PHP program 43
2.2 An example PHP program to demonstrate symbolic execution 45
2.3 Control and implicit flow from variable i to variable j 56
2.4 A Resin policy object 67
2.5 Control flow in Javascript 70

4.1 Extract of the DPL policy specification for the healthcare scenario 130

6.1 An example of a unit in SafeWeb 164
6.2 An example of a rule in the web front-end of the MDT portal application . . 172
6.3 An access control check used by the MDT portal application 173

Chapter 1

Introduction

Runtime taint tracking [SAB10] is a simple, powerful and popular technique to track data propagation in applications. It has been frequently used in the domains of security and privacy: confidential data protection [VEK+07], malicious attack prevention [YWZK09] and malware analysis [ESKK12] are all well-known use cases. Runtime taint tracking operates by compartmentalising an application into multiple isolated components that handle data of similar sensitivity. It then intercepts all operations that propagate data between components.

Despite being popular amongst researchers, taint tracking systems are rarely available or used outside of academia (Perl's taint mode [Doc12] is the most notable exception). We argue that the main reason behind limited popularity is that current state-of-the-art taint tracking systems are seldom both *practical* and *efficient*. To achieve component isolation, researchers have suggested novel execution platforms such as operating systems [ZBWKM06] and modified interpreters [YWZK09] but these are hard to adopt in practice. Others have suggested to perform taint tracking by transparently modifying application code [XBS06] but this typically leads to high performance overhead at runtime [KGJK12].

This thesis demonstrates three different approaches to build taint tracking systems that are both practical and efficient. First, we show that by combining extensive analysis of the Java execution platform with an efficient and domain-specific model for taint tracking, we avoid extensive, non-portable platform modifications that make previous systems hard to adopt. Second, we introduce *partial* taint tracking; an attempt to perform taint tracking only in a subset of an application's codebase. We demonstrate that partial taint tracking leads to a significant reduction of performance overhead at runtime without sacrificing the effectiveness of the technique. Third, we identify and leverage specific features of the Ruby programming language that facilitate taint tracking. We design a practical such system in Ruby, and we show how it can be used to improve security in the context of the National Health Service (NHS).

The following paragraphs introduce runtime taint tracking, underline the limitations of existing systems and summarise the contributions of this work towards a practical and efficient design for runtime taint tracking systems.

1.1 Runtime taint tracking

Runtime taint tracking is a simple data-centric technique to reason about data propagation (also referred to as *data flow*) in applications. Conceptually, the core idea is to associate *meta-data* with the data processed by an application and then use that meta-data to monitor or constrain the application's actions. The type of such meta-data, the methods used to keep the meta-data updated and how the meta-data are used by the taint tracking system are design parameters that vary greatly between different systems. This thesis collectively refers to the design decisions for the above parameters as the *taint tracking policy* of the system. The objective of a taint tracking system is to capture high level data propagation requirements from the user (for example, whether output data from a particular component should contain confidential information), adjust its taint tracking policy to cater for these requirements, and finally monitor or constrain the application's runtime behaviour.

The main advantage of runtime taint tracking is that it is *precise*. It reasons about an application while the application executes. The taint tracking system has access to the same information available to the application and it can closely monitor the application's actions. This is in contrast to similar *taint analysis* systems that perform static data flow analysis [ALSU07] in application source code (presented in detail in Section 2.2.1). Runtime taint tracking is also *simple*: instead of reasoning about *all* potential executions, it focuses on specific executions, one at a time. This renders the resulting systems easier to use than, for example, statically-typed languages (covered in Section 2.2.3), which encode data propagation requirements as part of the language's type system.

1.2 Application scenarios

In order to understand domains, in which runtime taint tracking systems operate, it is important to consider potential scenarios, in which a runtime taint tracking system can be applied. The following paragraphs introduce three representative scenarios that are further examined later in this thesis.

1.2.1 Tracking data flow in event processing

Event processing systems integrate infrastructure in domains such as finance and healthcare [Luc02]. They handle sensitive data on behalf of multiple clients, organisations and applications. Different principals communicate through an event processing system by exchanging event messages, or simply *events*. The event processing system propagates events and processes them. The processing that it performs should never convey confidential data of one client to another. This, however, is challenging to guarantee in practice if the processing combines data of different clients: a bug in the processing logic may disclose confidential data of one client to another.

A runtime taint tracking system can monitor the event processing system and track data flow across the events that it processes. If the event processing system combines information from multiple events to generate a new event, the taint tracking system will identify potential security policy violations, e.g. attempts to send the generated event to an unauthorised destination, and notify the relevant principals or prevent that operation.

To be effective, a taint tracking system designed for event processing must be efficient. Event processing applications, especially in finance, strive to minimise event processing latency—a feature that is seen as a competitive advantage in the marketplace [Duh09]. This is challenging to achieve because taint tracking introduces an overhead by updating meta-data at runtime.

1.2.2 Preventing injection attacks in web applications

The most common types of attacks on web applications involve code injection [Sec05]. Examples are Cross Site Scripting (XSS), SQL Injection (SQLI), Shell Injection and Eval Injection. Such attacks allow an attacker to execute code in either the web or database servers that host a web application or even in the browsers of a web application's clients. Injection attacks commonly exploit an application's trust in user-provided data. For example, if user-provided data are not checked for malicious content such as SQL statements while concatenating an SQL query (i.e. user data are not being *filtered* or *sanitised*), an attacker can trick the application into performing arbitrary, attacker-controlled actions in the back-end database.

A runtime taint tracking system can mark the user request data and monitor their propagation at the variable level in the web application. When the web application tries to use the request data to perform a potentially dangerous operation (e.g. issue an SQL query), the taint tracking system can inspect the user-supplied values and automatically sanitise them.

To be effective, a taint tracking system designed to prevent injection attacks should adapt to the needs of different applications. If a particular web application considers a specific class of user input as safe (e.g. because it originates from an administrator account), the taint tracking system should not interfere with the web application's operation. This means that a single taint tracking policy that is enforced across all web applications would be inadequate.

1.2.3 Guaranteeing patient data confidentiality in the NHS

Government agencies in the UK and elsewhere increasingly collect confidential data. Organisations in the NHS, for example, accumulate data sets about diagnosis and ongoing patient treatment for particular conditions[1]. This introduces an important challenge for such organisations: all applications that provide access to their data must guarantee data confidentiality, or must otherwise not be used outside a secure environment. However, building complex modern applications involves multiple developers who may have varying levels

[1]The Eastern Cancer Registration and Information Centre (ECRIC), `http://ecric.nhs.uk/`, last accessed: 5/9/2012

of understanding of an application's security requirements. A single implementation error is enough to expose confidential user data. As a result, applications in the NHS are subject to independent manual audit for security to prevent, amongst other threats, confidential user data disclosure [HMP+11].

A runtime taint tracking system can facilitate the development of new applications by reducing the need for bespoke security audits. When sensitive user data leave an organisation's central database, the taint tracking system can mark them as confidential. It can then monitor any application that processes that data and even enforce that data derived from the original data (which may also be confidential) are only disclosed to their intended recipients.

To be effective, a taint tracking system that targets government organisations should introduce minimal changes to existing infrastructure. Since the confidentiality of data depends on the correctness of the taint tracking system, the system itself will have to undergo security audits. A smaller and self-contained implementation of taint tracking is therefore more likely to be accepted when compared, for example, to an implementation based on extensive, third-party modifications to a language interpreter.

1.3 Problem statement

Runtime taint tracking systems benefit end users, developers and application administrators. However, outside the security research community, runtime taint tracking is not popular. This thesis argues that *to make runtime taint tracking a practical tool for monitoring and controlling data flow, we need designs that: (1) significantly minimise the performance overhead of the technique at runtime; (2) allow users to adapt the taint tracking policy of the system to the data propagation requirements of their application domain; and (3) do not require fundamental changes to existing infrastructure, applications and programming models.*

These three requirements are hard to achieve at the same time. To minimise the performance overhead of the technique, a taint tracking system could group application components with similar requirements and maintain simple meta-data for each such group [EKV+05]. This typically requires support from the execution platform to isolate application components and input from developers to divide an application into multiple components. Similarly, the ability of a user to adapt the taint tracking policy implies that the meta-data that a taint tracking system maintains are configurable. This can significantly reduce performance compared to a system based on simple, hard-encoded meta-data [CW09].

There have been several attempts to design taint tracking systems (see Section 2.3 for a detailed discussion). A common approach is to create designs with existing applications in mind. Typically an open-source execution platform (e.g. a language interpreter) is modified to propagate a specific type of meta-data transparently and to enforce a (rarely configurable) taint tracking policy. This approach minimises the changes required in application code. Examples include Java [CW09], PHP [PB05], Python [YWZK09] and Javascript [DG10]. The advantages of these systems are increased application compatibility and often high

performance. Yet, research-oriented prototypes rarely support all language features and are not kept up-to-date with newer language releases. The resulting taint tracking systems are thus not attractive because users prefer to maintain compatibility with official releases of a platform.

A second approach is to incorporate taint tracking as a basic feature of an execution platform and expose it to each application. Application developers gain access to the taint tracking system and have to adapt their applications to leverage its features. Several new operating systems have followed this approach, most notably Asbestos [EKV+05] and HiStar [ZBWKM06]. One of the fundamental requirements for these systems, the one that requires an extensive redesign of the execution platform, is support for isolation between different application components. An additional challenge is how to incorporate taint tracking so that it does not impact application performance significantly while it is generic enough to monitor data propagation for many different applications. This results in low-level application programming interfaces that are hard for developers to understand and use in practice.

1.4 Research contributions

This thesis introduces three different runtime taint tracking systems: DEFCon, PHP Aspis and SafeWeb. Each of these systems tracks data propagation in different types of applications and minimises the modifications required in the underlying infrastructure. We show that *by focusing on a specific class of applications*, a practical and efficient implementation of taint tracking is possible with few or no modifications to an execution platform.

DEFCon is a system to execute distributed event processing applications that handle sensitive data on behalf of multiple clients. DEFCon can host event processing on behalf of different clients, monitor their interactions, and control the propagation of data amongst them. It labels the events exchanged in the system and tracks their recipients. Labels are then used to limit the interactions of the recipients so that they cannot convey sensitive information to unauthorised destinations. The main contributions of DEFCon are:

Application-level isolation for Java. DEFCon isolates the event processing performed by different clients using modified Java Development Kit (JDK) libraries, in which statically-chosen interceptors carry out runtime checks at carefully selected code-paths. Since DEFCon only targets the specific class of event processing applications, it forgoes language features that would otherwise introduce significant challenges to achieve isolation for. In contrast to systems that target generic applications and introduce extensive redesigns [EKV+05, ZBWKM06], the process to produce the modified JDK libraries requires low manual effort and can easily be repeated for future JDK releases.

Decentralised Event Flow Control. *Decentralised Event Flow Control (DEFC)* is an efficient taint tracking policy that enables application developers and administrators to specify arbitrary data propagation requirements between components of their event

processing applications. DEFC involves *tags*, i.e. unique, opaque bit-strings that are attached to events and are updated at runtime as events are processed. DEFC tags are used by DEFCon to reason about and control data propagation without sacrificing performance.

High-level policy specification. The DEFCon *Policy Language (DPL)* is a language to express restrictions about data propagation between application components that are under the control of different administrative domains and may be instantiated in different DEFCon installations. With DPL, DEFCon provides a practical high-level language to specify data propagation requirements while the underlying taint tracking system still relies on low-level tags for efficient enforcement. A data propagation policy (also referred to as a *data flow policy*) expressed in DPL can be translated to DEFC tags.

PHP Aspis is a tool for protecting existing PHP web applications from injection attacks such as XSS and SQL Injection. It augments PHP values to include meta-data and transforms PHP statements to operate in the presence of meta-data and propagate these correctly. The meta-data are used to automatically sanitise user-provided, untrusted values and transparently prevent injection attacks. The main contributions of PHP Aspis are:

Partial taint tracking. PHP Aspis introduces *partial* taint tracking, which limits taint tracking only to those functions of the web application, in which vulnerabilities are more likely to occur. Partial taint tracking effectively captures the different levels of trust placed in different parts of web applications. It significantly improves performance because taint tracking does not occur in large parts of the application codebase.

PHP source code transformations. PHP Aspis does not require modifications to the PHP interpreter or to the web server. Instead, it performs taint tracking only using *source-to-source transformations*. The transformations maintain the application's structure in terms of classes, methods and files. This facilitates interoperation with parts of the application, in which taint tracking is not activated. PHP Aspis supports most PHP language features—including dynamic code generation—and can effectively protect existing web applications from real world exploits without the need of custom PHP interpreters.

Configurable taint categories. PHP Aspis can adapt to the data sanitisation requirements of different web applications. The meta-data that it maintains, their propagation and the actions that PHP Aspis takes based upon them (i.e. the taint tracking policy) are all configurable by the application administrator. PHP Aspis monitors the sanitisation efforts of different applications and avoids false positives.

SafeWeb is a taint tracking system that guarantees data confidentiality in web applications. SafeWeb intercepts application access to an organisation's main database. Individual web applications receive sensitive data in the form of labeled events and process them. SafeWeb

ensures that, after processing, labels are attached to data to reflect their confidentiality. It then enforces that no web application discloses data against the organisation's policy. The main contribution of SafeWeb is:

Compliance with Ruby. SafeWeb relies on standard features of the Ruby language and leverages well-maintained libraries. Ruby supports extensive meta-programming facilities that enable a taint tracking system to collect and update meta-data with minimal overhead at runtime and without any modifications to the language interpreter. This maximises compatibility and ensures that the use of SafeWeb does not introduce extensive maintenance overhead. The resulting system is practical and can reduce the need to audit new applications for security.

DEFCon, PHP Aspis and SafeWeb are evaluated in different scenarios and with different applications. DEFCon's effectiveness is demonstrated in the context of a stock trading application, in which a single server carries out processing on behalf of multiple clients. DEFCon guarantees confidentiality of trading, and it offers low latency for significantly more clients when compared to a similar Java trading system. PHP Aspis is used to secure an installation of Wordpress, a popular weblog platform. Partial taint tracking significantly reduces the overhead of the system at runtime while preventing most injection attacks that affected the Wordpress platform in the course of more than one year. Finally, SafeWeb is evaluated through the implementation of a web application for an organisation in the NHS. SafeWeb satisfies the organisation's security requirements and is practical because it reduces the need for extensive security audits when new applications are deployed.

1.5 Dissertation outline

The remainder of this dissertation is organised as follows:

Chapter 2 provides an overview of related systems that analyse and enforce data flow. It first motivates the need for such systems to improve the security of event processing applications and applications in the web. Subsequently, it examines static approaches and then presents runtime taint tracking as an alternative technique that is more precise. The chapter covers various taint tracking systems from the literature and suggests a model for runtime taint tracking systems to establish common terms and facilitate presentation.

Chapter 3 introduces DEFCon, an event processing system with support for taint tracking as part of its design. The chapter covers the low-level mechanism for specifying data flow policies and how these are later enforced. It finishes with a detailed evaluation emphasising the low overhead of taint tracking on event processing at runtime.

Chapter 4 completes the presentation of DEFCon by focusing on high-level data flow policies written in DPL and its support for distribution. The chapter identifies limitations when multiple administrative domains collectively specify data flow policy (e.g. the need for do-

mains to collaborate closely). It introduces specific features of DEFCon and DPL that facilitate data flow policy specification and its enforcement across domains.

Chapter 5 presents PHP Aspis, a system to secure web applications from injection attacks via partial taint tracking. The chapter starts with a strategy for enabling or disabling taint tracking in parts of an application's codebase and then devises a set of required transformations applied to application source code. The system is evaluated by securing an installation of Wordpress in the presence of plugins that are vulnerable to injection attacks.

Chapter 6 describes SafeWeb, a taint tracking system for Ruby web applications. The chapter presents the security requirements for new applications in an organisation within the NHS that handles sensitive patient data. The chapter then suggests runtime taint tracking to control data propagation end-to-end within the organisation's network. It finishes by evaluating the system's effectiveness against common types of vulnerabilities.

Chapter 7 concludes, summarising the work described in this thesis and outlining future research directions.

Chapter 2

Background

The lack of practical and efficient mechanisms to control data propagation impacts the security of applications in the web and in event processing. To provide *data flow guarantees* in applications, past research has suggested both static and dynamic approaches [JKK06b, ML00, ZBWKM06, KGJK12]. Static approaches analyse applications and reason about data flow without introducing runtime overhead [JKK06b, ML00]. Static approaches are therefore often preferable. However, the limitations of static approaches can be overcome at runtime, and runtime taint tracking emerges as a simple and effective technique for dynamic data flow analysis. Runtime taint tracking has been applied to a variety of problems, resulting in multiple designs that explore different directions and often have conflicting terminology.

This chapter in Section 2.1 discusses security challenges in the domain of web applications and event processing systems due to the lack of tools to analyse and enforce data flow. Section 2.2 presents an overview of static data flow analysis techniques. Section 2.3 focuses on different types of runtime taint tracking systems. To unify the presentation of taint tracking systems, the section proposes a generic model for runtime taint tracking that captures the available design options and defines common terms. This model is used throughout this thesis to present previous systems and to discuss the design choices in DEFCon, PHP Aspis and SafeWeb.

2.1 Controlling data flow in applications

An important challenge in application security is the control of data propagation. When users trust an application with their personal data, they expect that the data will not be released to any third party without authorisation. Similarly, when developers write code to process user data, they expect that only their code is executed and not code that originates from the user. In practice, applications contain bugs that violate these assumptions.

This section introduces two application domains, in which control of data flow is important for security. Section 2.1.1 describes injection vulnerabilities in web applications and shows

that they can be expressed as violations of expected data flow. It then introduces the concept of *data integrity* as a way to express data flow requirements. Section 2.1.2 discusses event processing applications. It underlines that existing security mechanisms in event processing systems cannot capture complex data flow requirements about data integrity or *data confidentiality*.

2.1.1 Injection vulnerabilities in web applications

Injection vulnerabilities are a persistent problem for web applications despite a recent increase in developer awareness. In 2010 alone, 23.9% of the total reported vulnerabilities to the Common Vulnerabilities and Exposures (CVE) database [MIT12] were classified as SQL Injection (SQLI) or Cross Site Scripting (XSS). Such vulnerabilities still remain because proposed solutions typically require manual tracking and filtering of user-generated data throughout the source code of an application. Yet, even a single developer error is enough to cause an injection vulnerability.

The rest of this section introduces standard application behaviour that injection attacks exploit and discusses sanitisation functions, i.e. the widely recommended method to prevent injection attacks. It then attributes injection attacks to the application's inability to enforce data flow and underlines the need for a tool to achieve protection automatically.

Injection vulnerabilities

Consider a weblog with a search field written in PHP. Typically, input to the search field results in a web request with the search term as a parameter:

```
http://goodsite.com/find?t=spaceship
```

A response of the web server to this request may contain the following fragment:

```
<p> The term ''spaceship'' cannot be found. </p>
```

The important element of the above response is that the user-submitted search term is included in the output. This can be exploited by an attacker to construct an XSS attack. The attacker first creates the following URL:

```
http://goodsite.com/find?t=<script%20src='attack.com/attack.js'/>
```

When a user clicks on this link, the following HTML fragment is generated by the web application at `goodsite.com`:

```
<p> The term ''<script src='attack.com/attack.js' />'' cannot be found. </p>
```

The victim's browser fetches the malicious Javascript code and executes it. Since the HTML document originated from `goodsite.com`, the browser gives to the downloaded script full

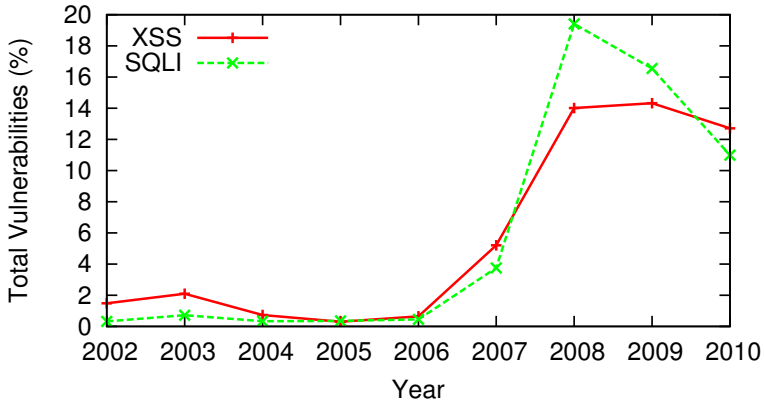

Figure 2.1: Historical percentage of XSS and SQL Injection vulnerabilities across all vulnerabilities reported to the CVE database.

access to the victim's web browser session (e.g. the ability to manipulate the page or to access any cookies set by `goodsite.com`). If the user has an account with `goodsite.com` and is already logged in, the malicious script can perform unverified operations on behalf of the user, e.g. it may erase all previous articles that the user has posted. In addition, the user's cookies can be sent to the attacker, and the attacker may then impersonate the user.

This type of attack is called a *Cross Site Scripting (XSS) attack* [CER00] because it involves loading a website while embedding a script from a second website, e.g. loading `goodsite.com` with a script from `attack.com`. The corresponding vulnerability, i.e. the weblog hosted by `goodsite.com` constructs its HTML responses by embedding the contents of incoming HTTP requests, is called an *XSS vulnerability*.

SQL Injection attacks are analogous: they take advantage of applications that rely on user input to form SQL queries. If the application constructs an SQL query by directly concatenating user-provided data, the attacker can influence the query semantics and perform arbitrary operations on the database [HOM08]. Similarly, Eval and Shell Injection attacks target PHP's `eval()` and `exec()` statements, respectively[1]. Since these statements execute arbitrary code at runtime, such attacks can compromise the host machine [PB05].

Figure 2.1 shows the percentage of reported SQLI and XSS vulnerabilities amongst applications in recent years, as classified by the CVE database [MIT12]. Note that the CVE database also contains vulnerability reports for problems and applications not specific to the web (e.g. memory corruption [CXN+05]). Overall, both XSS and SQLI continue to affect applications despite an increase in developer awareness.

Table 2.1 shows a classification of vulnerabilities for two popular open source PHP web applications, Wordpress[2] and Drupal[3]. PHP web applications do not suffer from traditional memory corruption vulnerabilities because memory management is performed by the lan-

[1]Many languages popular in web development provide similar functions to interpret application-generated code at runtime, e.g. Ruby provides the `Kernel::system` method and Javascript provides the `eval` function.
[2]The Wordpress project, `http://wordpress.org/`, last accessed: 7/9/2012
[3]Drupal Open Source CMS, `http://drupal.org/`, last accessed: 7/9/2012

Type	Number of Occurrences *Wordpress*	*Drupal*
Cross Site Scripting	9	28
SQL Injection	3	1
Information Leakage	1	0
Insufficient Access Control	1	9
Eval Injection	0	1
Cross Site Request Forgery	1	0

Table 2.1: Classification of vulnerabilities for Wordpress and Drupal as reported in the CVE database.

guage interpreter. As a result, code injection vulnerabilities are the most common type of vulnerabilities for both applications, with higher percentages compared to the vulnerability reports across all applications contained in the CVE (Figure 2.1). Additional vulnerability categories such as Insufficient Access Control (e.g. forgotten access control checks before specific operations) or Cross Site Request Forgery [Sec05] are significantly less frequent.

Prevention methods

Web applications have a genuine need to use user data in functions that may be exploited in injection attacks. In the weblog example, the application developers wish to inform the user about the exact search term that was not found in the local database. The traditional approach to prevent injection attacks is to allow user data to reach functions used in injection attacks *only if the data are properly sanitised or filtered.*

In the weblog example, a *sanitisation function* would translate all characters with special semantics in HTML to their display equivalents, e.g. "`<`" to "`<`" [WSA+11]. Sanitisation functions such as `htmlentities` or `escapeshellarg` in PHP are part of most languages in web development. After sanitisation, strings can safely be echoed to the client because they can no longer change the semantics of the output. SQL Injection filtering functions operate similarly but they also check and remove user-provided SQL keywords from the query [NTGG+05, YWZK09].

Unfortunately, sanitisation functions are difficult to apply in practice. Each function that can be used in an injection attack requires a different sanitisation function. For example, if the same string is echoed to the user and used as part of an SQL query, two different strings must be generated based on the original value. Applications should not filter input data centrally when they are received because there is no single data representation that is both meaningful and secure in all possible contexts. For example, the string "`WHERE`" is safe in HTML but not in an SQL query. Therefore, developers have to propagate the original user data and only sanitise them before they are used.

User-controlled sanitisation assumes that developers can effectively track the origin of data and enforce that user-generated data always pass through their respective sanitisation func-

tions. In practice, checks omitted by inexperienced developers or unforeseen interactions that result in unexpected data flow are likely to occur. One such example is to assume that a PHP script cannot be requested directly by an external user and therefore omit sanitisation. If, however, access to the script depends on the web server's configuration and the web server is not configured as expected, vulnerabilities are likely to exist.

Injection attacks as a data flow problem

XSS, SQL, Eval and Exec Injection attacks are *a result of a web application failing to enforce a particular data flow.* The data flow that the application should guarantee is that all user data are always sanitised before they reach the set of functions that can be used to perform injection attacks.

Another way to express the same data flow requirement is using the concept of *data integrity* [Bib77]. Data that originate from different sources should be considered of different "quality". For the problem of injection vulnerabilities, data from the web application are considered trusted (or of "high" integrity) while data from users are considered untrusted (or of "low" integrity). Thus, the data flow policy that should be enforced in web applications is that *only high-integrity data may reach the set of functions that can be used in injection attacks.*

Sanitisation functions relax the data flow requirement above by offering a way for low integrity user data to be used by the web application as high integrity, application-generated data. Sanitisation functions are thus considered *trusted to endorse* low integrity data and mark them as high integrity.

Web applications need a practical and efficient mechanism to enforce automatically this policy for *data flow integrity* and thus prevent injection vulnerabilities, independently of manual sanitisation efforts by developers. Manual security mechanisms have been shown to be ineffective to reduce the number of injection vulnerabilities in web applications [FW11]. The main problem with manual mechanisms is that developers either do not understand the security attacks and the protection mechanisms offered by a given framework or, more importantly, fail to use the provided mechanisms consistently everywhere in the application codebase.

An automated mechanism to prevent specifically SQL Injection are *prepared statements*[4]. Instead of creating SQL queries using string concatenation, an application defines static placeholder queries with parameters filled in at runtime. Parameters passed to the placeholders cannot change the semantics of the query, as its structure is determined in advance when the statement is prepared. By relying on prepared statements, the Drupal platform has exhibited few such exploits compared to Wordpress (see Table 2.1).

[4]PostgreSQL 8.1.23 Documentation—PREPARE, `http://www.postgresql.org/docs/8.1/static/sql-prepare.html`, last accessed: 7/9/2012

2.1.2 Security requirements in event processing

Data flow enforcement is also important in event processing. Event processing is a software paradigm for performing analysis and transformation of flows of event messages [Luc02]. Central to the paradigm are *event messages* (or simply *events*). An event represents the occurrence of a phenomenon, such as a stock tick in a financial application or a cancer biopsy request in a healthcare application. The "business logic" of an event processing application is organised in *event processing units* (or simply *units*). Units may actively emit events or reactively perform processing when an event is received. The result of event processing may be the modification of events received or new events emitted as output. Different formats for events are possible with a typical choice being a fixed structure such as key/value pairs. Event processing systems [TGP05] (e.g. Esper[5], Progress Apama[6] and IBM WebSphereMQ[7]) are used in fraud detection, finance and, in a corporate setting, for enterprise application integration and business process management [BEMP07].

Event processing has been shown to facilitate the design of systems that scale well [Arm07]. Scalability is achieved by decomposing business logic into multiple processing units that only share data through the exchange of events. This avoids the need for synchronisation, i.e. the main problem associated with shared-state concurrency. Event processing systems take advantage of multiple cores to execute processing units concurrently. Assuming that a large task can be performed by many processing units that exchange events and that these units do not introduce significant ordering constraints for their execution, event processing systems can scale linearly with the number of processor cores available.

Event processing also benefits systems that are amenable to distribution [Luc02]. Traditional paradigms for distributed systems, such as CORBA [OMG02] and Java RMI [Mic99], introduce tight coupling between systems by emulating the semantics of local procedure calls. With event processing instead, inter-connected systems exchange information by sending events. The sender and the recipient are decoupled because events are sent and received at different times. Participating systems are simplified and distribution is easier [MFP06] because the responsibility of dispatching events lies with the event processing system (also known as an *event-based middleware* [Pie04]).

The rest of this section introduces two applications of event processing, low latency trading and healthcare systems integration, and discusses how these applications can benefit from data flow enforcement.

Low latency trading

Low processing latency is a key requirement for modern stock trading systems. With the shift to algorithmic trading, the ability to react quickly to market changes affects financial

[5]Esper, `http://esper.codehaus.org/`, last accessed: 25/9/2012
[6]Progress Apama, `http://progress.com/apama`, last accessed: 25/9/2012
[7]IBM WebSphereMQ, `http://www.ibm.com/webspheremq`, last accessed: 25/9/2012

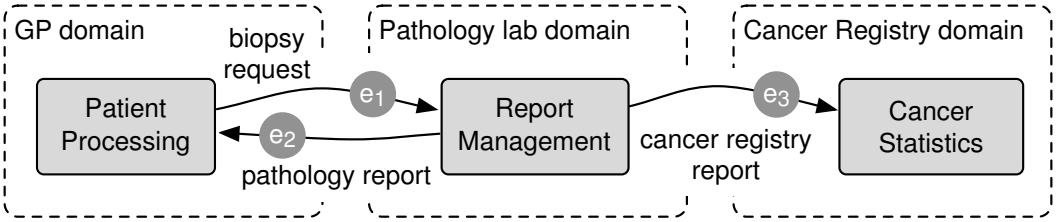

Figure 2.2: A multi-domain event processing scenario in healthcare. Events are exchanged between processing units belonging to multiple domains.

gains [Iat09]. It has been reported that reacting faster than the competition may translate to increased earnings of $0.01 per share, even for trades that other traders perform [Duh09].

To support algorithmic trading, stock exchanges provide appropriate interfaces and event flows. To achieve low latency, they charge for the service of having machines physically co-located in the same data centre as parts of the exchange [Fla07]. However, even with co-location within the same data centre rack, there is a minimum latency penalty due to inter-machine network communication. In addition, physical rack space in data centres close to exchanges is expensive and limited [Exc09].

The scalability offered by event processing can benefit applications with high message throughput and low latency requirements such as algorithmic trading. Instead of allocating the processing of traders that act for competing institutions to separate physical hosts, *traders can share a single, co-located machine*. This benefits traders because latency is reduced, co-location becomes more affordable and clients can trade with each other avoiding the exposure associated with the stock exchange [Zen09].

Hosting event processing on behalf of competing traders on the same machine, however, is challenging from a security perspective. The event processing system should never leak proprietary information regarding trading strategies, stock subscriptions and orders from one trader to another. Similarly, each trader's incoming stock tick feed must not be influenced by other traders in the same system—traders should always be able to take objective trading decisions. Finally, trading strategies (which may be contributed by different traders) should be unable to monitor the trading activity of their competition. All these requirements are hard to meet given the likelihood of software defects either in the event processing system or in event processing units exposing information. For such a co-location service to become viable, clients should be convinced about the integrity and confidentiality of their trading strategies.

Healthcare systems integration

In healthcare, data processing workflows involve distributed systems that span across multiple organisational *domains*. Figure 2.2 gives a representative example of the types of workflows found in the UK National Health Service (NHS). The figure illustrates domains (dashed regions) and application components inside these domains that exchange events.

The workflow is triggered when a patient visits their General Practitioner (GP) for a consultation. The GP takes a lump biopsy and forwards an electronic request with patient information to an NHS pathology laboratory asking for specimen analysis (message e_1). A pathologist performs the analysis and responds with a report to the GP (message e_2). The report is also forwarded to an NHS registry for cancer incidents (message e_3) but only if the analysis of the lump reveals evidence of cancer.

In the above scenario, event processing facilitates system integration because it naturally captures the existing distributed workflow. The GP, the pathology lab and the cancer registry form individual domains, which process events independently from each other. Three types of events capture the different messages exchanged: the biopsy requests (e_1), the pathology reports (e_2) and the cancer registry reports (e_3). Event processing units contributed by the different domains receive these events and implement the application's business logic: a patient processing unit (for each GP domain), a report management unit (for each pathology lab domain) and a cancer statistics unit (for the cancer registry).

For the above workflow security is important. The workflow involves events with confidential user data and their propagation is regulated by corresponding data protection legislation [NHS10]. Three security requirements must be enforced. First, only the requesting GP or a cancer registry may receive pathology reports. Second, cancer registries may only receive pathology reports about cancer cases. Third, only doctors in pathology labs may receive the confidential patient data included in biopsy requests. However, the three domains may handle security policy differently due to inconsistent interpretations at implementation time. Any information leaks will have serious consequences due to the sensitive nature of patient data.

Enforcing data flow requirements

Event processing systems should be able to enforce data security policies that specify system-wide, end-to-end *confidentiality* and *integrity* data flow requirements [BL73, Bib77]. For example, pathology reports should only be exposed to the GP that requested a biopsy (confidentiality requirement). The input data to a trading strategy should only be stock tick events provided by the stock exchange (integrity requirement).

Most access control schemes [Lam71] are not designed to offer confidentiality and integrity requirements end-to-end, i.e. across multiple event processing units. Typically, access control systems rely on *Policy Enforcement Points* (PEPs), i.e. specific points at which compliance with security policy is checked [RCFZ11]. The event processing system may help developers define and control PEPs by allowing them to manipulate the checks they perform. However, an important limitation exists: the access control system decides in advance whether to allow or deny an operation—i.e. without knowing how the data that are accessed by the operation will subsequently be used.

This limitation becomes evident in the confidentiality example above once implementation

errors are considered. A processing unit that is otherwise able to access pathology reports may still violate patient data confidentiality if, due to an implementation error, it does not correctly identify the receiving GP.

In contrast, confidentiality and integrity data flow guarantees should be enforced end-to-end by the event processing system, independently of the correctness of the implementation of processing units. The event processing system should track the flow of events between units and prevent event exchange when a violation of data confidentiality or integrity is detected; not only in advance before a unit receives input. In the example above, assume that the unit that accesses pathology reports is marked as having received a pathology report before accessing one. The event processing system will be in a position to safely deliver the pathology report to the unit as long as it can subsequently enforce that all output events of that unit will only be received by the corresponding GP (i.e. independently of the processing that the unit may perform).

2.2 Static methods for data flow analysis and enforcement

Static methods are popular for analysing and enforcing data flow in applications [ML00, JKK06b, SAH+10, Sim03]. This section examines three representative examples: *taint analysis*, *symbolic execution* and *security-typed languages*. Taint analysis methods [JKK06b, HYH+04] expand on classic data flow analysis of application source code [ALSU07] while symbolic execution systems [SAH+10, CDE08] emulate program execution under various different inputs. Finally security-typed languages [ML00, Sim03] augment the language type system with data flow information to facilitate static analysis.

The first advantage of static methods for data flow analysis and enforcement is that they can be applied offline. They incur no runtime overhead and, as a result, they can be adopted in environments in which execution speed is a priority. Second, developers are notified during development about potential data flow violations instead of when the problems manifest in production. This helps construct applications free of many types of security vulnerabilities. Third, static methods for data flow analysis, if applied offline on a large corpus of existing applications, can identify undiscovered security vulnerabilities. This enables security studies of large application datasets aiming to pinpoint common instances of security problems.

However, the lack of runtime information limits the applicability of static methods. Many applications have data flow requirements that depend on runtime information. For example in the healthcare scenario (§2.1.2) enforcement that patient data is only released to the correct doctor in the pathology lab is hard to guarantee statically because the potential type of cancer is not known yet. Static analysis is also challenging for languages with dynamic features, which are popular in the web (e.g. PHP). Features such as dynamic code generation or dynamic function calls lead to programs with behaviour that changes at runtime, and thus programs that cannot be fully analysed statically.

Static methods are also more complex than their runtime counterparts. The source of this

complexity is that static methods reason about *any* potential execution of a program. The advantage is that data flow that occurs by *not* executing particular code paths at runtime is also captured (see the discussion of implicit data flow in Section 2.3.2). The disadvantage is that adapting static analysis to enforce custom data flow tailored to individual applications requires significant developer expertise.

Another limitation of static methods is the problem of worst-case assumptions. Static analysers attempt to detect all instances of a security problem. When the tool that performs static analysis is faced with a scenario that it cannot fully assess, it takes the conservative approach and assumes the worst possible outcome [JKK06b]. Worst-case assumptions have a direct impact on the result of the analysis: the analysis will identify safe programs as unsafe introducing *false positives*. Since developers have to assess the result of the analysis manually, increased numbers of false positives reduce the usability of a tool.

2.2.1 Taint analysis

Taint analysis [JKK06b, TPF+09, BKMW10] is a method to identify illicit use of untrusted data in applications. The techniques that it employs are largely borrowed from classical data flow analysis [ALSU07], which are introduced in this section. The main limitation of taint analysis is the lack of runtime information. Thus, after presenting taint analysis, this section continues with a discussion of *string analysis* [CMS03, Min05, HO05]. String analysis is often suggested to improve the results of taint analysis by reasoning statically about the potential content of strings at runtime.

Overview of data flow analysis

Data flow analysis is an influential static analysis method originally developed in the context of code optimising compilers [ALSU07]. It has since been used and extended by the security research community to enforce data flow confidentiality [TPF+09] and integrity [JKK06b, TPF+09] requirements.

In data flow analysis, the analyser associates meta-data with each point in the analysed program. The meta-data represent the program state that may be observed at that point and that is relevant for the particular analysis. The content and the type of the meta-data vary according to the goal of the analysis. For example, if the analysis aims to pinpoint injection vulnerabilities, the meta-data may consist of a set of variables that are potentially set by the user [JKK06b].

Data flow analysis calculates the meta-data for each program point according to some constraints. The constraints stem from the semantics of statements and the control flow of the application. Statement semantics describe how each particular statement affects the meta-data for the program points before and after the execution of that statement. The control flow of the application then introduces constraints on the meta-data for the program points

across two different statements; after the first is executed but before the second. For two consecutive program points their meta-data must be equal. However, the control flow of a program is not linear and forms a graph known as the *Control Flow Graph* (CFG). As a result, the CFG puts additional constraints on the meta-data of the program points for statements at its edges. In code optimisation, the meta-data generated by the data flow analysis shows whether the requirements of a given optimisation are satisfied and thus if the optimisation is applicable.

As an example of data flow analysis, consider the reaching definitions problem [ALSU07]. In the reaching definitions problem the compiler computes which variable definitions are visible when control reaches a given program point. This enables the compiler to generate warnings when an undefined variable is used.

To solve the reaching definitions problem, data flow analysis stores for each program point the set of defined variables along with their definition points. An algorithm starts from the first instruction of the program at which the set of defined variables is known. A potential reaching definitions algorithm traverses iteratively the CFG to calculate the set of defined variables for each program point [ALSU07]. To do so, the algorithm uses the statement semantics; according to each statement, new variables may be defined and the definition of existing variables may change. The algorithm converges to a solution when no modifications are made to the meta-data of any program point in a single iteration.

By carefully selecting appropriate meta-data, data flow analysis can be used to enforce data flow integrity guarantees. Pixy [JKK06b] and similar systems [HYH+04, BKMW10, XA06, LL05], as discussed next, follow this approach.

Taint analysis in detail

Taint analysis is a type of data flow analysis suggested to enforce integrity requirements in web applications [JKK06a]. Examples of its use in the web include PHP applications [JKK06b, HYH+04, XA06], Java applications [LL05, TPF+09] and browser extensions written in Javascript [BKMW10]. On the server-side, taint analysis detects injection vulnerabilities, i.e. data flow that originates from the user and reaches functions that are used in injection attacks (§2.1.1). On the client-side, taint analysis detects data flow from the user to privileged browser APIs, possible due to vulnerable browser extensions [BKMW10]. This is dangerous because untrusted web applications may hijack the browser and execute privileged operations that circumvent the browser's security restrictions. The rest of this section discusses taint analysis using Pixy as a representative example.

Pixy [JKK06b] is a taint analysis tool that automatically detects XSS vulnerabilities in PHP applications. It locates statements in PHP applications that return data to the client (e.g. `print` or `echo`) and can be used in XSS attacks. It achieves this by applying data flow analysis to identify statements that use input parameters without sanitisation.

To perform taint analysis, Pixy stores and updates a single bit of meta-data that marks

each variable as potentially tainted if the variable may contain data that originated from the user. Initially, only the variables that contain HTTP request data are tainted. Pixy then applies a standard iterative algorithm to the application's CFG and calculates the meta-data for each program point. Pixy models PHP library functions and operator semantics so that operations on tainted data result in tainted values and calls to sanitisation functions result in untainted return values. When a potentially tainted variable is used in an operation that requires sanitisation, taint analysis reports an injection vulnerability.

Pixy combines taint analysis with two additional types of data flow analysis that improve its effectiveness: *alias analysis* and *literal analysis* [JKK06a].

PHP supports variable references, i.e. two variables that reference the same physical memory location. These variables must be considered equivalent in taint analysis. For this, alias analysis maintains meta-data about groups of variables that reference the same memory location. This information is used in variable assignment operations because it allows Pixy's taint analysis to update the meta-data for every variable affected by an assignment.

A PHP application may contain code that is not executed, e.g. an `if` branch only activated under some configurations. To allow taint analysis to avoid such code paths, which should not affect taint analysis, literal analysis maintains meta-data about the potential values that each variable may hold at runtime. This allows taint analysis to rule out some of the branches followed by the application and, as a result, achieve more precise results.

The three static data flow analyses suggested by Pixy have a number of limitations. First, literal analysis is only effective for constant values and cannot precisely specify variable values when these depend on user input. In fact, Pixy only uses literal analysis results to prune branches in conditional statements and not for other dynamic language features that are typically combined with runtime data, e.g. the target function in variable function calls. Second, the lack of knowledge about the actual variable values limits Pixy's taint analysis when arrays are involved. For example, when an application references the `i`th element of an array that contains some tainted elements using the variable `$i`, Pixy cannot statically compute which element is referenced. Thus, it cannot fetch the element's meta-data and has to mark the element as tainted to avoid false negatives, i.e. considering a data flow to be safe when it is not. Finally, Pixy does not automatically analyse files imported with the `import` keyword, and it requires developer input to specify the actual files to analyse. Again, literal analysis cannot capture the exact target file of `import` statements since the actual file name often depends on the runtime configuration of the host system.

Overall, Pixy is an important system for static taint analysis in PHP that identifies XSS vulnerabilities in existing code. Compared to other static analysis systems for PHP [HYH+04, XA06], Pixy better supports common language features such as references and multi-level arrays and its taint analysis is more accurate [JKK06a]. When compared to similar taint analysis systems for Java [LL05], Pixy offers comparable false positive rates while supporting a significantly more dynamic language than Java.

Without runtime information, however, Pixy's support for most dynamic aspects of PHP is

```
1 <?php
2   $s="b";
3   for ($i=0 ; $i<$n ; $i++)
4     $s = "a".$s;
5   echo $str;
6 ?>
```

```
1 S → b
2 S → aS
```

Listing 2.1: An example of string analysis in a PHP program. The resulting grammar that describes the contents of the generated string is shown on the right hand side.

limited. In PHP, the developer can use a dynamically constructed string as (1) the script filename to include (*dynamic* `import`), (2) the name of a variable to read or write (*variable variables*), (3) the name of a function to invoke (*variable function calls*), (4) the name of a class to instantiate (*variable* `new`) and (5) the code to execute (`eval` *support*). These language features limit Pixy's practicality. For example, Wordpress, the popular weblog platform that is used in Section 5, makes extensive use of these language features.

Dynamic language features are a common problem for static taint analysis systems. VEX, the system for taint analysis in browser extensions, does not support Javascript's `eval` with dynamically generated strings [BKMW10]. For Java, Livshits and Lam's taint analysis does not handle dynamically loaded classes [LL05] while TAJ explicitly models common J2EE operations, such as calls to Enterprise JavaBeans, because it does not support Java's dynamic features used in their implementation [TPF+09]. As presented next, *string analysis* collects information about how strings are constructed, and this information can be used to improve support for dynamic language features.

Improving taint analysis with string analysis

String analysis [CMS03, Min05, HO05, WS07] is a static analysis technique that approximates the structure of strings generated by an application at runtime. String approximation is important for taint analysis because it enables precise taint analyses with minimal user input and, for dynamic languages, it provides better support for their dynamic features.

The core idea of string analysis is to consider string operations in the original program as production rules for a context-free grammar [Min05]. The output of string analysis is a context-free grammar, which approximates the contents of each string generated by the program at runtime. Listing 2.1 shows an excerpt of a PHP program and the resulting grammar, after string analysis [Min05]. In this example, static analysis may be unable to identify the exact final value of `$str`—it depends on the runtime value of `$n`—but it can reason about `$str`'s structure. In particular, the assignment in the fourth line of the PHP program in Listing 2.1 corresponds to the second production rule in the right-hand side grammar. Such a grammar enables queries using regular expressions on the potential content of each string [Min05], e.g. "can `$str` contain the value `[/a.*/]`?".

String analysis operates on a simplified version of the CFG, the *flow graph*, that is constructed

based only on string operations [CMS03]. The flow graph uses nodes to represent string creation and manipulation statements (such as calls to the `subString` method in Java) and edges to link strings with the statements that they are involved in. The static analyser walks the flow graph and creates a context-free grammar with explicit string operations, i.e. grammars similar to Listing 2.1, in which terminal symbols (or else terminals) represent string-processing library functions. In the final step, the functions are removed from the grammar using *library function approximations*, and the context-free grammar is generated.

Library function approximations are required because some of the operations performed on strings by the corresponding library functions cannot be described by context-free grammars. Such examples are character replacement or regular expression functions [Min05]. When the analyser uses an approximation, the accuracy of the result suffers because false positives and false negatives are possible.

String analysis eliminates a need for endorsement annotations in taint analysis. In a system such as Pixy, developers have to annotate sanitisation functions manually, effectively trusting them as integrity endorsement functions (§2.1.1). Instead, string analysis systems have information about the structure of each string. Thus, with string and taint analysis combined, when a variable is used in a function that can be exploited in an injection attack, static analyses devise a context-free grammar to capture the variable's potential content. Individual terminals in this grammar are annotated with taint meta-data. Using this information, injection vulnerabilities are only reported when the grammar permits tainted terminals to change the semantics of the function [WS07]. The advantage of string analysis is that it can pinpoint errors in the sanitisation functions themselves. There are cases, however, in which the application uses different sanitisation functions according to the context: an application may allow its administrators to sidestep the normal sanitisation operations regarding HTML content. In such scenarios, annotations are still required [WSA+11].

String analysis is also used to support the use of dynamic language features in the analysed program. Wassermann and Su [WS07] rely on string approximations to resolve statically the target of dynamic `import` statements in PHP. Without information about the data that the program receives at runtime though, string analysis has only limited potential to improve support for other dynamic language features such dynamic code generation. String analysis approximates only the structure of strings, and for taint analysis to analyse code, the precise content of the generated strings is required.

Overall, taint analysis is an important technique for static data flow analysis that is limited by the use of dynamic language features. The next section introduces symbolic execution, which employs traces from concrete program executions to guide the analysis when dynamic language features are used.

```
1 <?php
2   $uid = isset ($_GET['uid']) ? $_GET['uid'] : ''
3   if ($uid=='' || $uid > 1000) {
4       exit $uid->error;
5   }
6   {...}
7 ?>
```

Listing 2.2: An example PHP program to demonstrate symbolic execution.

2.2.2 Symbolic execution

Symbolic execution is primarily a testing technique, designed to automatically generate test suites with high code coverage [CDE08, CGP+08]. Recently, it has been suggested as a method for data flow analysis in web applications [CF10, SAH+10, WYC+08]. This section outlines the basic principles of symbolic execution and discusses its applicability as a data flow analysis technique.

Overview of symbolic execution

Symbolic execution emulates the execution of a program under every potential input. In contrast to taint analysis, symbolic execution follows every program path separately. On each path, it inspects the values that the program uses and identifies cases, in which a value is being used in an unsafe way, e.g. a pointer that is dereferenced without a check for a `null` value. Symbolic execution is applied offline and incurs no overhead at runtime. Therefore, it is classified here a static method for data flow analysis.

Symbolic execution can identify violations of arbitrary pre-conditions for specific program operations. In general, pre-conditions are necessary checks that the application must invoke before performing an operation. Pre-conditions are hard-coded in the symbolic execution system [SAH+10, WYC+08] or added to the analysed program as annotations [CF10].

Symbolic execution assumes program variables contain *symbolic values*. A symbolic value does not represent concrete data such as `$v=5`. Instead, it captures the application's operations as a set of constraints. For example, "`$v` is greater than 0" can be the symbolic value of variable `$v` in a code path that is only executed if `$v>0`. Application operations are replaced with operations on symbolic values that produce symbolic results with additional constraints. When the code branches after inspecting a symbolic value, symbolic execution follows both branches each with a complementary new condition associated with the symbolic value. When the exploration of all program paths finishes, symbolic execution can use the constraints on symbolic values to generate concrete input.

As an example, assume that a symbolic execution system analyses the PHP program in Listing 2.2 for dereferencing errors. Initially, the variable `$_GET`, which represents the HTTP request, is marked as symbolic. The check in line 2 has two potential results: `$uid` is assigned

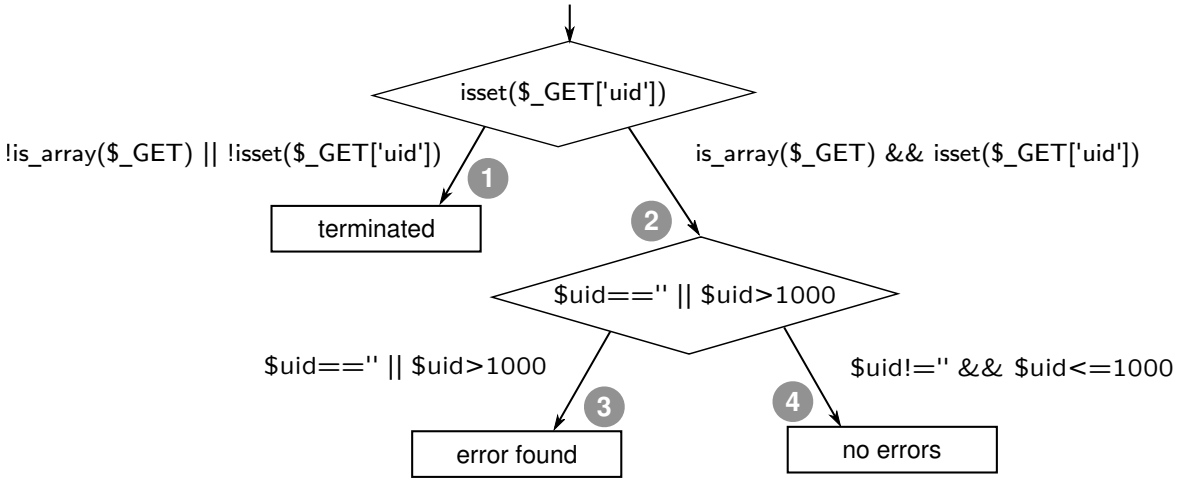

Figure 2.3: Exploration of the program from Listing 2.2 using symbolic execution.

either to the concrete empty string value or to the symbolic value in `$_GET['uid']`, with the restriction that `$_GET` is an array and `$_GET['uid']` exists. Both cases must be checked so symbolic execution follows both paths, each with different constraints on the value of `$_GET`. These two cases appear in Figure 2.3 labeled (1) and (2), respectively, right after the rhombus, which captures the check. The former case only involves concrete values and executes until termination in line 6 (Listing 2.2). The latter case, however, involves the symbolic value `$uid` being compared to concrete values. The checks in line 3 thus generate two complementary constraints on `$uid`, one when the condition is true (labeled (3) in Figure 2.3) and one when the condition is false (labeled (4) in Figure 2.3). Both constraints can be satisfied for some values of `$uid`. Therefore symbolic execution continues with both paths until it has explored the whole program.

The detailed exploration of the program using symbolic execution can generate concrete test cases. Given the previous analysis, for example, three test cases that derive from the constraints on `$_GET['uid']` are sufficient to explore the excerpt in Listing 2.2: `$_GET['uid']` not set, `$_GET['uid']=0` and `$_GET['uid']=1001`. A symbolic execution system may pinpoint programming errors by only generating test cases that lead execution to particular code paths. In the previous example, there is no constraint that forces `$uid` to be an object in the path labeled (3) in Figure 2.3. As a result, the corresponding test case, in which `$_GET['uid']` is not set, reaches line 4 in Listing 2.2, crashes the application and exposes a dereferencing error.

Data flow analysis with symbolic execution

Symbolic execution identifies various different errors in applications. Since errors are nothing more than violations of pre-conditions for specific program operations, symbolic execution systems [SAH+10, WYC+08, CF10] have devised pre-conditions that, if satisfied, enforce data flow confidentiality and integrity requirements.

Rubyx [CF10] is a symbolic execution system for Ruby-On-Rails (RoR)[8] web applications that identifies XSS vulnerabilities. Rubyx introduces the boolean property `trusted` in the Ruby `String` class as part of its runtime libraries. During symbolic execution, symbolic values that represent user data have initially `trusted` set to `false` and RoR' sanitisation functions are modified to change `trusted` to `true`. The pre-condition required to avoid XSS vulnerabilities is: "for all statements that return data to the user, the data's `trusted` property must be `true`". This effectively enforces a policy for data flow integrity, i.e. untrusted user data should flow though sanitisation functions before they are sent to the client.

A similar system for client-side XSS mitigation is Kudzu [SAH+10]. Kudzu performs symbolic execution for Javascript applications. A client-side Javascript application is vulnerable to XSS when it passes non-sanitised user input to a Javascript evaluation construct such as `eval`. In contrast to Rubyx, Kadzu does not trust the sanitisation operations of the application and verifies them manually. To achieve this, it models the Javascript string functions as constraints on symbolic values and uses the result constraints to check whether they preclude known XSS attack vectors.

Symbolic execution as a static data flow analysis technique offers two advantages over taint analysis: reduced numbers of false positives and improved support for dynamic languages. First, symbolic execution systems incur fewer false positives because they verify with concrete input values the data flow violations that they discover [SAH+10, CF10, WYC+08]. Second, symbolic execution systems better support dynamic languages by leveraging concrete execution traces [SAH+10, WYC+08]. When symbolic execution reaches a path, in which the use of a dynamic language feature requires precise knowledge of a value (e.g. to calculate the name of a variable), the symbolic execution system generates such concrete input that makes concrete execution (i.e. execution without symbolic input parameters) reach the same program point. The symbolic execution system instruments that particular concrete execution of the program, discovers the specific value that the program generates and proceeds with symbolic execution based upon that value. This reduces generality but allows symbolic execution to achieve better code coverage.

Symbolic execution, however, has a number of limitations, with the most important being limited scalability with the size of the analysed applications. During the analysis of larger applications, the number of different paths that symbolic execution needs to explore increases significantly. Symbolic execution systems optimise the order, in which they explore paths but they employ timeouts as the potential number of paths may be impractical for inspection. In practice, symbolic execution systems rarely achieve full code coverage. This means that their results suffer from false positives because they do not account for every potential data flow in the analysed application.

Another challenge for symbolic execution systems is modeling the environment. Accesses to files, databases or interactions with the operating system must be modeled to increase code coverage and to avoid false positives. Kudzu, for example, introduces a browser event

[8] Ruby on Rails, `http://rubyonrails.org/`, last accessed: 12/9/2012

generator so that all application event handlers are triggered. KLEE [CDE08], a symbolic execution system for C applications, supports a simple model of the file system. It allows the analysed applications to store and retrieve symbolic values. Similar models, however, do not exist for other components such as databases. When the analysed application reads data from a database, existing systems treat the returned value as symbolic without any constraints [CF10]. This increases the number of paths that need to be checked needlessly.

A third limitation of symbolic execution is constraint solving. Each time a new code path is introduced after a conditional statement, a constraint solver [GD07] inspects the constraints and decides whether they can be satisfied or not. Constraint solvers have limited capabilities in practice, e.g. for cases in which the program calculates a cryptographic hash, solvers fail to generate a solution for the constraints that arise from the calculation of the hash in a reasonable amount of time [AKD+08]. This effectively prevents symbolic execution from automatically analysing code past a certain point (e.g. an hash-based access control check) and requires domain- and application-specific annotations for symbolic execution to proceed.

To summarise, symbolic execution is a promising technique to analyse applications for violations of data flow requirements. However, for large applications, for applications that employ dynamic language features or for applications that use external services such as data stores, symbolic execution is currently inadequate to achieve effective data flow analysis.

2.2.3 Security-typed languages

Volpano and Smith [VS97] suggested in their seminal work to augment a traditional language type system with data flow annotations. This would enable developers to express arbitrary confidentiality and integrity data flow policies, which the compiler can enforce. Jif [ML00], the programming language that Myers and Liskov subsequently introduced, is known as a *security-typed language* because data flow requirements are explicitly declared as part of the type of each variable. Jif's design has influenced a number of languages [SM03, Sim03] and successor systems [VEK+07, ZBWKM06, KYB+07], which are presented below.

Programs written in a security-typed language such as Jif [ML00] or FlowCaml [Sim03] enforce *non-interference* and are *composable* [VS97, SM03]. Non-interference [GM82, KWH11] dictates that data belonging to a security category cannot interfere with data belonging to another category. For confidentiality, non-interference requires that variations of secure input data do not result in variations of public output data while for integrity, variations of untrusted input data must not affect trusted output data. Non-interference forces developers to declare correctly the security types of generated data for their application to compile successfully. Due to composability, when two programs that enforce non-interference are combined in a larger program, the resulting program also enforces non-interference [SM03]. Both these properties make security-typed languages an important tool for creating applications that enforce data flow by design.

The rest of this section presents security-typed languages using Jif as a representative ex-

ample. Jif introduces the *Decentralised Information Flow Control* (*DIFC*) label model to express data flow policies in its security types. DIFC is the basis of DEFCon and of other dynamic systems, both presented later in the thesis. This section also discusses the limitations of static data flow analysis in security-typed languages and introduces the need for runtime extensions.

Decentralised Information Flow Control and Jif

DIFC, the label model used by Jif's security types, is an extension of the label models used in traditional Information Flow Control (IFC) systems [BL73, Bib77]. Such Mandatory Access Control systems are used in military multi-level security [US 83]. Principals and objects are typically assigned security labels denoting levels. Principals represent the users of the application. Access decisions are governed by a "can-flow-to" partial order relation. As an example, a principal operating at level "secret" can read a "confidential" object but cannot read a "top-secret" or write to a "confidential" object. Through this model, a system can enforce confidentiality of data flows: information marked as "secret" only flows to principals with "secret" (or higher) clearance. The privilege to override confidentiality IFC restrictions, e.g. write "secret" data to a "confidential" object, is known as the *declassification* privilege. In an IFC system, only the few principals that are trusted to respect an organisation's data disclosure policy should possess the declassification privilege.

The main idea in *Decentralised* IFC (DIFC) is that the responsibility to create new labels and to distribute the privileges that allow principals to override the corresponding data flow policies is *decentralised across applications*. In contrast to traditional IFC systems, which use a centralised trusted entity to create labels and distribute privileges, in DIFC each application devises its own labels and uses them to protect the data that it processes. This allows each application to express its own data flow policies that the DIFC system should enforce and not be limited by the select policies that an administrator has preconfigured.

In the context of Jif, consider $L = \{o_1 : r_1, r_2\}$ as an example of a DIFC label. L consists of the *principals* o_1, r_1 and r_2. Communication channels, such as network sockets, are associated with the principals that they communicate with. Processes execute on behalf of principals. The label L specifies that any data annotated with it are *owned* by o_1 (i.e. o_1 has the declassification privilege) and that, apart form o_1, only principals r_1 and r_2 are allowed to read the data (i.e. they can act as *readers*).

Jif extends Java by adding DIFC labels to its type system [ML00, CVM07]. The compiler examines all program statements based on the DIFC labels of the variables involved and the semantics of each Java operation. The Jif compiler enforces that data stored in a variable protected by a label, such as L, do not reach a variable or a communication channel with a more permissive label, e.g. a label that contains additional readers than r_1 and r_2. If this happens, the compiler rejects the program, and this prevents a potential data flow violation at runtime. The compiler then generates executable applications that propagate data according

to the data flow requirements stated by the DIFC labels. The compiled programs run on the unmodified Java Virtual Machine after the security annotations have been removed.

Jif enables developers to protect data by devising arbitrary labels and use them to restrict data flow for each variable. DIFC labels are not restricted to confidentiality; they can be used to express policies about the integrity of data flows as well [ML00]. Jif also assumes that different principals have different privileges in regard to labels. The declassification privilege, for example, i.e. the ability to read a variable protected by L and store data to a variable labelled with an additional reader, is reserved only for processes that run on behalf of the owner, i.e. o_1. Since the user that executes a program is only known at runtime, Jif programs need runtime support to ensure that static analysis' assumptions hold at runtime.

Static analysis and runtime extensions

Jif supports most features of Java but highlights the limitations of static analysis. The DIFC label model, as used by Jif, is unable to specify data flow requirements when the application principals are only known at runtime. The healthcare scenario from Section 2.1.2 is such an example because new patients arrive while the system operates. To support this, a Jif application would have to be modified and recompiled with additional principals in its labels. Newer versions of Jif support *dynamic labels and principals* to avoid recompilation [And99, CVM07]. In such cases, Jif reverts to dynamic data flow checks because static analysis alone is not enough. Dynamic features of Jif have been shown to be essential for building non-trivial applications [HAM06, CVM07].

Access to a database or a file system is another challenge for Jif's static checks. SIF is a servlet container for Jif web applications [CVM07]. In SIF, database access is performed through a set of methods with security-typed input parameters. This means that, statically, the Jif compiler only enforces that all data sent to such methods are labelled with a common DIFC label, i.e. the label of the input parameter of the method. This may be acceptable for data flow policies, such as the policy to prevent injection attacks from Section 2.1.1. In practice, however, many applications have different data flow requirements for data stored in the same database. For example, cancer registry reports from Section 2.1.2 cannot be disclosed to a GP while pathology reports, potentially stored in the same database, can. In such cases, Jif requires privileged application code that dynamically serialises labels and stores them in the database at runtime.

Finally, another limitation of security-typed languages is that they require applications to be rewritten in a novel programming language with static typing and low-level, label-based data flow policies. While this may be a reasonable requirement for domains in which enforcement of data flow is a priority, the additional complexity is not typically justified. In fact, subsequent research indicates that building realistic applications using Jif requires additional runtime tools, such as policy management infrastructure, that are currently not available [HAM06]. The prevalence of dynamically typed languages in the web render the adoption of a static- and security-typed language such as Jif particularly challenging.

2.3 Runtime taint tracking

Runtime taint tracking is a technique for analysing and enforcing data flow in applications [SAB10]. In contrast to static analysis, the program statements that an application executes as well as their execution order are known at runtime. These include statements generated dynamically, e.g. when fetching code at runtime from remote locations. At runtime, the result of program statements is also known to the taint tracking system. The analysis therefore can focus on the current execution and not on alternative paths, which may never affect data flow. These properties render runtime taint tracking suitable for data flow analysis in large systems written in dynamic languages, which are hard to analyse statically.

At the same time, runtime taint tracking is a simple technique for developers to understand and use. Conceptual simplicity has motivated research on systems that expose taint tracking to developers as a stand-alone security mechanism [VEK+07, ZBWKM06, YWZK09, SBL09]. Such systems use developer input to devise a data flow policy and then restrict processing as a result of that policy. Assuming that the part of the system that enforces data flow policy has been implemented without errors, correct data flow follows in any application built on top. In contrast to security-typed languages, developers continue to program in familiar languages and only have to learn how to interact with the taint tracking system.

Multiple runtime taint tracking systems have been suggested in the past but the lack of common terminology hides the shared design principles across them. With this in mind, Section 2.3.1 establishes a model of runtime taint tracking systems and introduces the main design decisions that differentiate them. Section 2.3.3 uses this model to discuss the design choices in a range of recent runtime taint tracking systems.

2.3.1 A model for runtime taint tracking

Definition 2.1 (Runtime Taint Tracking)**.** Runtime taint tracking is a security technique to track data flow inside a running application. It operates by associating *taint meta-data* with the data that the application processes. The application is instrumented to create, modify, and propagate the taint meta-data transparently at runtime according to the statements that it executes. The taint meta-data are used to analyse the data flow inside the application or to enforce a specific *data flow policy*.

How a runtime taint tracking system achieves its intended purpose, i.e. data flow analysis or enforcement, depends on a set of design decisions. In general, there are three important decisions: (1) *how the system isolates data*, (2) *how the system tracks data flow across isolated data* and (3) *how the system uses the output of data flow tracking to enforce data flow*. The choice of application programming language often restricts how the designers of a runtime taint tracking system may answer these questions. As explained next, the reason is that different languages offer varying support for isolating application data and varying

inspection points to which checks for data flow tracking and enforcement can be attached.

To gain a better understanding of the design choices above, Figure 2.4 illustrates a generic runtime taint tracking system. The figure shows in an abstract form an application that is compartmentalised to facilitate data flow analysis along with the structures needed to perform taint tracking. The figure, along with the different design choices for isolation, data flow tracking and enforcement, are explained in the following paragraphs.

Data isolation. Data isolation is a prerequisite for effective data flow tracking. It prevents the application from exchanging data using mechanisms that are not explicitly controlled or monitored by the runtime taint tracking system. When data exchange is monitored by the taint tracking system, the results of data flow analysis do not contain false negatives (i.e. data flow that occurs in the application but is not detected by taint tracking).

To achieve data isolation, the system first separates the analysed application from its environment. All outside communication is inspected by the runtime taint tracking system, effectively preventing any unmonitored access to external data. In Figure 2.4, this is depicted as a grey border around the application. The external isolation is important because most data flow policies and analyses specify concerns about how the application receives data from or communicates data to the outside world. For example, an application that performs stock trading (§2.1.2) should only receive stock tick data from the stock exchange and never disclose confidential data about the trading that it performs.

Data, however, must be isolated so that data flow is effectively tracked inside the application. Thus, the application is internally compartmentalised into multiple *isolated components*. Compartmentalisation keeps the data of each isolated component separate similar to the external isolation of the entire application. In Figure 2.4, this internal compartmentalisation is depicted using colourless borders. The runtime taint tracking system explicitly monitors all data exchange across isolated components. An isolated component can be as small as a single program variable or as large as a set of executable scripts.

Data flow tracking. Assuming effective data isolation, the propagation of data in the application can be tracked. To achieve this, the taint tracking system maintains *taint meta-data* for each isolated component. The taint meta-data may range from simple bits that mark the data as "tainted" or "confidential" to more complex constructs. Taint meta-data store information about the security properties of the actual data in the same component.

Maintaining accurate taint meta-data at runtime is the sole purpose of a taint tracking system when it is used to analyse data flow, e.g. to pinpoint malicious application behaviour. In the case that the system also aims to enforce a data flow policy, accurate taint meta-data enable the taint tracking system to stop any policy-violating data flow before it occurs.

The granularity at which data is internally isolated and tracked, thereafter referred to as the *tracking granularity*, is an important property of most runtime taint tracking systems. It

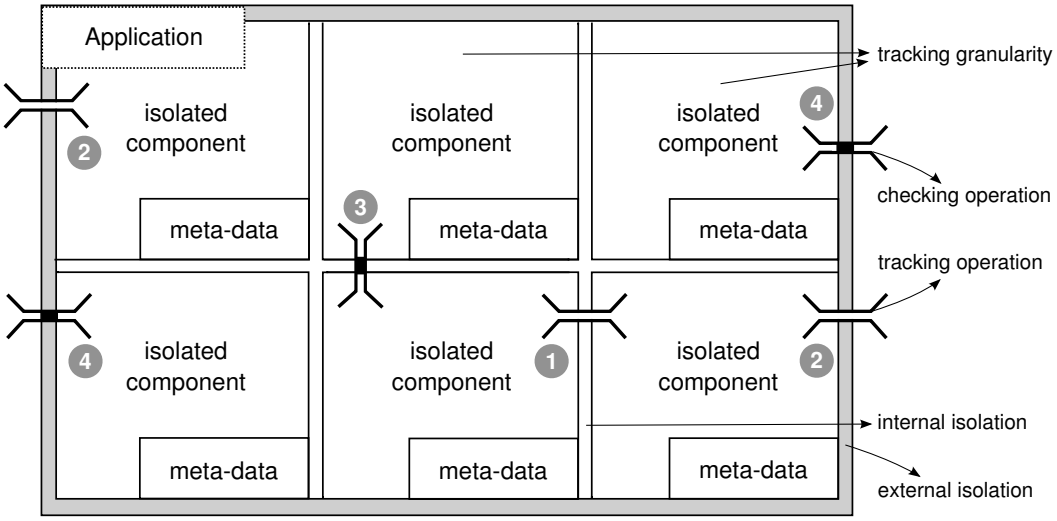

Figure 2.4: Overview of a generic runtime taint tracking system. The main design parameters are the tracking granularity, the isolation of components, and the tracking and checking operations imposed.

is rarely configurable by the system's users and affects the ability of the system to analyse and enforce data flow. Finer tracking granularity results in more precise data flow tracking. Increasing the granularity also increases the runtime overhead for storing and updating meta-data, and this can slow down the application significantly. Two typical examples of tracking granularities that many of the systems presented in Section 2.3.3 use are process-level granularity and variable-level granularity.

While the application exchanges data across isolated components, the taint tracking system (1) intercepts the data exchange, (2) inspects the data and (3) updates the taint meta-data of each component. These three actions consist a *tracking* operation, which is automatically invoked by the taint tracking system to maintain accurate taint meta-data. Data exchange occurs frequently therefore, unless tracking operations are efficient, performance suffers.

To track data flow effectively, tracking operations must be invoked at every data exchange point across isolated components and when an isolated component communicates with the outside world. Such points appear in Figure 2.4 marked (1) and (2), respectively. In practice, the internal isolation provided by many runtime taint tracking systems does not always guarantee that tracking operations are invoked. For example, taint tracking systems that use a process-level tracking granularity are typically unable to intercept data exchange as a result of processes manipulating the scheduling of the CPU. Two collaborating processes may use this mechanism to exchange arbitrary data, circumventing the taint tracking system.

Data flow enforcement. When a runtime taint tracking system supports data flow enforcement, it provides tracking operations that also perform policy checks. These *checking* operations inspect the taint meta-data of the isolated components when data exchange occurs, reason about the policy compliance of the data exchange and act to prevent data flow that violates policy. As presented in Figure 2.4, checking operations may be invoked on data

Design Parameters	Design Decisions in the Literature
Tracking granularity	Process, memory segment, memory byte, object, variable, string character, scope of code, group of scripts
Taint meta-data	DIFC labels, 1 ("taint") bit, 2 bits, 32 bits, arbitrary bitmap, transformer object, policy object
Tracking operations	Taint bit propagation, union, overridable method
Checking operations	"Can-flow-to", taint bit check, context sensitive string evaluation, filter object

Table 2.2: Overview of design choices from taint tracking systems presented in Section 2.

exchange across isolated components (3) or on data exchange with the outside world (4).

Checking operations are similar to tracking operations but differ in purpose and in how frequently they are invoked. While tracking operations ensure that the system has an accurate view of the data stored in each isolated component in the form of taint meta-data, checking operations leverage this information to enforce data flow policy. As a result, the frequency of checking operations depends on the number of points during program execution in which the system must decide whether a particular action is permitted or not. The number of such points depends on the data flow policy being enforced. For many runtime taint tracking systems and data flow policies, only few inspection points are needed compared to the number of data exchanges across isolated components. For example, to enforce an integrity policy that prevents injection vulnerabilities in web applications (§2.1.1), checking operations should only ensure that a small set of functions are never called with user-supplied data. Instead, tracking operations must be called every time the application combines the data stored in two variables. Thus, checking operations are typically less frequent than tracking operations.

At a high level, a taint tracking system translates a data flow policy to a specific tracking granularity, taint meta-data and tracking/checking operations. This thesis refers collectively to the choices in each of these design decisions as the *taint tracking policy* of the system. Systems have different flexibility in devising taint tracking policies. At one end of the spectrum, runtime taint tracking systems enforce domain-specific data flow policies with limited or no input from the application administrator or user [NTGG+05, DKZ09, VNJ+07]. In such systems, the taint tracking policy is largely fixed and the user has few or no configuration options. At the other end, systems that expose taint tracking to the developers offer the ability to set the values of taint meta-data [VEK+07, ZBWKM06, KYB+07], control the actions of checking operations [BMW+11] or even alter the type of taint meta-data that the system maintains [YWZK09].

2.3.2 Covert channels

A common assumption in runtime taint tracking systems is that the developer does not actively try to evade tracking. This is known as the *benevolent developer* assumption [BMW+11, DC10]. The benevolent developer assumption reflects the inability of runtime taint tracking

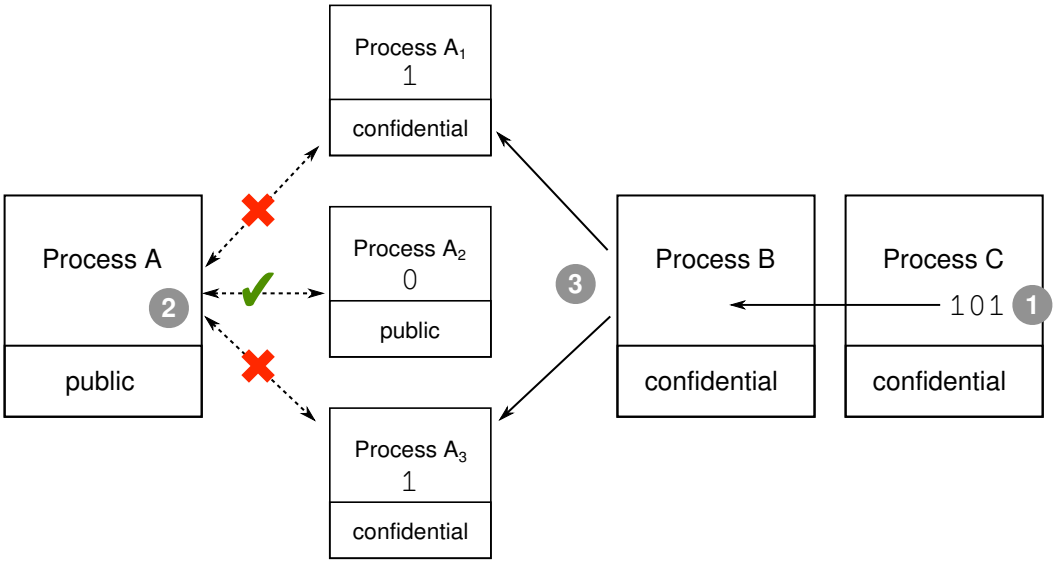

Figure 2.5: Exploiting implicit taint meta-data changes to leak confidential information [KT09].

systems to guarantee that *all* data flows are monitored effectively. An unmonitored data flow is know as a *covert channel* [KWH11].

Covert channels are divided into *storage* and *timing* channels. Both types of channels may be used by malicious developers to evade tracking. Covert storage channels, such as a shared memory location that is not under the control of the taint tracking system, typically allow high-bandwidth, reliable communication. Access to covert storage channels must be prevented or else the taint tracking system cannot reliably track data flow. In contrast, taking advantage of timing channels requires monopolising or exhausting system resources, and such behaviour is likely to be noticed. As an example of a timing channel, consider a process that, if a confidential event occurs, increases its CPU utilisation. Collaborating processes may observe a reduction in their performance and deduce that the confidential event occurred.

The following paragraphs introduce two important types of covert storage channels that significantly limit the design and applicability of runtime taint tracking systems. *Implicit taint meta-data changes* are a covert channel that impacts the design of taint tracking systems with coarse tracking granularities. *Control* and *implicit* flows are covert channels that limit the applicability of variable-level taint tracking systems when operating on malicious code.

Implicit taint meta-data changes. Implicit meta-data changes are an important design flaw for taint tracking systems with coarse tracking granularities because they can be exploited as a covert channel. Asbestos [VEK+07] (later presented in Section 2.3.3) suffers from this problem and subsequent systems took explicit measures to avoid it.

Figure 2.5 presents two processes, A and B, which collaborate to learn and disclose a confidential value while evading taint tracking. Assume that the taint tracking system monitors these processes and updates the taint meta-data of each process according to the confiden-

```
1 int<confidential> i=get_boolean_result();
2 int<public> j=false;
3 if (i) j=true;
4 print(j);
```

Listing 2.3: Control and implicit flow from variable i to variable j.

tiality of the data that it receives. A third process C stores the confidential binary value 101.

To mount an attack, process B requests from process C the confidential value. In step 1, process C sends the data. Tracking operations in the taint tracking system update the taint meta-data of process B implicitly so that the process can receive the data. Process B stores the confidential value 101, however, it is unable to communicate the value to A without it being reflected on A's taint meta-data.

The problem with this design is that updating taint meta-data implicitly is *externally visible*, i.e. other processes that were previously able to communicate with B are then unable to do so. This effectively "leaks" one bit of information to any other process. When implicit taint meta-data changes are combined with the ability to spawn a large number of processes, there exists a high-bandwidth covert channel that is not monitored by the taint tracking system.

In the example of Figure 2.5, processes A and B exploit this channel. In step 2, process A spawns three child processes, A_1–A_3, that communicate back to A in regular intervals. Process B can now follow a simple strategy to disclose the confidential value 101 to A: it identifies which bits in the confidential value are set and, in step 3, sends to the corresponding processes A_1–A_3 an empty message (here A_1 and A_3). The taint tracking system detects this, updates the taint meta-data of A_1 and A_3, and therefore prevents their subsequent communication with process A. This effectively discloses to process A the confidential value 101 without it being reflected in A's taint meta-data.

Successor systems to Asbestos avoid this covert channel by requiring explicit changes to taint meta-data [ZBWKM06, KYB+07]. For a process to receive an incoming confidential message, it has to change its taint meta-data explicitly so that they reflect the confidentiality of the message. This eliminates the covert channel in the scenario from Figure 2.5 because all three processes A_1–A_3 have to change their meta-data explicitly to receive confidential messages from process B. Process A only learns that processes A_1–A_3 are *interested to learn* confidential information—not that they they have received such. Krohn and Tromer [KT09] formally proved non-interference in the context of the Flume operating system, which uses a taint tracking policy that requires processes to modify taint meta-data explicitly.

Control and implicit flow. Runtime taint tracking systems that use a variable tracking granularity may need to track *control* and *implicit data flow* between variables. While control flow can be tracked at runtime with the cost of increased false positives, implicit flow is an important limitation for all taint tracking systems that operate only at runtime.

Listing 2.3 shows an example of control and implicit flows in pseudocode. In lines 1 and 2,

variables i and j are declared and marked with different taint meta-data. The conditional assignment in line 3, however, effectively stores the confidential value of i to the public variable j. This happens both when i is true and the assignment is executed, *or when i is* false *and the assignment is skipped.*

The first type of data flow is known as a control flow [GMPS11]. The value of the confidential variable i is implicitly known in every statement that executes as a result of the control statement in line 3. This means that every statement that updates a variable in the if-branch that is taken at runtime may disclose the value of i. Some of the systems that will be presented in Section 2.3.3, track control flow by considering the taint meta-data of the tested expression (i.e. i's meta-data in this example) in every assignment that is performed as a result of the control statement. In the above example, the runtime taint tracking system can detect that the assignment to the public variable j can leak confidential data because the control statement tests a confidential variable. The downside is that the number of false positives increases because most of the assignments that benevolent developers perform as a result of control statements are not used to store the value of the tested expression.

The latter data flow, however, i.e. when i is false, is an example of an implicit flow because the value of i flows to j without an explicit assignment. Implicit data flow occurs when the branches of a control statement alter different sets of variables. It is an important problem, because at runtime, the taint tracking system can only detect which variables are updated in the branches that the application actually follows. This means that any data flow that a malicious application may try to generate by omitting to update specific variables would remain undetected [AF09, AF10]. Recent research suggested to prevent implicit flow by halting program execution early [SR09, Rus10, AF09] but this may prevent legitimate applications from executing [AF10]. For this reason, variable-level runtime taint tracking is combined with static analysis when the system must handle potentially malicious code [VNJ+07].

Cavallaro et al. [CSS08] demonstrate the above types of covert channels and also suggest additional types of strategies that malicious code can use to evade a runtime taint tracking system that tracks data flow at the variable granularity. In particular, malicious code can overwrite pointers and trick benign applications into disclosing sensitive data as part of their normal operation. Such attacks can disclose confidential data without malicious code accessing them therefore they remain undetected. Subsequent research [SB09] showed that pointer tainting which was typically suggested to protect against such exploitation strategies is inadequate in practice due to increased numbers of false positives.

2.3.3 Runtime taint tracking systems

Runtime taint tracking systems are popular in the security research community. Table 2.2 shows an overview of the various design choices made by taint tracking systems described next. The tracking granularity varies significantly across systems and is never configurable by end users. For taint meta-data, Jif's DIFC labels are a popular choice because they can

express both integrity and confidentiality policies. Other systems use generic policy objects, i.e. data flow policies are specified using code written in the same language as the application. The advantage of specifying data flow policy with code is that no translation of data flow policy to specific tags or labels is required. When the taint tracking system only tracks data flow for a particular type of analysis, a single bit of meta-data is often enough.

The choice of taint meta-data limits the design choices for checking and tracking operations. DIFC labels are ordered by a "can-flow-to" relation (§2.2.3). Thus, all systems with DIFC labels use "can-flow-to" in checking operations. Systems with bit-based taint meta-data propagate the value of the taint bit when one isolated component receives data from another isolated component. According to the bit's value, checking operations prevent specific application actions. Systems with policy objects often allow the user to control how policy objects propagate in tracking operations and how policy objects are checked in checking operations.

This section presents influential runtime taint tracking systems. It begins with implementations in the operating system, continues with the domain of web application security and privacy, and closes with taint tracking systems for desktop and mobile applications.

Operating Systems

Runtime taint tracking has inspired the development of a new breed of operating systems, which enforce data flow and limit potential data leaks in the presence of application-level vulnerabilities. Asbestos [VEK+07, EKV+05], HiStar [ZBWKM06] and Flume [KYB+07] are runtime taint tracking systems that use the DIFC label model from Jif for their taint meta-data combined with process-level tracking granularity.

The motivation to incorporate taint tracking inside the operating system lies in the difficulty to adhere to the *Principle Of Least Privilege (POLP)* [SS75]. POLP requires that each application is split into multiple isolated compartments and each compartment uses only the minimum set of privileges necessary to perform its task. The authors of Asbestos observe that this is cumbersome to achieve in existing operating systems such as Linux using `chroot`-style tools [KEF+05]. In practice, most developers deliver over-privileged binaries that are prone to large-scale data leaks when a vulnerability is discovered.

To support applications structured according to the POLP and simultaneously enforce data flow, the authors of Asbestos suggest to incorporate runtime taint tracking within the operating system. This section describes Asbestos, HiStar and Flume and discusses the challenges that arise from supporting taint tracking inside the operating system.

Asbestos. The main promise of Asbestos is to enforce that data are processed according to the application's data flow policy independently of the correctness of the application [VEK+07]. In Asbestos, an application declares a data flow policy, and the operating system enforces it using runtime taint tracking.

Each Asbestos process can create a new label and use it to protect the data that it generates. Asbestos uses Jif's DIFC model (§2.2.3). Unlike Jif, however, Asbestos does not employ the concept of principals. Instead, each Asbestos label consists of a set of *tags* with a corresponding *sensitivity level*. Tags are opaque bit streams randomly generated by the operating system. Each tag captures a security category, such as "patient documents" or "administrator input". For each tag, there are four sensitivity levels, i.e. 0 to 3, which are used to implement confidentiality and integrity policies (explained next).

From the scope of taint tracking system design, Asbestos uses process-level tracking granularity and DIFC-based labels as taint meta-data. For each process, Asbestos maintains labels that limit how the process communicates with other processes. Its checking operations implement a DIFC "can-flow-to" partial order relation according to the tags and sensitivity levels of two labels, the sender's *tracking label* (L_s) and the receiver's *clearance label* (L_r). To allow communication, each tag in L_s must exist in L_r and the corresponding sensitivity levels in L_r must be higher or equal to the sensitivity levels in L_s. Higher sensitivity restricts data propagation: data labeled using the tag "patient documents" with sensitivity 3 can only flow to processes that have the same tag with sensitivity 3 in their clearance label. Since Asbestos processes communicate only using asynchronous message passing, no information is conveyed to the sending or the receiving process when a message is discarded due to an unsuccessful checking operation.

Asbestos supports both confidentiality and integrity data flow constraints using the four sensitivity levels per tag and two labels per process. The process' tracking label reflects the sensitivity of the data that the process has observed and changes automatically according to the labels of the messages that the process receives. The process' clearance label imposes limits on the changes of the tracking label. When a process p allocates a tag t to enforce confidentiality, the labels of all other processes in the system are automatically updated with t at level 2 for the tracking label and at level 3 for the clearance label. Process p may now protect its messages with t at level 3 so that the tracking label of each receiving process is increased to level 3 on message reception. This effectively enforces confidentiality because the labels of all subsequent messages from any receiving process contain t at level 3. In order to *declassify* data by reducing the sensitivity level, a process must have the $\star$ *privilege*. Such a process is known as being *privileged for tag* t. Since only the process that allocates the tag gets the $\star$ privilege (here that is process p), user data confidentiality is enforced. Levels 0 and 1 are used analogously for integrity data flow constraints.

Process-level tracking granularity introduces the problem of *cross-contamination*. Asbestos encourages data flow containment by using different tags for data that belong to different users. When a single unprivileged process receives such data, its tracking label contains *all* the tags used to protect each individual user's data. This reflects that the process has inspected sensitive data on behalf of all users. Any output data of this process is of limited use because it is "cross-contaminated" with all the tags that the process has received. To remove tags used by specific users, the process must have the $\star$ privilege for each such

tag. Possessing the $\star$ privilege though means that a process may entirely ignore data flow restrictions for a user's tag. Cross-contamination therefore hampers Asbestos' promise for data flow enforcement.

Asbestos avoids cross-contamination using the concept of *event processes*. An event process is a reactive process-like abstraction, which efficiently emulates multiple isolated processes automatically forked on the reception of a message. When an event process receives a message protected with a unique user tag, a new instance of the process is forked with an appropriate tracking label to process the message. Any subsequent messages sent by the instance are only labeled with the particular user tag, avoiding cross-contamination. On completion, the state of this instance is erased, and another instance is forked when a new message arrives. The operating system controls this mechanism and prevents data flow across instances of the same event process.

Expressing high-level data flow policies in Asbestos is challenging. The implementation of taint tracking inside the operating system is geared towards performance and expressiveness but not towards usability. Asbestos may enforce arbitrary data flow policies: it invokes checking operation to check policy compliance on every message dispatch and does not use tracking operations. For the checking operations to be efficient, the taint meta-data, i.e. DIFC labels, are simple collections of tags and levels that are easy to compare. As a result, high-level data flow policies must be translated to low-level taint tracking policies that directly manipulate, amongst others, tags, sensitivity levels, the $\star$ privilege and set up event processes. Asbestos assumes that this translation is performed by trusted code in each application. In practice, this is an error-prone process that may invalidate all the data flow guarantees of Asbestos if a mistake is made.

To facilitate this translation, a policy language has been suggested to declare data flow [EK08]. Developers provide a high-level policy file that describes the processes involved in the data flow and the expected pair-wise communication between them, e.g. "process a can send messages to processes b and c, but b can only reply to a". Asbestos automatically translates this data flow policy to a taint tracking policy and executes the corresponding binaries with the correct tags and privileges.

The main contribution of Asbestos is a runtime taint tracking system design with user-defined data flow policies and low performance penalty for applications. Good performance is achieved by three important design choices: process-level tracking granularity, low-level DIFC-based taint meta-data and message-based inter-process communication. Process-level tracking introduces overhead only when a process communicates with its environment and not on each internal operation (e.g. for each variable assignment). When a checking operation is needed, the tag-based label model for taint meta-data enables self-contained checks that do not rely on a centralised data structure (e.g. Jif's principal hierarchy [ML00]). Finally, message-based asynchronous communication decouples the sender and the receiver, and no additional action has to be taken to avoid disclosing information if checking operations prevent communication. This means that when a label check fails, the message can be

discarded safely (e.g. the sender never learns that the receiver was unable to receive the message). These design choices have influenced subsequent runtime taint tracking systems, such as HiStar [ZBWKM06], Flume [KYB+07] and DEFCon (presented in Chapters 3 and 4).

Asbestos is hard to adopt from a practical viewpoint. To enforce data flow, Asbestos requires both changes to the operating system and a non-trivial application porting effort. Runtime taint tracking changes several parts of the operating system, including the file system, to enable label persistence [VEK+07] and error reporting to enable debugging data flow policy errors [EK08]. Application developers must compartmentalise their applications to avoid code that handles data with different data flow requirements and adopt a message-passing model for inter-process communication. These changes undermine potential adoption because they require a steep learning curve for application developers and result in a significant maintenance effort for the operating system.

HiStar. HiStar [ZBWKM06] is an operating system that leverages runtime taint tracking and aims to avoid covert communication channels, which reduce its ability to enforce data flow. It is influenced by Asbestos and reuses its label model for taint meta-data. In contrast to Asbestos, HiStar requires processes to explicitly change their own labels before exchanging data and thus avoids Asbestos' problem of implicit taint meta-data changes (§2.3.2). In addition, HiStar monitors the storage resources allocated to processes and prevents storage exhaustion, which can be manipulated as a covert channel. These changes improve HiStar's data isolation and data flow enforcement.

Another contribution of HiStar is its support for enforcement of data flow across multiple machines. In Asbestos, only processes with the declassification privilege for each tag in their tracking label are able to exchange data with other machines. This reflects the system's inability to track data flow outside the boundary of the operating system. DStar, HiStar's framework and protocol for distributed taint tracking, introduces the notion of an *exporter* [ZBWM08]. Exporters enable communication across two processes that execute on different DIFC systems. To enforce data flow, exporters rely on the local enforcement of a DIFC operating system and only impose checking operations on data flows across machines. However, since DIFC tags are specific to a machine, there is no inherent correspondence of tags across machines, which can be used in cross-system checking operations. As a result, DStar establishes new, globally meaningful DIFC tags that correspond to the local tags used in each machine.

HiStar's kernel is designed around a small set of constructs; threads, address spaces and memory segments, whose interactions are monitored using DIFC labels. The tracking granularity in HiStar is the memory segment. This is similar from a practical viewpoint to the process granularity of Asbestos but allows HiStar to support shared memory concurrency in addition to message passing. To avoid covert communication channels, HiStar does not permit changes to the labels of most kernel constructs after initialisation.

HiStar supports a Unix-like environment implemented on top of a DIFC-based kernel. Ab-

stractions such as processes, file descriptors or signals are supported at the user level as untrusted library code. Compatibility with Unix facilitates porting of existing software to HiStar with minimal effort, easing the transition from an existing Unix-based environment. Nevertheless, most of the practical challenges for Asbestos' adoption, i.e. operating system maintenance and application compartmentalisation, still apply to HiStar rendering its adoption comparably challenging.

Flume. Flume [KYB+07] is a successor system to both Asbestos and HiStar. Similar to Asbestos and HiStar, it does taint tracking at the process granularity and uses DIFC-based tracking meta-data. In contrast to Asbestos and HiStar, Flume focuses on practicality. It adds taint tracking capabilities to Linux as a user-level library and introduces a simplified DIFC label model that is easier to understand and use.

Flume is implemented on Linux as a user-level reference monitor that intercepts system calls. This design facilitates adoption because Flume benefits from regular Linux updates and broad device support. Linux applications that do not use taint tracking execute without modification. Applications that leverage taint tracking to enforce data flow are linked to a custom `libc` library that interposes calls to the reference monitor. The downside when compared to HiStar is that the trusted computing base increases significantly because it encompasses the Linux kernel.

Flume's DIFC label model avoids Asbestos' sensitivity levels for simplicity and only uses tags. To maintain the ability to enforce both confidentiality and integrity requirements about the data flow in applications, Flume assigns two labels to each process: the *confidentiality* (L_c) and the *integrity label* (L_i). The "can-flow-to" relation in Flume only uses tags. For confidentiality, data flow is permitted from process A to process B if all tags in A's confidentiality label are present in B's confidentiality label (i.e. A's confidentiality label is a subset of B's) and, for integrity, all tags in B's integrity label are present in A's integrity label (i.e. A's integrity label is a superset of B's). As an example, a process with tag t but not tag d in its confidentiality label can send messages to a process with $L_c = \{t, d\}$ but not to a process with $L_c = \{\}$. A process may allocate a tag and use it in either of its labels.

Flume's label model consolidates the privileges of clearance and declassification. On allocation of a new a tag t, Flume assigns to the requesting process the privilege to add the tag to its labels (t^+) and the privilege to remove it from its labels (t^-). When using t to enforce data confidentiality, having t^+ gives a process privilege to receive data protected with t, i.e. clearance. After receiving the data, such a process is restricted by t and thus cannot communicate freely. The only other processes that are able to receive its output are processes with t in their confidentiality labels. The process may avoid this restriction by possessing t^-, which is known for confidentiality as the declassification privilege. A Flume process with data flow requirements typically allocates a new tag but shares with other processes only one of the two privileges.

By extending Linux, Flume avoids the maintenance cost of a new operating system. Adapting

applications to take advantage of taint tracking though still requires significant effort. The authors of Flume report that a version of MoinMoin wiki, a Python web application, required changes to 1000 lines of original MoinMoin code and the addition of 1000 lines of C++ code in launchers for setting up labels and privileges. Such effort renders Flume's adoption in domains such as the web challenging. In the web, most applications are written in interpreted languages, and their authors strive to minimise platform dependence by eliminating the use of features specific to the operating system. At the same time, it is web applications that most often handle data of multiple users and violate the POLP by employing over-privileged components. As described in the next section, runtime taint tracking systems designed for web applications typically track data flow in the language interpreter and use variable- or character-level tracking granularity to avoid modifications to existing code.

Web applications

Runtime taint tracking is a popular technique in the domain of web security and privacy. Web applications typically store sensitive data on behalf of many of users and this makes them an attractive attack target. As explained in Section 2.1.1, the most common attacks on web applications, i.e. injection attacks, exploit developer errors in the propagation of untrusted user data. Multiple researchers suggested runtime taint tracking systems to avoid injection vulnerabilities [NTGG+05, VNJ+07] or, more generally, to guarantee that a web application processes, stores and discloses user data according to a data flow policy [YWZK09, SBL09].

In contrast to system software development, most languages used in the web are interpreted, e.g. PHP, Ruby and Javascript. Runtime taint tracking implementations in the interpreter are easier to adopt compared to a new operating system because they require less pervasive changes to an organisation's infrastructure (although they still incur the maintenance cost of a proprietary interpreter). Nevertheless, such systems have proven effective in enforcing data flow because they focus on well-understood data flow requirements of a single domain, integrate with the programming language and take advantage of common components in the web architecture (e.g. relational databases). Various implementations exist both for the server-side and client-side.

This section begins server-side. Runtime taint tracking systems here either improve the security of the web server that hosts the business logic of a web application or guarantee the privacy of the user data that it processes. The section then covers the problem of how taint meta-data persist across different sessions and finishes with similar taint tracking systems for the client-side layer of web applications in browsers.

Server-side security. Runtime taint tracking is an established technique for improving security of server-side applications written in interpreted languages [Doc12, Fla08, NTGG+05]. Perl and Ruby support taint tracking by default but the checking operations that they impose do not target XSS and SQLI [Foy07, Fla08]. Subsequent research has lead to improved taint tracking systems, which effectively prevent common web application vulnera-

bilities, including XSS, SQLI and vulnerabilities that result from insufficient access control checks [NTGG+05, BMW+11, DKZ09].

The built-in taint tracking implementation in Ruby, known as the Ruby taint mode [Fla08], is representative of most systems described in this section. It is derived from Perl's similar mode [Doc12]. In contrast to the operating systems of the previous section, Ruby uses an object-level tracking granularity. For each object, Ruby stores one bit of taint meta-data that marks the object as "tainted" or "untainted". When a script is executed, the Ruby interpreter marks each object, which stores data that originates from the external environment, as tainted (e.g. the objects that contain the script's command-line parameters). The interpreter automatically invokes tracking operations to propagate taint meta-data each time a Ruby object is used to derive another object. For example, when a string's `slice` method is called, the resulting string slices inherit the taint meta-data of the original string. Similarly, when strings are concatenated, the result taint is the *union* of the taints of the concatenated strings. Developers may explicitly set the taint meta-data of any object using the `taint` and `untaint` methods. Developers should also inspect tainted objects and mark them as untainted when their value is considered secure. Ruby's checking operations then use the one bit of taint meta-data to abort dangerous operations, which should not use tainted parameters. Examples of such operations are importing scripts with `require` or the use of `eval` with a tainted string parameter.

The main limitation of Ruby's taint mode is the lack of configuration options with regard to the data flow policy enforced. Developers can only set the value of the global variable `$SAFE` and select one of four predefined sets of Ruby operations to be protected with checking operations. Ruby thus can only enforce one of four predefined integrity data flow policies, all of which focus on isolating untrusted scripts from their environment rather than preventing web application vulnerabilities. The predefined data flow policies are, however, sufficient to avoid some classes of injection attacks, such as Shell injection or Eval injection. The lack of protection from XSS and SQLI attacks has lead frameworks such as GuardRails [BMW+11] (which is covered later in this section) to implement additional taint tracking libraries that ignore the built-in mechanism.

The first attempt to use taint tracking focused on injection attacks in web applications was in PHP. Anh Nguyen-Tuong et al. [NTGG+05] suggested that performing taint tracking at the granularity of individual *string characters* enables the taint tracking system to automatically mitigate injection attacks without developer intervention. With character-level taint tracking, the system maintains taint meta-data about which individual characters originate from the user and which from the application. This information enables the taint tracking system to sanitise transparently only those characters that originated from the user—the sanitised string can then safely be used by the application. Instead, a system that maintains variable- or object-level taint meta-data can only discard the tainted string as a whole. Any attempts for sanitisation of the whole string would introduce false positives because application-generated code that is part of the string would also be sanitised.

Anh Nguyen-Tuong et al. implement their system as a modified PHP interpreter, which stores one bit of taint meta-data per character and imposes tracking operations on each PHP string manipulation function. In contrast to Ruby and Perl, tracking operations propagate taint meta-data per individual character: e.g. when `substr` is invoked, the characters in the result string are only tainted if the corresponding characters in the original string were tainted. The character-level meta-data are used by the system's checking operations when the application generates its HTML responses (e.g. using `print` statements) or issues SQL queries (e.g. using calls to `mysql_query`). In the former case, the checking operations replace tainted string characters with HTML equivalents (e.g. `<` with `<`). In the later case, the checking operations parse the SQL query and reject it if tainted SQL identifiers or symbols are detected. This technique of using different checking operations according to the operation that the application performs is called Context Sensitive String Evaluation (CSSE) by Pietraszek and Berghe [PB05] and was later extended by GuardRails [BMW+11].

Taint tracking systems that store taint meta-data at the granularity of individual string characters have also been implemented for Java web applications. Halfond et al. [HOM08] use positive tainting, i.e. they mark only those strings that are declared inside the application as tainted. This design is effective against injection attacks, even when the taint tracking system fails to identify untrusted sources of user data. It also requires that developers explicitly mark external sources of data as trusted (e.g. files stored in the local file system) and may increase the runtime overhead for applications that primarily process trusted data. Chin and Wagner [CW09] detail the tracking operations required to achieve character-level tracking while ensuring that existing applications continue to work without modifications. Both systems are realised using a modified Java class library in which selected classes are augmented with taint meta-data and tracking operations, e.g. `String` and `StringBuilder`. This approach enables taint tracking with limited changes to the Java platform but cannot track taint for Java primitive types. Additionally, modifications to the class library introduce incompatibilities with Java Virtual Machines (JVMs) of different vendors due to undocumented assumptions about the implementation of library classes [CW09].

GuardRails [BMW+11] is a runtime taint tracking system for Ruby-on-Rails web applications. GuardRails performs character-level taint tracking but it stores a reference to a *transformer* object as taint meta-data. The transformer object contains a list of transformation (i.e. sanitisation) functions along with the context in which each function should be invoked. For example, a transformer object for XSS may specify different sanitisation functions when a tainted string is found inside the context of a `<script>` tag compared to the context of an HTML page's head. In terms of taint tracking system design, transformer objects enable the developer to choose arbitrary operations to be invoked when the system's checking operations detect violations of data flow policy and enables developers to control which program statements trigger checking operations.

In contrast to PHP and Java, Ruby is a pure object-oriented language with powerful metaprogramming features that allow applications to augment existing library functions and

classes (see Chapter 6 for more details). Using these features, GuardRails is implemented as a Ruby library without changes to the language interpreter.

Runtime taint tracking has also been used to prevent authentication and access control vulnerabilities in web applications. Dalton et al. [DKZ09] observe that a runtime taint tracking system can infer when a user is authenticated, independently of how authentication is implemented in a web application. Their system, Nemesis, extends the PHP interpreter to store two bits of tracking meta-data per string. The first bit, referred to as the "taint" bit, marks strings that originate from the user. The second, the "credentials" bit, marks the user password when it is fetched from the database during a login attempt. Nemesis' tracking operations propagate each of the two bits of meta-data similar to Ruby, e.g. using union-based semantics when two characters are collapsed. Nemesis imposes a single checking operation on PHP equality operations. When a string with the "tainted" bit set is equal to a string with the "credentials" bit set, Nemesis assumes that the equality indicates a successful authentication attempt. This allows a shadow authentication and access control system to detect when a user logs in and to prevent security vulnerabilities when developers omit access control checks in their applications.

Server-side privacy. Runtime taint tracking systems were also suggested to protect the privacy of user data in the server. The taint tracking policies that such systems support are more configurable compared to their counterparts that prevent injection vulnerabilities; developers may modify the taint meta-data and control when checking operations are invoked. Runtime taint tracking systems that improve server-side security are designed to track data flows specific to injection attacks and rarely offer any configuration options (GuardRails' transformer objects enable control over the checking operations). Instead, user data privacy depends on the semantics of each application, and there is no single data flow policy suitable for all applications. Therefore, to guarantee user data privacy, the taint tracking system must be able to enforce arbitrary data flow policies.

Resin is a runtime taint tracking system for PHP web applications, which improves server-side security and can be used to guarantee user-data privacy [YWZK09]. For example, Resin can enforce that a user's password is never disclosed by an application despite implementation errors. In Resin, taint tracking is performed at the character granularity and the taint meta-data and the tracking and checking operations are all configurable.

Resin requires developers to annotate the user data that they want to protect with references to *policy objects*. In contrast to label-centric taint meta-data, policy objects specify the data flow policy as PHP code. For example, assume that a web application must ensure that a password is only released via email to the user that it belongs to. Instead of encoding this as a label attached to data, a policy object may store the name of the password owner, logic to compare the name to the current user and actions to take when these two do not match. To encode data flow policy similar to this, Resin policy objects expose a `check` method that receives information about the context of where user data are about to be used. An example

```
1  class PasswordPolicy extends Policy {
2    private $email;
3    function __construct($email) {
4      $this->email = $email;
5    }
6    function check($context) {
7      if ($context["type"] == "email" && $context["email"] == $this->email)
8        return;
9      global $Me;
10     if ($context["type"] == "http" && $Me->privChair) return;
11     throw new Exception("unauthorized disclosure");
12   }
13 }
```

Listing 2.4: A Resin policy object that ensures user passwords are never disclosed via email to the wrong recipient [YWZK09].

of this policy, as implemented by Resin's authors [YWZK09], appears in Listing 2.4.

Resin's tracking operations are invoked on string operations to propagate references to policy objects. In most cases (e.g. in calls to `substr` or in string concatenation), this is straightforward. Policy objects though may need to be merged, for example, when two characters are added to calculate a checksum. In these cases, Resin merges the policy objects involved and generates a new policy object that calls the `check` methods of both initial objects. This is another version of Ruby's union semantics. However, this mechanism in Resin can be controlled by overriding the `merge` method of policy objects.

Resin also supports configurable checking operations implemented as *filter objects*. A generic filter object is attached by default to all PHP functions that communicate data between the application and its environment, e.g. functions used for file input/output. Each filter object implements two methods, `read` and `write`. When data are returned from or passed to a function protected by a filter object, the corresponding filter method is invoked. Typically, the filter method inspects any policy objects associated with the data and calls their `check` method. According to its result, the function call may be aborted. In the password example, it is a filter object attached to PHP's `mail` function that calls the policy object's `check` method and guarantees that the policy in Listing 2.4 is enforced.

Resin is the most flexible character-level taint tracking system. Due to its ability to configure taint tracking policy, it is suitable for guaranteeing user data privacy and for improving security. Resin is implemented as a custom PHP interpreter and its authors suggest additional changes to the web server, i.e. to have the web server directly call the `check` methods of policy objects attached to static files. However, unless taint tracking is considered part of the PHP language specification and is officially adopted, third party implementations are impractical. As the PHP manual puts it:

> *"modifications to the Zend[9] engine should be avoided. Changes here result in*

[9] Zend is the name of the official PHP interpreter.

> *incompatibilities with the rest of the world, and hardly anyone will ever adapt to specially patched Zend engines. Modifications can't be detached from the main PHP sources and are overridden with the next update using the "official" source repositories. Therefore, this method is generally considered bad practice"*

In the past, taint tracking support has been suggested as a feature to the PHP community but it has not been adopted, partly because of fears that it may lead to a false sense of security for inexperienced developers [Ven06].

Taint meta-data persistence. Data flow in web applications typically involves a database. Using a database, traditional injection attacks such as XSS become more effective: an adversary may store a persistent malicious string and attack every user without requiring that each user clicks on a suspicious link. Web applications may also store in the database sensitive data for multiple users. Thus, bugs in the web application may result in unauthorised data disclosure for any of its users. Runtime taint tracking systems cannot effectively improve security or user data privacy without tracking data flow in the database. For example, if a user-provided string is stored in the database and taint meta-data are lost in the process, that string may still be used in injection attacks when the application later retrieves it.

The first systems that performed taint tracking in web applications did not store taint meta-data in the database [NTGG+05, PB05, HOM08]. Pietraszek and Berghe [PB05] treat all data stored in the database as untrusted and mark them as tainted when fetched by the web application. This approach may lead to false positives if the application sanitises user data before it stores them in the database. The alternative, i.e. considering all data stored in the database as untainted, is prone to false negatives. A false negative may result in undetected injection attacks when the web application omits sanitisation.

DIFC-J [PB05] associates taint meta-data with individual database columns. It is a Java taint tracking system that tracks taint at string granularity. Developers specify the taint meta-data of each database column. When the application attempts to store data in a database column, a checking operation verifies that the meta-data of the input matches the meta-data of the column. When data is read from a column, a tracking operation sets the meta-data of the output to the meta-data of the column. Web applications, however, may store in a single column data that belong to multiple users, e.g. user passwords. DIFC-J's approach requires that all such data are labeled the same. Such a design therefore does not support the use of separate labels per user and prevents the taint tracking system from tracking the data of each user individually.

Resin improves upon DIFC-J and associates taint meta-data with individual database cells. Resin initially intercepts SQL `CREATE TABLE` queries and rewrites them to introduce one additional meta-data column for each data column in the query. When SQL queries insert data to the database, Resin determines the policy objects that correspond to the data of each cell, serialises them and stores them in the meta-data column. Similarly, when data are fetched from the database, any associated policy objects are deserialised and attached

to the result query. From the perspective of the taint tracking system, this design fully maintains the character-level taint meta-data and renders the database equivalent to any other non-persistent data structure.

A comprehensive attempt for storing taint meta-data into the database is DBTaint [DC10]. DBTaint follows Resin's design and stores taint meta-data separately for each database cell. In contrast to Resin, DBTaint uses the composite types available in PostgreSQL[10] and extends all default data types with a taint meta-data field. DBTaint is implemented as a custom JDBC driver, which transparently rewrites application queries.

DBTaint supports prepared statements that, due to composite types, require additional taint parameters. When using prepared statements applications bind data to placeholders in queries using numeric indexes. Since the indexes that the application provides will no longer match the indexes in the transformed queries, DBTaint modifies the bind indexes provided by the application with indexes that correspond to how the query is rewritten. Overall, DBTaint does not require any changes to the database server or the application. It may also be retrofitted to support most taint tracking systems only with minor edits to the type of the taint meta-data that it stores.

Client-side security. Runtime taint tracking has been used at the client-side, i.e. in the the browser, to reduce the impact of XSS vulnerabilities in web applications. Vogt et al. [VNJ+07] suggest a taint tracking system for Javascript, which stops XSS attacks in the browser as they occur. The taint tracking system detects flows of sensitive data from the browser to any domain other than the page's initial origin and asks the user for approval when this happens. In contrast to server-side systems, the injected Javascript code is executed but the taint tracking system stops its attempts to leak data. Taint tracking thus acts as an additional security layer in the browser, which is independent of a developer's ability to eliminate XSS vulnerabilities.

Vogt et al. follow a design similar to Ruby's taint mode, i.e. variable-level tracking granularity, one bit of tracking meta-data and non-configurable tracking and checking operations. Sensitive user data, such as the session cookie, which identifies a client to the web server, are marked as tainted. Checking operations use the taint meta-data to prevent sensitive data flow to third-party domains. Notice though that, at the client-side, the benevolent developer assumption does not hold: malicious scripts injected via XSS may actively try to evade the taint tracking system. This forces the runtime taint tracking system to track both control and implicit data flow (§2.3.2).

Control flow occurs when information is encoded in the control statements of a program. For example, consider the script in Listing 2.5 injected via XSS to a page. When executed, the script inspects the user's tainted cookie and tests the session identifier for a given value (line 10). If the script is allowed to contact a remote server as a result of the comparison,

[10]PostreSQL Documentation: Composite Types, `http://www.postgresql.org/docs/8.1/static/rowtypes.html`, last accessed: 5/9/2012

```
1  <script type="text/javascript">
2      if (document.cookie==null) return;
3      var cookies=document.cookie.split(";");
4      for (i=0; i<cookies.length; i++)
5      {
6          var k=cookies[i].substr(0,cookies[i].indexOf("="));
7          var v=cookies[i].substr(cookies[i].indexOf("=")+1);
8          if (k=="SSID")
9          {
10             if (v=="AAYSMkxHM5324JPQ")
11             {
12                 var l=document.getElementById('logo');
13                 l.src="http://attack.com/img52.jpg";
14                 break;
15             }
16         }
17     }
18 </script>
```

Listing 2.5: Control flow in Javascript. The script discloses a particular value of the session identifier (line 10) to a remote server without explicit data flow via variable assignments. The image URL (line 13) may or may not correspond to an actual image.

e.g. by changing the `src` property of an image (line 13), this conveys to the remote server `attack.com` the outcome of the comparison.

To track control flow, the taint tracking system associates one bit of taint meta-data to *scopes of code*, i.e. sets of statements, which may execute as a result of a control statement. In the example above, a scope spans all the instructions from lines 12 to 14. A scope becomes tainted when it is executed as a result of an `if` or `while` statement with a tainted expression in its condition. Every variable assignment that occurs inside a tainted scope generates a tainted result and every static value defined there is considered tainted. In the example of Listing 2.5, the constant string in line 13 would be considered tainted, and the system would prevent data disclosure. For implicit data flow, Vogt et al. employ a simplistic form of static data flow analysis. Nevertheless, their system is inherently unable to prevent XSS attacks that do not involve a remote domain, e.g. an XSS attack that covertly changes the user's password will go undetected.

Another source of security vulnerabilities in the client is a browser's extensibility mechanism. In Firefox, extensions are implemented as privileged Javascript code that freely interacts with individual pages and remote domains. This is in contrast to untrusted scripts embedded in web pages, which are restricted to manipulate pages only from the same domain. In practice, however, an untrusted web page script may trick a vulnerable extension script to execute on its behalf, for example, to `eval` untrusted Javascript code. If this happens, the code is executed with the extention's full privileges, and thus a web page script may take control of the user's browser.

Djeric and Geol [DG10] observe that most extension vulnerabilities in Firefox can be modeled as violations of the integrity of data flows and suggest to use runtime taint tracking to mitigate attacks. Their system marks all objects defined in untrusted page scripts as tainted and checks plugin calls to `eval` and dynamic function calls for potential tainted parameters. Dynamic function calls can be used to execute arbitrary functions if the target function of the call is chosen by an attacker. Djeric and Geol implement taint tracking by modifying the Javascript interpreter following the simple design from Ruby: object granularity, one bit of taint meta-data and checking operations attached to specific Firefox functions.

Client-side privacy. Web applications typically combine client-side code from multiple sources to display rich user interfaces, create mashups or show advertisements. Existing browser security mechanisms, however, cannot control how user data flow to different servers in the presence of third-party source code. Researchers have suggested browser-based taint tracking systems to analyse data flow [DJLS10] and to enable the browser to enforce data flow constraints in web applications [YNKM09, SBL09].

Jang et al. [DJLS10] implement taint tracking in the Chrome browser and use it to analyse 50,000 popular websites for privacy-violating data flows. Similar to Resin, their system offers configurable taint tracking policies. Users may attach generic Javascript objects to variables as taint meta-data and configure the location and actions of checking operations. In contrast to Resin, their implementation also tracks control data flow by associating taint meta-data with scopes of code. Their system favours simple strings for taint meta-data and does not provide extensible tracking operations.

The flexibility in devising taint tracking policies allowed Jang et al. to detect various privacy-violating data flows at a large scale. The authors crawled websites while tracking different types of privacy-violating data flows. Examples include monitoring of user mouse movements and attempts to disclose the history of visited pages. Their results identified hundreds of offending web sites, showing the practicality of taint tracking for data flow analysis. Since their system is purely dynamic, however, it cannot detect implicit data flow. This allows websites that are aware of the taint tracking system to evade analysis.

BFlow [YNKM09] and xBook [SBL09] are two taint tracking systems that can additionally enforce data flow policy in the browser. They suggest to do this for third-party applications embedded in web sites that offer extensibility mechanisms, e.g. Facebook. When such a third-party application receives sensitive user data from the host website, it is typically able to communicate with external servers without restrictions. BFlow and xBook suggest to track the sensitive information that third-party applications receive and to enforce that their external communications adhere to a user-approved data flow policy.

BFlow and xBook are effective against malicious code and support configurable data flow policies for different applications. They constrain malicious code through tracking taint meta-data at "zone" granularity. Zones are groups of scripts that monitor and control communication with other scripts and external servers. Communication across zones occurs

through asynchronous message passing. This avoids the implicit flows inherent to tracking taint at variable granularity (§2.3.2). To support configurable data flow policies, both systems use DIFC labels as taint meta-data. Labels are initialised per zone according to a user-approved manifest file or according to the server's response. Checking operations employ the "can-flow-to" relation to enforce data flow policy.

The main limitation for BFlow and xBook is their reliance on server-side counterparts to track data flow effectively. In BFlow, the server is expected to return correctly labeled data according to the label of the incoming request and process data with different labels separately. BFlow's authors suggest using an DIFC-based operating system for this purpose. In xBook, a dedicated trusted server hosts third-party applications and tracks data flow with similar mechanisms as in the browser. Such requirements make the adoption of the taint tracking system hard as neither users nor web application developers may benefit from data flow enforcement without the other party also adopting the suggested system, as well. Hails [GLS+12], a similar system for confidential data privacy in the web, requires both server and client-side components, and in addition, web applications must be written in Haskell.

The above discussion highlights the applicability of taint tracking in the web, spanning from tackling well-understood problems, such as injection vulnerabilities, to enforcing application-specific data flow policy. Most of the systems, however, that implement taint tracking introduce unrealistic requirements (e.g. use of a proprietary interpreter or a modified browser), which hamper adoption.

Java applications and taint tracking via code rewriting

Data flow analysis and enforcement is useful in other application domains. Modern mobile applications and traditional desktop applications, for example, handle and communicate user data in practically unconstrained fashion. Privacy-conscious users are limited to use only applications from developers that they fully trust. The breed of DIFC operating systems presented earlier provides a pathway for enforcing data flow policy in the future but it is seldom practical for analysing existing applications or enforcing data flow without significant re-engineering effort.

Different researchers have suggested taint tracking systems which incur few changes to the execution platform either because they focus on the Java class library or because they apply taint tracking via code rewriting. Both techniques are easier to maintain and adopt than a novel operating system. As presented in the previous section, easy-to-maintain implementations of taint tracking for Java leverage the class library of the language and avoid modifying the bytecode interpreter [CW09, HOM08]. Applications such as web browsers, instant messengers and text editors, however, are often written in languages that are compiled to machine code. To support taint tracking there, code rewriting is a simple but sometimes inefficient choice. This section concludes with research related to data flow tracking in Java applications and with code rewriting.

Java applications. Taint tracking systems that track data flow inside Java applications are implemented either by changing the semantics of bytecode operations to propagate taint meta-data [EGC+10, NSCT08] or by using a modified class library [CW09, HOM08] as presented previously. A customised bytecode interpreter inside the Java Virtual Machine (JVM) is more accurate because it can track data flow that occurs via primitive types but, as similar systems for PHP [NTGG+05] and Python [YWZK09], is hard to maintain in practice.

TaintDroid [EGC+10] is an extension to the Android operating system[11] that analyses data flow within third-party applications. Android applications are written in Java and use a manifest file to specify the permissions that they require from the operating system. Android permissions control application access to sensitive user data but are limited in practice. If, for example, the user accepts an application's requests for the internet access permission and for a permission that gives access to sensitive user data, there is no mechanism to track if the application discloses the data that it receives. TaintDroid's purpose is to detect such behaviour and enable real-time privacy monitoring on existing Android applications.

TaintDroid performs taint tracking for DEX, a machine language equivalent to Java bytecode that runs on Android's Dalvik VM. TaintDroid executes DEX bytecode and stores 32 bits of taint meta-data per DEX variable. This is equivalent to tracking data flow at the granularity of Java variables. Each of the 32 bits of taint meta-data corresponds to one source of sensitive information, e.g. one bit marks data that originated from the GPS location sensor and another bit marks data from the phone agenda. TaintDroid propagates taint meta-data in each DEX operation following similar semantics as Ruby's taint mode, considering each meta-data bit individually (i.e. bits at different positions are never collapsed in a single bit). When the analysed application sends information to external servers, tracking operations log attempts with non-zero taint meta-data. Since applications are not expected to evade taint tracking actively, control and implicit flows are not addressed. Enck et al. [EGC+10] use TaintDroid to analyse multiple popular Android applications and report that one third of the tested applications disclose to third parties private data, such as the device ID and the SIM serial number.

A JVM that offers data flow enforcement for generic Java applications is Trishul [NSCT08, NSCT07]. Similar to TaintDroid, Trishul uses a set of bits as taint meta-data and a variable-level tracking granularity. In contrast to TaintDroid, Trishul supports configurable data flow policies. Using a policy file, the user controls the instantiation of taint meta-data as well as the location and actions of checking operations. Trishul is implemented by modifying an open source JVM that is no longer maintained [BHL00]. To improve performance, extensions of Trishul integrate with the Just-In-Time (JIT) compiler and take advantage of unused Streaming SIMD Extensions (SSE) registers to store taint meta-data [NSCT08]. Nevertheless, Nair et al. report a 167% overhead compared to the unmodified JVM when using Trishul with CPU-bound applications.

Trishul assumes a malicious developer and thus tracks control and implicit data flow (§2.3.2).

[11] The Android Open Source Project (AOSP), `http://source.android.com/`, last accessed: 13/9/2012

This is achieved by combining runtime taint tracking with static data flow analysis [NSCT07], which identifies the set of variables that are updated in each program scope. This analysis occurs when a class is first loaded. Every time that Trishul is about to execute a control statement, it uses the results of the static analysis to identify all variables that may change in the statement's alternative branches. It then updates the taint meta-data of *all* those variables with the meta-data of the control expression independently of whether each particular branch is executed. This guarantees that every variable has updated taint meta-data, even if the variable is not updated as part of the specific branch taken at runtime.

Tracking implicit data flow in Trishul is more secure but may result in overly conservative meta-data propagation and increased numbers of false positives. The important limitation, however, with both systems is that, without support from the official platform vendors, adopting a custom JVM is challenging in practice. For example, TaintDroid's implementation in Android 2.1 was quickly made obsolete by the JIT-enabled Dalvik release with Android 2.2[12].

Source and binary code rewriting. An alternative technique to implement runtime taint tracking is code rewriting. Instead of modifying the execution environment, a code rewriting system transforms each application to propagate taint meta-data explicitly. This is achieved by introducing additional statements in the language that the application is written in. The main advantage of code rewriting is that the resulting applications are compatible with existing execution environments. This avoids the maintenance effort for custom operating systems or interpreters and facilitates adoption. For applications distributed in binary form, code rewriting via binary instrumentation or even performed by an emulator are both practical alternatives compared to custom hardware [SLD04, CXN+05]. An important limitation of code rewriting is its runtime overhead. Explicit tracking operations are invoked before each program operation, and they are often slower than the program operation that they intercept. Therefore an important challenge for code rewriting systems is how to optimise the tracking operations without sacrificing the precision of taint propagation.

Xu et al. [XBS06] use code rewriting to implement runtime taint tracking for C applications. Their system enforces integrity data flow policies to prevent injection-style memory exploits, such as the dereferencing of a user-provided pointer or using untrusted user data for `printf`'s format parameter. The taint tracking policy in their system uses one bit of taint meta-data per memory byte, tracking operations similar to Ruby and a simple policy language to configure the location and actions of checking operations.

Xu et al. propagate taint meta-data by transforming every C assignment and arithmetic/bit expression to fetch the taint of the values involved and to taint the result appropriately. Taint meta-data are stored in a global bit-array indexed by memory address. This correctly captures the shared taint meta-data when a single memory location is aliased by multiple

[12]Dalvik JIT, Android Developers Blog, `http://android-developers.blogspot.co.uk/2010/05/dalvik-jit.html`, last accessed: 13/9/2012

pointers but introduces significant overhead: transformed C operations on local variables now require access to a global array. Such accesses disrupt locality optimisations performed by the compiler. Xu et al. report that their unoptimised transformations cause an application slowdown by a factor of five and suggest a number of optimisations for storing taint meta-data to improve performance. Despite these optimisations, their final prototype slows CPU-intensive applications by 76% on average.

Chang et al. [CSL08] suggest a similar system that targets C applications and uses source code transformations. To reduce the impact on performance at runtime, the system performs static taint analysis on the transformed application. The taint analysis identifies portions of the application, in which data flow is known in advance therefore no data flow tracking has to occur at runtime. This approach significantly improves performance but its applicability in dynamic languages that are harder to analyse statically remains to be explored. Nevertheless, we later show (cf. §5) that a similar approach can be applied to dynamic languages and can lead to similar performance improvements.

Argos [PSB06] modifies an emulator, Qemu [Bel05], to implement taint tracking. In contrast to Xu et al. Argos does not create binaries that perform taint tracking explicitly. Instead, Argos extends the target code that Qemu generates while translating blocks of instructions for execution to also propagate taint meta-data. This technique is similar to source code rewriting but it has the advantage that it can be applied on third-party binaries even when no source code is available. The disadvantage of the system, however, is increased performance overhead at runtime, which makes applications execute 15–30 times slower. For this reason, Argos was primarily suggested for malware analysis.

A similar system that targets Windows x86 binaries is TaintEraser [ZJS+11]. TaintEraser rewrites x86 instructions to propagate one bit of taint meta-data per memory byte. Binary rewriting is performed using Pin[13], a generic x86 instrumentation tool. The use of Pin slows down tracking operations significantly because Pin maintains a second call stack and swaps CPU registers when switching from the application call stack to propagate taint meta-data. Zhu et al. report that an unoptimised version of TaintEraser slows applications down by up to two orders of magnitude. In some cases, the transformed binary is no longer usable because internal timeouts stop the program's execution, e.g. Yahoo Messenger attempts to sign-in for a predefined interval and then aborts.

To improve the performance of their system, Zhu et al. introduce *function summaries*. A function summary is a tracking operation that calculates the taint meta-data of a function's return value given the taint meta-data of its input parameters. They notice that a small number of operating system functions account for large portions of application execution time. Since these functions have well-defined semantics, generic function summaries embedded in TaintEraser achieve taint propagation without the cost of instruction-level instrumentation. In spite of the extensive use of function summaries, a transformed version of Internet Explorer is 1.4–4.6 times slower than the original.

[13]Pin: A Dynamic Binary Intrumentation Tool, `http://www.pintool.org`, last accessed: 25/9/2012

Code rewriting is a practical method to implement taint tracking because it requires no modifications to existing applications or execution platforms. However, it suffers from increased overhead. Recent attempts that focus on optimising checking and tracking operations at the binary level [KGJK12, BSB11] incur less overhead but slowdowns of 2-4× are still typical. In Chapter 5, this thesis shows that by performing taint tracking only inside part of an application, the performance overhead of code rewriting is significantly reduced.

2.4 Summary

This chapter provided an introduction to the problem of data flow analysis and presented methods that researchers employ to enforce data flow policy in applications.

We begun with an overview of security challenges in web applications and event processing systems. We observed that data flow confidentiality and integrity are two complementary concepts useful to capture various data propagation policies. In web applications, injection attacks are a violation of a simple integrity policy that should restrict the use of untrusted user data in specific application operations. In event processing, applications that handle confidential data require restrictions on data disclosure that event processing systems do not currently provide.

The chapter continued with a discussion of static methods that are used to analyse and enforce data flow. We presented taint analysis (i.e. a form of data flow analysis), symbolic execution (i.e. the emulation of an application's actions under every potential input) and security-typed languages (i.e. languages that capture data flow requirements as part of their type system). Static methods do not introduce runtime overhead and are effective even with malicious code. However, they have limited applicability when the data flow policy relies on runtime information, cannot analyse applications that use dynamic features of a programming language and are overly conservative resulting in false positives.

We then focused on runtime taint tracking and introduced a model to capture the important properties of relevant systems. Characteristic properties of every taint tracking system are its tracking granularity, the type of taint meta-data that it maintains and the operations that it invokes at runtime to propagate or check taint meta-data (tracking and checking operations, respectively). Taint tracking introduces overhead at runtime, yet it is a simple and accurate technique suitable for data flow analysis and enforcement in systems that are hard to analyse statically.

The chapter closed with a detailed presentation of runtime taint tracking systems suggested in the literature, with a particular focus on applications of taint tracking in operating systems and web applications. We showed that while past research attempts have explored many design options, the suggested implementations often require extensive modifications to the underlying infrastructure or incur important performance penalties. In particular, taint tracking systems implemented as part of the operating system support generic applications but require a significant effort to port existing applications and are hard to maintain. Code

rewriting was shown to offer a practical alternative for implementing taint tracking and support existing applications but performance suffers.

In the next chapter we describe DEFCon, a high-performance event processing system with data flow enforcement capabilities. DEFCon only supports event processing applications in Java. This removes much of the complexity that stems from supporting arbitrary Java applications and enables a taint tracking implementation that is efficient and is easy to maintain with future versions of the Java platform.

Chapter 3

Taint Tracking for High-Performance Event Processing

This chapter presents DEFCon [MPE+10a], a distributed event processing system written in Java with support for data flow enforcement using runtime taint tracking. DEFCon improves the state of the art in runtime taint tracking because at the same time (1) it incurs minimal overhead and can support demanding event processing applications, e.g. in finance, (2) it allows users to modify extensively the data flow policy that it enforces, and (3) it is implemented with easily-reproducible changes to Java library classes. A central observation in DEFCon is that event processing applications need only access a subset of the functionality provided by the Java libraries, and this is used to simplify the implementation of the system. DEFCon shows that taint tracking can benefit specific application domains without the problems that are commonly associated with it, i.e. changes to existing infrastructure and runtime overhead.

DEFCon introduces DEFC, a taint tracking policy that uses labels to enforce data flow requirements both inside a single system and across systems that belong to different domains. Data flow requirements are expressed in DPL [MPE+10b], a high-level policy language to specify the flow of events between distributed software components. DPL policies are translated to DEFC labels for enforcement. DPL provides an expressive and practical language for most common data flow policies while DEFCon achieves efficiency by using labels for enforcement. To guarantee that software components only exchange data via channels that it monitors, DEFCon uses a lightweight isolation scheme, which avoids extensive modifications to the Java class library.

This chapter covers the single-node design of DEFCon. We begin with an overview of the performance requirements in event processing and the threat model for attacks (Section 3.1). We then cover the main features of DEFCon, i.e. the DEFC taint tracking policy for enforcing data flow in Section 3.2 and the Java isolation mechanism in Section 3.3. Section 3.4

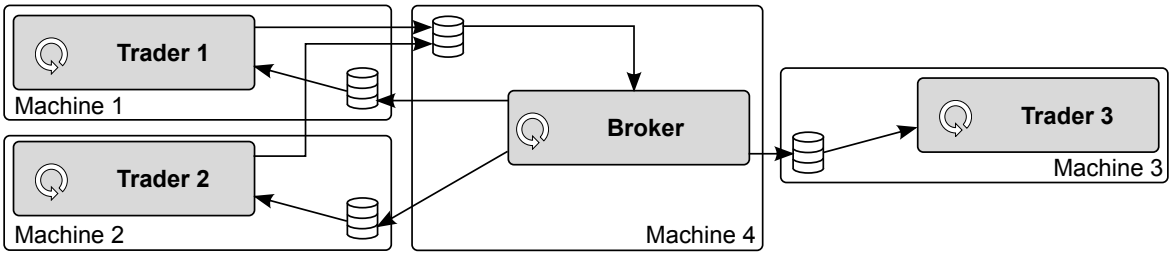

Figure 3.1: An example of an event processing system for stock trading. Event processing units exchange events via message queues.

presents the implementation of the DEFCon prototype and its low-level API for specifying data flow policy. Finally, we evaluate DEFCon in Section 3.5, and the chapter finishes with a summary in Section 3.6.

3.1 Requirements

DEFCon supports event processing applications [Luc02]. The unique property of event processing is that data are structured as event messages, or in short, *events*. The processing logic of the application is implemented as individual event processing *units*. Each unit receives, processes and subsequently emits events. An event may be processed and transformed by multiple units. Units are deployed on physical machines, yet they may perform processing on behalf of a different administrative domain than the one that controls the physical machines. The *event processing system* is responsible for event dispatch between units, either within a single machine or, via the network, across multiple machines.

Event processing systems are used frequently in financial data processing and algorithmic trading[1] because they facilitate scalability and distribution (§2.1.2). In algorithmic trading, traders implement *trading strategies* as processing units that issue stock orders, i.e. events. Another unit (i.e. a broker) receives these orders, matches them and generates additional events to notify traders about successful transactions. Figure 3.1 illustrates such a scenario, in which clients perform stock trading. Each client implements the core of its trading logic with an event processing unit, e.g. Trader 1 and Trader 2. The broker is also implemented as a unit that receives order events, matches them and publishes trade events. Trade events are communicated to all other clients in the system, e.g. Trader 3, as part of a stock tick feed. Communication in this example occurs via message queues.

Such a trading scenario introduces important performance challenges because high performance, e.g. low stock tick processing latency, directly translates to higher profit [Fla07]. The reason why low latency increases profits is that when opportunities arise, the transactions of traders that react first affect stock prices and this makes future transactions for slower traders less profitable [Duh09]. Traders thus place their event processing systems in close physical

[1] Apache ActiveMQ use cases, `http://activemq.apache.org/use-cases.html`, last accessed: 25/9/2012

proximity to the broker to minimise latency (a practice known as *co-location*). This is, however, expensive because order matching (also known as *brokering*) is typically performed at major stock exchanges and nearby space is limited [Exc09]. Most traders who cannot afford co-location accept the additional delay of receiving the stock tick feed remotely. As a result, they have a disadvantage compared to competitors with co-located systems.

A way to reduce costs for smaller traders is to share event processing infrastructure co-located with the stock exchange. Instead of buying hardware and space separately, they can share a single execution platform and split the co-location costs. As mentioned in Section 2.1.2, such a shared platform may additionally match buy/sell orders between traders and act as a broker, a scenario known as a "dark pool". Even smaller traders, however, want to keep their trading algorithms secret from the competition. Confidentiality is therefore critical if a single machine executes processing on behalf of multiple traders. The same is also true for integrity: a malicious trader should never be able to manipulate the trading strategies of other traders in the same system, e.g. by tricking them into buying stocks of their choosing.

The next two sections present first the performance requirements of a simple *pairs trading* strategy and then the security requirements for a scenario in which mutually-distrustful trading strategies are hosted by a shared machine.

3.1.1 Performance requirements (latency and throughput)

Pairs trading [Vid04] is a trading strategy based on the observation that some stocks have highly correlated prices and eventually converge to their past price ratio after temporary divergence. A simple example of pairs trading involves the following three steps:

1. Identify two stocks that have had historically a high correlation. Calculate the mean and the standard deviation of their price ratio and decide on the price ratio, which triggers the decision to trade.
2. Start observing the stock prices of the two firms. When their ratio becomes greater than the triggering ratio from step 1, buy shares of the stock with the low price and sell shares of the stock with the high price.
3. Continue observing the stock prices. Once their price ratio drops below the ratio from step 1, reverse the positions from step 2.

Latency is crucial for successfully tacking advantage of pairs trading. The actions of step 2, when performed collectively by many traders, tend to move the market in the opposite direction, i.e. the price of the cheaper stock increases and the price of the more expensive stock decreases. Those traders who react first have the largest profit and limit the profit margin for slower, remote traders.

Event throughput is also important when multiple traders use pairs trading in a shared, co-located machine. In order to support monitoring of arbitrary stocks, the event processing system must dispatch the incoming stock tick events from the stock exchange to every

processing unit. This is a demanding task when the number of traders and the event rate in the stock tick feed are high (see Section 3.5 for a detailed discussion). In order to support monitoring arbitrary stocks without limiting the number of units in the system, events must be transfered at high rates between units.

3.1.2 Security requirements (threat model)

The main challenge for an event processing system that hosts processing on behalf of different clients comes from unauthorised parties perceiving or influencing information contained in events. The event processing system should guarantee that events are only accessed by their respective recipients (event confidentiality) and that only specific units are able to modify the content of particular events (event integrity).

The event processing system should protect from event processing units performing intentional violations of event confidentiality and integrity, and from unintentional violations, in which units contain bugs. Examples include developer errors in the implementation of units that disclose sensitive events or allow unauthorised parties to affect the content of events; back-doors introduced by developers to enable unauthorised access to data; and inconsistent handling of events by units at runtime due to developers understanding data flow policy differently. The underlying cause for all these problems is that the enforcement of data flow policy depends on the correctness of the implementation of every unit that processes an event. In practice, any unit in a system can potentially violate the confidentiality of the events that it receives and destroy the integrity of the events that it emits.

The event processing system should not focus on protection from malicious code provided by untrusted third parties, which may attack the event processing system itself. A reasonable assumption is that the event processing system is a closely guarded asset of an organisation and the organisation only permits the execution of code on behalf of accountable parties. The threat model therefore does not consider cases, in which the adversary exchanges data between processing units by monopolising system resources such as the main processor (see also the discussion of covert timing channels in §2.3.3). This is again reasonable to assume because such channels are unreliable and typically offer low bandwidth. Similarly, the threat model does not include adversaries that employ denial-of-service attacks. Such attacks often render the system unusable and, as such, it is unlikely that they are employed by legitimate users of the system—protection against them is left for future work.

Overall, the event processing system should protect from adversaries that attempt to learn data that must otherwise be kept private or affect the processing of other users by introducing bogus data. Adversaries that are willing to attack the event processing system itself are not considered in this work. The operating system, the language runtime and the event processing system are assumed to be trusted because they are maintained by the organisation that supports the event processing system. This assumption reflects the observation that a service can only be as trusted as the organisation that maintains it.

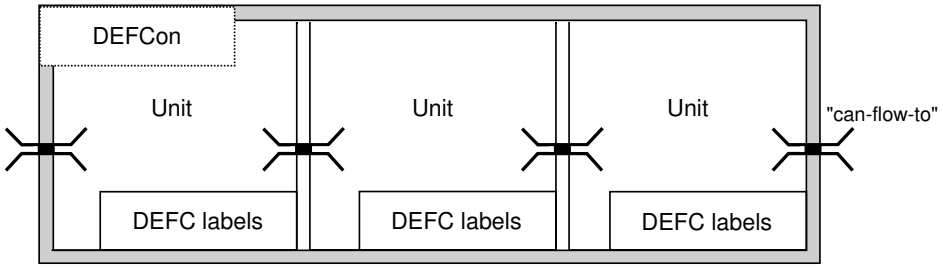

Figure 3.2: DEFCon as a runtime taint tracking system. DEFCon uses unit tracking granularity and labels that consist of tags for taint meta-data. Checking operations inspect taint meta-data for every event that units emit or receive.

3.2 Decentralised Event Flow Control

Decentralised Event Flow Control (DEFC) is a taint tracking policy (§2.3.1) that constrains the flow of events in an event processing system. DEFC provides to unit developers a set of methods to devise arbitrary data flow (or *event flow*) policies. For this, it exposes primitives to control the value of the taint meta-data and therefore the outcome of checking operations.

Figure 3.2 shows DEFC in the context of the runtime taint tracking model from Section 2.3.1. DEFC associates taint meta-data with the state of each unit and, in addition, it associates meta-data with the individual events that units emit (not shown in the figure). Checking operations involve two steps: when a unit emits an event and before a unit receives an event. Upon emitting an event, checking operations enforce that the taint meta-data of the event correctly reflect the taint meta-data of the unit. Upon delivering of an event, checking operations ensure that the receiving unit is eligible to receive the event. Together these two steps enforce that the "can-flow-to" relation (see Section 3.2.2) holds between the sending and the receiving unit. Unit taint meta-data and event taint meta-data are specified by DEFC.

The novel features of DEFC are aimed specifically at event processing. These are:

1. Each event is associated with multiple labels to support fine-grained data flow tracking (see Section 3.2.1).

2. Separate *delegation privileges* control the ability of a unit to delegate privileges to other units—this allows units to enforce that their events are processed by a particular sequence of units (see Section 3.2.3).

3. Privilege propagation occurs dynamically via *privilege-carrying events*, a mechanism that uses the event-processing system to avoid introducing covert channels (see Section 3.2.5).

4. Units can process only event parts that they are allowed to access without cross-contaminating every other part stored in the event (see Section §3.2.6).

As explained next, these features of DEFC facilitate taint tracking when events store data that adhere to different data flow policies and when events are are transformed by multiple units in a sequence.

3.2.1 Anatomy of an event

A key element of DEFC is the association of taint meta-data with individual events. Events consist of multiple event *parts*. Each part contains a *name*, stored *data* and a DEFC *security label* to control access to data. Event parts enable multiple collaborating units to process events while, due to different labels per part, each unit can only access parts of the data stored in an event. This means that units can only read from and write to those parts of events that they have privileges for and cannot affect the data flow for other units that may occur via the same events. Dispatching events as single, connected entities while restricting access to parts using labels supports the principle of least privilege (§2.3.3).

An alternative to using a single event with multiple, independently labeled parts would be to send multiple events, each carrying only the data appropriate for a given recipient. This would lead to a larger number of dispatched events and the relationship between these events would be lost.

Figure 3.3 illustrates an event in the trading scenario from Section 3.1. The event consists of three parts: type, body and trader_id. It contains a bid order issued by Trader 1. Different requirements exist for confidentiality, and the event consists of three parts. The first part, type, is not confidential (R1). The second, body, should only be accessible by the Broker unit (R2). The final part, trader_id, contains information that should be disclosed to the Broker unit but the Broker unit may not further communicate the information that it contains (R3).

3.2.2 Security labels

DEFC tracks data flow and enforces data flow policy via the use of *security labels* (or simply *labels*). Labels are the input to the taint tracking system's checking operations and protect the confidentiality and integrity of attached events. They are similar to labels in Flume [KYB+07]. For example, in the scenario from Section 3.1, labels can be used to enforce requirement R3: when the Broker learns the id of a trader from the trader_id part of an bid order, it may only communicate with the trader that issued the order.

Figure 3.3 also illustrates the labels of the bid order event. Security labels are pairs of a *confidentiality component* S and an *integrity component* I. The notation (S, I) is used to denote these two components when referring to a label. S and I are each *sets of tags*. A tag represents an individual and indivisible concern over data. If a tag is placed in S, the concern involves data confidentiality; if it is placed in I it involves data integrity. Tags are opaque to units and are implemented as unique, random bit-strings. This chapter refers to them using symbolic names, giving hints whether they are used for confidentiality or integrity security concerns, e.g. i-trader-1 is an integrity tag used by Trader 1.

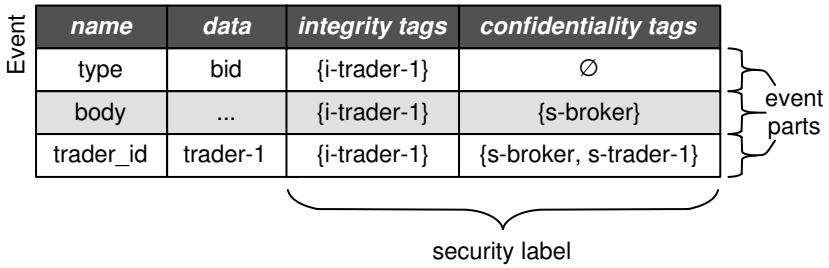

Figure 3.3: A DEFCon event with multiple named parts. Different confidentiality and integrity tags protect data in each part.

Tags that are added to the confidentiality component of an event label are "sticky", i.e. when a unit receives the event and processes it, the tags in the confidentiality component of the event propagate to the labels of subsequent events that it emits. Adding a tag to the confidentiality component of an event label restricts access to the event for other units. All units must propagate this tag to their output (unless a unit exercises privileges over the tag). The opposite happens with integrity tags—they are "fragile". Once a unit mixes data from events that contain an integrity tag with data from events that do not, any events that it emits will not contain the integrity tag (unless, again, the unit exercises privileges).

Assume that the Broker unit in the trading scenario from Section 3.1 receives data from an event with the confidentiality component {s-broker, s-trader-1} in its label. If a second event arrives labeled with {s-broker, s-trader-2}, the label of any subsequently emitted events will include all tags (i.e. s-broker, s-trader-1 and s-trader-2) in its confidentiality component. This correctly captures the confidentiality of output data due to both data sources. Similarly, assume that the Broker unit emits stock tick events with {i-broker} in their labels' integrity components. If a unit combines such events with data from events labeled {i-trader-1}, the result data should no longer be associated directly with the Broker unit. Thus, any events emitted will have empty integrity {}.

Overall, the requirements for the trading scenario from Section 3.1 that were described in Section 3.2.1 may be enforced with the following labels (illustrated in Figure 3.3):

- The type part of the event is unprotected (R1).
- The body part is protected with the tag s-broker to enforce that the event is matched by the Broker unit (R2).
- The trader_id part is further protected with a tag unique to the sender in order to prevent the Broker from disclosing the order's issuer (R3).

Labels are ordered and form a lattice [Den76]. For their confidentiality component S, ordering is achieved by the subset relation: information labeled with S_a can flow to units labeled with S_b if and only if $S_a \subseteq S_b$. For their integrity component I, ordering is achieved with the superset relation: information labeled with I_a can flow to units labeled with I_b if and only if $I_a \supseteq I_b$. Thus, the "can-flow to" relation between labels L_a and L_b, illustrated with

the symbol $\prec$, is defined as:

$$L_a \prec L_b \quad \text{iff} \quad S_a \subseteq S_b \wedge I_a \supseteq I_b$$

where

$$L_a = (S_a, I_a) \quad \text{and} \quad L_b = (S_b, I_b)$$

The "can-flow-to" relation orders labels according to how "restrictive" they are. If $L_a \prec L_b$, L_a is less restrictive than L_b, and every unit that may receive an event protected with L_b should be able to receive an event protected with L_a.

3.2.3 Tag and delegation privileges

Event processing units are stateful, i.e. they maintain their state in between receiving two events. Instead of associating labels with different data that a unit stores, a single label (S_u, I_u) represents the overall confidentiality and integrity of the data stored in the unit. The value of this label at a given point in time specifies a unit's current *contamination level*. DEFC favours unit-level over variable-level tracking granularity, which would introduce performance penalty when executing unit code and may require extensive modifications to the language interpreter (§2.3.3).

Units need the ability to add tags to and remove tags from their label. Since these operations change the events that units have access to, units can only perform them if they possess the necessary *privileges*. There are two types of runtime *tag* privileges, represented with the sets O_u^+ and O_u^-. If a unit has a tag t in the O_u^+ set, it can add t to either the confidentiality (S_u) or the integrity component (I_u) of its label. If t appears in O_u^- , the unit can remove t from S_u or I_u.

The implications of adding or removing a tag depend on whether a tag is used for confidentiality or integrity. When a unit adds the tag $t \in O_u^+$ to S_u, the unit is able to receive confidential data stored in event parts that are protected with t. This is effectively a transition to a higher level of secrecy, and the unit exercises a *clearance* privilege (i.e. clearance *to high secrecy*). When a unit adds $t \in O_u^+$ to I_u instead, any event parts that it subsequently outputs will contain t in their integrity compartment. This action *endorses* the unit's current state to be of a higher level of integrity—the unit exercises an *endorsement* privilege. Similarly, if a unit removes $t \in O_u^-$ from S_u, its state is declassified, and it can now emit events at a lower level of secrecy. This is an example of a unit exercising a *declassification* privilege. Finally, if a unit removes $t \in O_u^-$ from I_u, the unit is able to receive event parts without t at their integrity component. This is effectively a transition to a lower level of integrity, and the unit exercises a *clearance* privilege to achieve it (i.e. clearance *to low integrity*).

DEFC supports dynamic privilege management, i.e. units can delegate to other units some of the tag privileges that they posses. The ability to delegate a tag privilege is itself a privilege known as a *delegation privilege*. Delegation privileges can be used to enforce specific

event processing topologies—this is in contrast to Asbestos, HiStar and Flume (§2.3.3). For example, assume that in an expanded trading scenario from Section 3.1, a new Regulator unit must inspect every event that Trader 1 sends to the Broker. This can be achieved by delegating the appropriate tag privileges, which enforce the particular communication pattern between Regulator, Trader 1 and Broker while withholding the relevant delegation privileges. This scenario is described in detail later in Section 3.5.

Delegation is subject to another set of privileges: $O_u^{-\text{auth}}$ and $O_u^{+\text{auth}}$. The semantics of these delegation privileges are explained using a short-hand notation. For tag t and unit u:

- t_u^+ is short for $t \in O_u^+$ (tag privilege);
- t_u^- is short for $t \in O_u^-$ (tag privilege);
- $t_u^{+\text{auth}}$ is short for $t \in O_u^{+\text{auth}}$ (delegation privilege);
- $t_u^{-\text{auth}}$ is short for $t \in O_u^{-\text{auth}}$ (delegation privilege).

The u subscript is omitted if the context is clear or when these privileges are discussed in abstract terms.

The delegation privilege $t_u^{+\text{auth}}$ allows unit u to delegate to a receiving unit r the tag privilege t^+ and the delegation privilege $t^{+\text{auth}}$ itself. After declassification and according to the actual privilege chosen for delegation (t^+ or t^-), either t_r^+ or $t_r^{+\text{auth}}$ holds. Likewise, delegation privilege $t_u^{-\text{auth}}$ allows the delegation of tag privilege t^- or of delegation privilege $t^{-\text{auth}}$. Privilege delegation occurs via privilege-carrying events (presented in Section 3.2.5), which ensures that the process does not introduce covert channels.

Tags are generated on demand by DEFCon. When a unit u requests a new tag t, it is assigned $t_u^{-\text{auth}}$ and $t_u^{+\text{auth}}$. The unit can use $t_u^{-\text{auth}}$ and $t_u^{+\text{auth}}$ for itself in order to acquire t_u^- and t_u^+, respectively. The tag and all privileges are themselves transferable to other units.

When a unit u combines t_u^- with t_u^+, it has full privilege over t. This effectively enables u to ignore any event flow restrictions with respect to t. Without $t_u^{-\text{auth}}$ and $t_u^{+\text{auth}}$, u is still unable to transfer these privileges.

3.2.4 Input/output labels

Processing units must posses a convenient method to exercise their privileges when receiving and emitting events. DEFC achieves this by exposing to each unit two labels: an *input label* $(S_u^{\text{in}}, I_u^{\text{in}})$ and an *output label* $(S_u^{\text{out}}, I_u^{\text{out}})$. The input label is equivalent to the unit's contamination level (S_u, I_u) and restricts the events that u may receive. The output label restricts the labels of any events u emits. A unit may alter the tags that these two labels contain in order to change the restrictions that the taint tracking system imposes on its input and output. To do so, it needs to possess—and in this way exercise—the corresponding tag

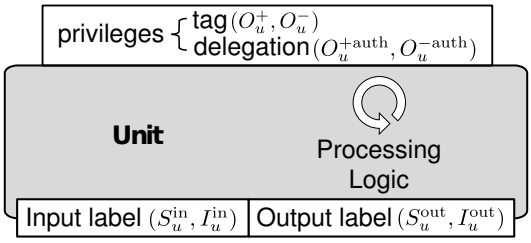

Figure 3.4: Unit labels and privileges in DEFC.

privileges for every tag added (i.e. t^+) or removed (i.e. t^-) to or from its labels. Figure 3.4 summarises all unit labels and privileges in DEFC.

The separation of input/output labels is a convenient mechanism that reduces the need to exercise privileges each time an event is received or emitted. As an example, assume that a Broker unit receives orders from Trader units and must vouch for the integrity of every event that it emits. By adding the tag i-broker to the integrity component of its output label (I^{out}), it vouches for the integrity of every subsequent trade event. This does not require the Broker to exercise privileges explicitly. Similarly, if the Broker unit must learn and communicate the identity of a trader from a bid event (such as in Figure 3.3) without imposing any restrictions on the units that receive it, it can add s-trader-1 to S^{in} but not to S^{out}. Similarly to the integrity scenario, tag privileges are only used once when the change in the input/output labels occurs.

DEFC requires explicit requests for all changes to the input/output labels and thus avoids information leaks due to implicit taint meta-data changes (§2.3.2). This means that, for example, if an event protected with a confidentiality tag t is sent to a unit and the unit has t^+ but $t \notin S^{in}$, the event is not delivered to the unit.

3.2.5 Dynamic privilege propagation

DEFC uses events as an in-band mechanism to propagate privileges between units at runtime. *Privilege-carrying events* in DEFC support an additional field per event part that is opaque to units and stores privileges. When a unit receives an event and requests to read one of its parts, any privileges associated with that part are bestowed upon the unit.

As an example of dynamic privilege delegation, consider the bid order event that must store in a trader_id part the name of the issuing trader (Figure 3.3). Assume that a Regulator unit is the only unit that should be able to receive this event part. Trader units can allocate a different tag t for each bid event that they emit and use it to protect the confidentiality of the trader_id part. To delegate t^+ and t^- to the Regulator, Trader units can add an additional part to the bid event that is visible to the Regulator unit only and stores t^+ and t^-. The Regulator, after reading this additional part, can add t to its input label and learn the trader's identity from the trader_id part.

For a unit to use any of the received privileges, it must have a reference to the particular tag

that these privileges refer to. Since privileges are bestowed implicitly upon reading an event part, units have to communicate references to tags explicitly as part of the data stored in an event part. Units are expected to know by design when they are supposed to delegate or receive privileges and act accordingly. In the previous example, Trader units should store in the additional event part that the Regulator accesses a reference to t as part of the data.

3.2.6 Partial event processing

Some applications in event processing can be seen as processing events along a *main data flow path*. In this processing pattern, an event is sent to units, which in turn attach additional parts and re-emit it. DEFC supports such event processing applications and enables them to operate on *parts* of the events along the data flow path. This is in addition to scenarios, in which units emit new events in response to the events that they receive. Emitting events with additional parts only changes the labels of any parts that were accessed by a unit. When a new event is emitted instead, the label of each event part has to be as restrictive as the unit's current output label.

As an example of *partial* event processing, the Broker unit that receives the bid event in Figure 3.3 can process the order without knowing the identity of the trader that issued it. The Broker unit may receive the body of the order, try to match the offered price with ask events and add a new part with an error message to explain why no match was possible. The Broker is unaware of the trader_id part and thus that part's label should not change.

After an event is received and processed by a unit u, the unit must invoke a `release` API call (see Section 3.4.1) to trigger event delivery to other units. After `release`, the event may be delivered to other units. The label of each event part that u modified—and not any others—must be changed to be as restrictive as the output label of u. The modified event is not delivered again to units that, due to their input label, are only able to receive the event parts that u did not change. Such a behaviour would result in covert channels based on counting the number of times an event is received by a unit. Similarly, if units with different output labels attempt to store to the exact same event part different values, the event stores both values. This precludes covert channels that would otherwise arise by having units at a higher confidentiality level affect the data that units at lower confidentiality can receive.

3.3 Unit isolation in Java

DEFCon, the runtime taint tracking system that uses DEFC for data flow enforcement, should control all channels that allow units to communicate. In every exchange of data between units, DEFCon must invoke checking operations to enforce DEFC constraints. Without the ability to monitor all communication channels between units, a unit with the clearance privilege to receive confidential events but not the privilege to declassify their content (i.e. the unit has t^+ for every tag t in the confidentiality components of accessed

event parts but not t^-) could exploit such unmonitored communication channels to avoid DEFC enforcement. As discussed in Section 2.3.1, units must execute in isolation and should not be able to communicate directly with each other or with applications outside the runtime taint tracking system.

At the same time, it is required that DEFCon offers low latency and high throughput communication between units to support demanding applications such as low latency trading (§3.1.1). One option would be to use process-level isolation as offered by the operating system. As shown in Section 3.5, process-based isolation increases the event processing latency due to the additional cost of inter-process communication (e.g. the serialisation of complex data structures stored in events) and the overhead of context-switching. Therefore processing units in DEFCon execute within the same operating system process, and isolation is achieved by introducing new mechanisms as part of the programming language runtime.

The programming language used in DEFCon is Java. Java is chosen because it offers a mature and efficient platform, static typing, and it is a typical choice for industrial-strength event processing applications. Event processing units are implemented as Java classes and use a specific API for event exchange with other units (see Section 3.4). Units leverage a shared memory address space for efficient event dispatch. DEFCon has access to Java unit bytecode and, when loading unit classes into memory, it enforces that they are restricted to the supported API. DEFCon precludes the use of Java Development Kit (JDK) libraries that are not required in event processing (e.g. UI frameworks or reflection) or offer unmonitored communication with components outside the DEFCon system (e.g. I/O libraries). Enforcing the use of a specific API, however, is not enough to isolate processing units fully, as described below.

Ensuring that two Java objects cannot share information is not trivial because isolation was not a goal when Java was first designed. Even if two Java objects do not have references to each other or to a shared object, there are multiple communication channels that they can use to exchange data. These include *storage* channels, which arise from shared state exposed due to the semantics of Java (e.g. object locks) or due to the design of classes in the JDK (e.g. through mutable static fields), and *timing* channels, which occur from manipulating system resources (e.g. CPU utilisation). DEFCon prevents the use of storage channels because they can be used for reliable, high-bandwidth communication. Monitoring timing channels is left for future work. Timing channels are harder to use because they typically rely on resource monopolisation—such behaviour is likely to be noticed and attributed to the users involved given the assumption of accountable users in the threat model (§3.1.2).

Java exposes shared state to applications in multiple different ways, each time creating additional storage channels. There are three fundamental types of storage channels that units can potentially use:

1. In OpenJDK 6, the JDK libraries contain about 4,000 *static fields*. For example, the JDK uses a static integer field `Thread.threadSeqNum` to identify threads. Two units

may exchange data in an uncontrolled fashion by manipulating this field implicitly, e.g. by launching specific numbers of threads.

2. The JDK libraries contain more than 2,000 *native methods*. These may expose state of the JVM and, in the worst case, this state may be manipulated similarly to static fields. For example, the `String` and `Object` classes both contain native methods that retrieve state from internal data structures of the JVM.

3. The Java language offers *synchronisation* primitives, i.e. `synchronized` blocks and `wait/notify` methods. These enable units to manipulate an object's lock and thus exchange data even if an object does not otherwise contain any fields that can be modified.

As presented next, several past attempts have aimed to offer isolation in Java. Nevertheless, they are not suitable for DEFCon due to two main requirements:

Low maintenance effort. Adding support for isolation to any production JVM should be easy and only require minimal manual effort. Extensive code changes to implement isolation are hard to maintain when the Java platform is updated. Such an approach would be of limited practical applicability.

Efficiency. The isolation methodology should permit efficient communication across isolated components. Isolation should neither significantly increase the latency nor reduce the throughput of message passing between units.

3.3.1 Previous proposals for isolation in Java

Previous efforts to add support for isolation in Java typically require extensive manual effort. Production JDKs have large codebases and are hard to modify without knowledge about their internal architecture. Similarly, it is hard to guarantee the absence of covert communication channels in the JVM. In contrast, research JDKs and JVMs are easier to inspect and modify but are often outdated and seldom match the performance of their production counterparts.

J-Kernel [HCC+98] and Joe-E [MWC10] are two research efforts that provide isolation for Java. They prevent shared state by rejecting user classes that use mutable static fields. For JDK channels, custom proxies are introduced that prevent user classes from accessing shared state, e.g. proxies prevent access to the static fields of the `System` and `File` classes. KaffeOS [BHL00], a research JVM, requires the manual assessment of all static fields in the JDK libraries. JDK classes are rewritten to avoid using static fields, modified to expose different state per isolated component, or "reloaded". With reloading, a different JDK class is exported to each isolated component with unique values for static fields. However, the reloading mechanism suggested cannot be applied when a class is referenced transitively by other shared classes (e.g. as it is the case for the `Object` class). An extensive manual assessment of a large number of classes is still required.

Sun's own MVM [CD01] and later I-JVM [GTM+09] avoid manual assessment by transparently replicating all static fields per isolated component. The JVM is modified to keep track of which isolated component executes and to expose a different set of values for static fields. The authors of MVM additionally assess all native methods to identify cases, in which global state is exposed. The effort required to support newer JVM releases is therefore considerable. MVM itself was only completed for the Solaris/SPARC architecture and is no longer maintained.

MVM uses distinct heap spaces for each isolated component and requires serialisation for communication. Incommunicado [PVCD02] improves upon this design with a more efficient form of deep-copying instead of serialisation. Both designs limit performance because the use of separate heaps nullifies the benefit of a single memory address space.

KaffeOS and I-JVM provide for efficient communication of isolated components by allowing objects to be shared without copying. Yet, their design is not appropriate for DEFCon because, once an object is shared, there is no way to monitor the communication of two isolated components. J-Kernel and JX [GFWK02] expose a proxy to allow access to objects created in other isolated components. While a proxy would enable DEFCon to monitor communication, synchronous method invocation creates timing covert channels that are easy to exploit: isolated components may communicate by modulating the CPU time spent inside method calls.

3.3.2 Isolation methodology in DEFCon

DEFCon isolates units by controlling any shared state that units have access to. Units are only given references to objects that DEFCon controls. This is important because if two units get to share a reference to an object of their choice, they can use it to communicate in an unrestricted fashion. Subsequent analysis also enforces that units cannot take advantage of any storage channels in the platform to communicate.

As a first step to achieve isolation, units communicate with each other only via message passing. When two units exchange data in the form of objects attached to events, DEFCon should provide pass-by-value semantics. The low latency requirement of DEFCon precludes an implementation that involves deep-copying of messages—units should exploit the single memory address space to minimise communication latency. Providing units with references to shared objects is not acceptable either because it would violate isolation. Instead, DEFCon only supports the dispatch of immutable objects inside events. Units copy the data in these events only when needed. The mechanism to achieve efficient dispatch of immutable objects in events is described in Section 3.4.1.

As a second step towards isolation, DEFCon prevents units from accessing covert storage channels by employing a set of techniques to analyse dangerous *targets* in the JDK. As explained before, dangerous targets include static fields, native methods and synchronisation primitives. The next section (§3.3.3) presents how DEFCon prevents access to channels

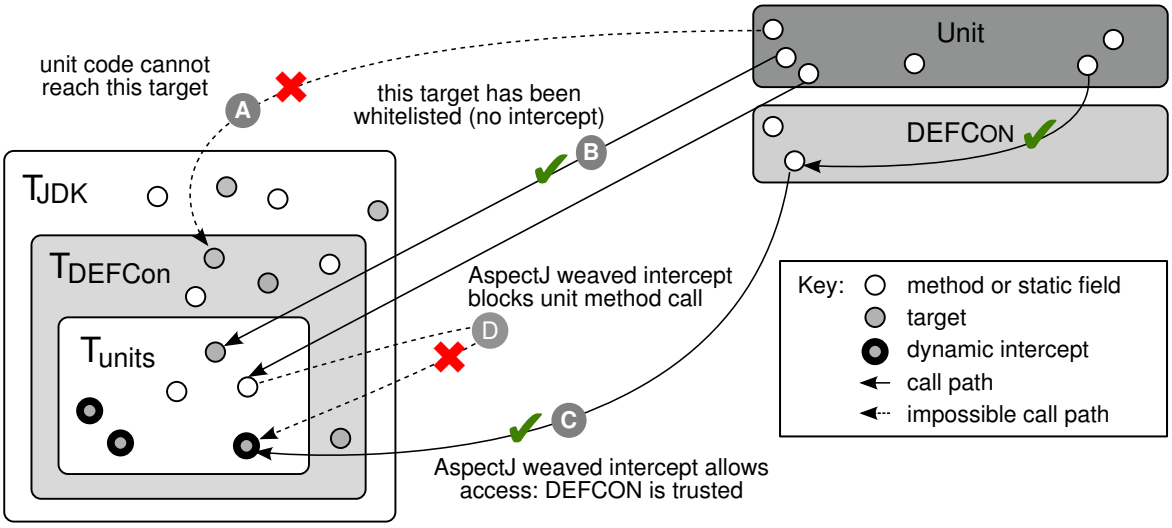

Figure 3.5: Isolation enforcement between units in DEFCon. Isolation relies on a combination of static white-listing (examples A and B) and dynamic intercepts (examples C and D).

created by static fields and static methods, and Section 3.3.4 focuses on channels due to synchronisation primitives.

3.3.3 Restricting channels due to static fields and native methods

Figure 3.5 illustrates a categorisation of dangerous targets in the JDK (i.e. static fields and native methods) according to their reachability from different DEFCon components. Units—given that they cannot import arbitrary Java libraries—can only reach a subset of all dangerous targets which are denoted with T_{units}. The DEFCon implementation uses additional Java libraries, which means that it can reach targets in T_{DEFCon}—a superset of T_{units}. The DEFCon implementation, however, does not itself use all Java libraries; T_{JDK} includes targets reachable neither by DEFCon nor by unit code. The following analysis is based on the Java libraries available to units in the financial scenario presented in Section 3.5.

Static dependency analysis

Dangerous targets that are not reachable from DEFCon or unit code (T_{JDK}) can be removed from the JDK without further impact. To simplify subsequent analysis, any library classes not used by DEFCon or the financial processing units in Section 3.5 are removed from the JDK (e.g. AWT/Swing classes). This preliminary pruning of the JDK results in a subset that contains approximately 20% of the classes in the original JDK. This subset includes more than 2,000 dangerous targets.

Many of the remaining targets (T_{DEFCon}) are only reachable from DEFCon code because event processing units are constrained to classes relevant to event processing. Event processing units typically only load classes from the `java.lang` and `java.util` packages—there is no reason for (non-malicious) units to import classes from packages such as `java.lang.reflect`

or `java.security`. To enforce that units only use a subset of classes available in the JDK, DEFCon provides a custom class loader for units. The custom *unit class loader* inspects the classes that units load and only accepts requests to load classes from a *white-list*. This means that units cannot directly reach some of the dangerous targets in T_{DEFCon}. For example, the call labeled 'A' in Figure 3.5 cannot occur due to the white-list of the unit class loader.

Allowing units to load classes only from a white-list is not sufficient to ensure that units can only reach targets in classes that are in the white-list. When the unit class loader permits a unit to load a JDK class from the white-list, the actual loading of the class is delegated to the "bootstrap" class loader of the JVM. The "bootstrap" class loader identifies any additional JDK classes that the original class references and loads these as well. Since this behaviour cannot be controlled, the unit may transitively reach targets in such additional classes by invoking methods of the class that it was allowed to load.

Reachability analysis

To address this problem, a static reachability analysis is performed on the JDK that, starting from classes in the unit class loader's white-list, transitively calculates the set of targets that a unit may reach, which is T_{units}. The analysis first enumerates method-to-method execution paths possible in the JDK. It then identifies any dangerous targets that each method accesses directly. Finally, it combines the two data sets to calculate T_{units}.

An important challenge for the static reachability analysis is dynamic method dispatch. When a method contains a call to a given method signature, the analysis assumes conservatively that *any* compatible method that has the same signature may be the target of the method invocation at runtime. Method signatures are compatible when (1) the signature is part of a Java interface and multiple JDK classes implement that interface, or (2) when the signature is defined as part of a class and there exist subclasses of that class. Dynamic method dispatch introduces false positives in the static reachability analysis. Despite the fact that only the pruned JDK after the dependency analysis is considered, T_{units} still contains about 1,200 dangerous targets reachable from `java.lang`. The breakdown is approximately 320 native methods and 900 static fields.

Heuristic-based white-listing

Not all of the targets identified in T_{units} after the reachability analysis are in fact dangerous. The number of dangerous targets in T_{units} can be further reduced by a set of heuristics to identify and white-list targets that units cannot exploit as storage channels:

- 66 static fields and 20 native methods declared in the `Unsafe` class cannot be used as storage channels. The class enables applications to access JVM memory and as such, the Java Security Framework monitors attempts to call methods of the `Unsafe` class.

Targets in this class can be considered safe because any unmonitored access to JVM memory while the Java Security Framework is active would be a critical JVM bug.

- Final static fields that store immutable values cannot be used as storage channels. These include final static fields that store strings, primitive wrapper classes (e.g. `Integer` and `Boolean`) and primitives (e.g. `int` and `boolean`). Their values are in effect constants and units cannot change them in order to communicate data.
- Some private static fields, despite not being declared final, cannot be used as storage channels. These are fields that store primitives, immutable values or vectors of constants and are only written once in the class implementation. Such fields are in effect constants.

By applying these heuristics, a number of targets in the JDK are white-listed. After this step, the number of dangerous targets in T_{units} is further reduced to approximately 500 static fields and 300 native methods. Figure 3.5 illustrates a call that reaches a white-listed target in T_{units} marked with 'B'.

Automatic runtime code injection

Additional mechanisms are required to ensure that any remaining dangerous targets that units can access cannot be exploited as storage channels. Other projects for isolation in Java [BHL00, CD01] manually assess all native methods and duplicate static fields per isolated component. DEFCon minimises the number of native JDK methods that need to be checked and avoids direct modifications of JVM code.

DEFCon employs aspect-oriented programming (AOP) [KLM+97] to modify the behaviour of JDK classes indirectly in a programmatic fashion. AOP is used to inject code automatically at specific points during JDK execution. DEFCon uses the MAJOR/FERRARI framework [VBM08] for its ability to modify JDK bytecode (i.e. in addition to application bytecode) using the AspectJ language. To control access to the remaining targets in T_{units} after the static analysis and the heuristic-based white-listing, multiple *pointcuts* are defined. Each pointcut contains code (i.e. an *advice*) that prevents units from potentially exploiting the associated target. Advice code replicates static fields per unit and attaches checks to control how native methods are accessed, as follows:

Static fields. Replicating static fields is straightforward as long as the object stored in the field can be cloned without sharing any references with the original object. If this is possible, and upon unit access to the static field, a deep-copy operation clones the object and returns a unit-specific copy to the caller. The deep-copy operation occurs on read accesses to the static field. When, however, the field stores an immutable value, the deep-copy operation can be delayed further—until a write operation changes the field's value. If copying the value stored in the static field is not possible, advice code generates a security exception and unit execution stops.

Native methods. Native methods in T_{units} should only be invoked from DEFCon code. When such methods are called, advice code inspects the execution stack. If code from the DEFCon prototype has performed any of the method invocations in the stack, the call is permitted because the DEFCon implementation is trusted not to contain storage channels in the API exposed to units (see Section §3.4). In Figure 3.5, such a call is marked with 'C'. If no DEFCon code is found on the stack, advise code raises an exception and stops unit execution (example marked with 'D').

Security exceptions are raised at runtime when an unsafe scenario is encountered. When combined with the reachability analysis in the JDK, they ensure that no units in DEFCon may exploit a storage channel. No modifications to the JVM are required.

Manual white-listing

Security exceptions, however, prevent unit execution. When attempting to deploy the units used in the evaluation scenario from Section 3.5, 15 native methods and 27 static fields triggered exceptions and stopped execution. These targets were manually inspected and since none could be exploited as storage channels, all of them were added to the white-list. Next are some examples of these targets along with a justification of the decision to add them to the white-list.

`java.lang.Object.hashCode`. This is a native method that returns a hash value for each object in the JVM. Since Java applications cannot control this value for a specific object, this function effectively returns a constant.

`java.lang.Object.getClass`. This is a native method that returns a `Class` object used to represent the class of a given object. Class objects are unique and constant thus this method returns the equivalent of a constant static field.

`java.lang.Double.longBitsToDouble`. This is a native method that returns its parameter as a `double`. It does not access any JVM state and therefore it may not be used by units to exchange data.

`java.lang.System.security`. This is a private static field that is exposed to units via the `getSecurityManager` method of the same class. The Java Security Framework prevents units from modifying the current security manager. Additionally, the DEFCon security manager does not contain any mutable fields that units can access. Therefore units may not use this static field—or the object that it contains—to exchange data.

A further consideration with runtime code injection is performance. Each time a pointcut is activated, it triggers the associated advice to decide whether access to the target is safe. To reduce the overhead of the approach in the evaluation scenario (see Section 3.5.1), the execution of units is profiled to identify dangerous targets that are accessed regularly. These

targets are manually inspected and subsequently white-listed, as well. In total, 15 additional targets (6 static fields and 9 native methods) were inspected and added to the white-list for performance reasons.

3.3.4 Restricting synchronisation channels

Java objects may contain mutable state that enables unrestricted communication if two processing units reference the same object. Therefore, two units must never share a reference to an object that contains mutable state. A way for DEFCon to prevent shared references is by deep-copying or serialising any object dispatched between units. Deep-copy and serialisation are effective but result in additional overhead and should be avoided in high-performance event processing applications. DEFCon instead ensures that every object dispatched to units does not contain mutable state and thus references to it cannot be used to exchange data (see Section §3.4.1).

Java objects, however, contain a single piece of mutable information: the synchronisation monitor (also know as the *object lock*). The synchronisation monitor can be manipulated using `synchronized` blocks and the `wait/notify` methods. Two units may use the synchronisation monitor as a covert storage channel to exchange data.

The synchronisation monitor introduces a further channel that is specific to Java and is independent of how DEFCon dispatches references to objects across units. Strings in Java can be "interned", i.e. the JVM reuses instances of the same `String` object when applications reference strings with the same value. The benefit of interning is that strings can be compared with the reference comparison operator (`==`) instead of the much slower `equals` method. The synchronisation monitor of a shared object, however, can be used as a storage channel. `Class` objects (e.g. as returned by `Object.getClass`) are similar to strings, i.e. they are immutable and references to them are reused. Other isolation projects [CD01, GTM+09] suggest to allocate a different copy of `String`/`Class` objects per isolated component. This approach is inadequate for DEFCon because units should exchange events efficiency in the same memory address space without copying them.

Automatic runtime injection

DEFCon follows a different approach that changes how processing units may use the `synchronized` keyword. Instead of enabling units to synchronise on objects of arbitrary Java classes, DEFCon only allows synchronisation on objects that are guaranteed to never be shared across isolated components. To indicate this, a new tagging interface called `NeverShared` is introduced. For a type T to implement `NeverShared`, three prerequisites must hold:

1. DEFCon prevents objects of type T from being put inside events because events are shared between units without copying;

2. no native method of the JDK that is white-listed may return a reference to the same object of type T to different units; and

3. there is no static field of type T in the JDK that is white-listed as safe. Two different units may otherwise access the object stored at the static filed at runtime.

Processing units may be `synchronized` on an object of a type T only if T implements `NeverShared`. Since `String` does not satisfy the first prerequisite and `Class` does not satisfy the second, they do not implement `NeverShared`. Therefore neither `String` nor `Class` objects can be used for synchronisation in unit code.

Units may define their own classes that implement `NeverShared` and use objects of these classes for synchronisation. DEFCon introduces an AOP aspect to identify the type of objects used as operands in `synchronized` blocks. If the type is statically known to implement `NeverShared`, the aspect introduces no runtime overhead. If instead the precise type is unknown and the decision cannot be taken statically, the aspect introduces an advice that checks the type of the operand at runtime. The runtime check raises an exception if (1) the class of the object used in the `synchronized` block does not implement `NeverShared` and (2) unit code is detected on the call stack. This identifies cases, in which units do not perform synchronisation explicitly, yet synchronisation occurs in the implementation of the JDK methods that they invoke. Manual inspection is required for such JDK methods.

Manual inspection

The JDK contains methods that should not be invoked from unit code because they synchronise on object monitors. If the object that they synchronise on is shared across units, the object's monitor may be exploited indirectly as a covert channel, i.e. without an explicit `synchronized` block in the unit code. For example, methods of the `StringBuffer` class and `ClassLoader.loadClass` all synchronise on `StringBuffer` and `ClassLoader` objects, respectively. However, these classes satisfy the three prerequisites of the `NeverShared` interface, and shared object instances across units are not possible. To allow such safe JDK methods to be invoked from unit code, DEFCon uses another aspect to indicate that classes such as `StringBuffer` and `Classloader` implement `NeverShared`. This aspect is invoked before the aspect introduced in the previous paragraph that detects `NeverShared`.

Overall, OpenJDK 6 was secured in four days, and this involved the manual inspection of only 52 targets (15 native methods, 27 static fields and 10 synchronisation targets) and no modifications to the JVM.

3.4 Implementation of DEFCon

We implemented a DEFCon prototype in Java. Apart from supporting DEFC and enforcing unit isolation, as presented in the previous two sections, the prototype is designed to satisfy

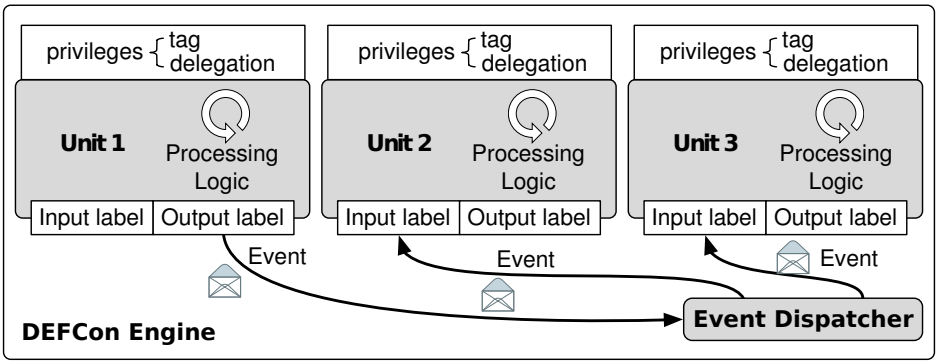

Figure 3.6: Overview of the DEFCon architecture. DEFCon provides for efficient event dispatch while enforcing event flow requirements via label checks.

two goals: *safety*, i.e. the prototype must not introduce additional covert channels that violate unit isolation, and *efficiency* for event dispatch.

Figure 3.6 shows the architecture of the DEFCon prototype. The DEFCon *engine* provides the execution environment for event processing units. Units exchange events by leveraging the facilities provided by the engine. The engine exposes an API for event exchange and, additionally, monitors unit execution in order to enforce data flow according to DEFC. Overall, the tasks that the DEFCon engine performs are as follows:

DEFC management. The DEFC model requires a set of management tasks at runtime. The engine keeps track of allocated tags, associates input and output labels with units and maintains sets of privileges for each unit. The engine does not expose input/output labels to units—units receive opaque tag handles that can only be used to modify labels via the engine API, as described in Section 3.4.1.

Event dispatch. The engine exposes publish/subscribe event communication primitives. Units register their interest for events by issuing subscriptions, which specify predicates on event content. Units receive call-backs in return, which are invoked with references to events when a match occurs. The *event dispatcher* is the engine component that registers all unit subscriptions and matches them against events that are subsequently published, as described in Section 3.4.2.

Unit life-cycle management. The engine is responsible for instantiating and terminating processing units. DEFCon monitors the changes that units perform to their labels to restrict the events that they may subsequently receive or emit. As will be described in Section 3.4.3, units may also request to be activated automatically upon receiving an event and be deactivated after its processing finishes. In such cases, the engine actively instantiates and terminates units.

3.4.1 API for event generation and dispatch

The DEFCon engine API, which allows units to publish events and control their propagation, is presented in Table 3.1. The API consists of three different method groups. First, methods starting from `createEvent` and `addPart` down to `publish` and `relase` enable units to create new events, manipulate their content and sent them to other units in the same engine. The second group of `subscribe`, `subscribeManaged` and `getEvent` methods is how units express their interest in specific events and how they access event content. Finally, `intantiateUnit`, `changeInOutLabel` and `changeOutLabel` enable units to process events with more restrictive labels either by instantiating new units or by changing their own labels. The semantics of the API methods are presented in detail in Table 3.1. The following paragraphs expand on two properties of the DEFCon API, i.e. contamination independence and the use of freezable objects to avoid serialisation.

Contamination independence

A unit's contamination level imposes restrictions on the API methods that the unit invokes. For example, `addPart`, a method used by units to add a new part to an event, requires the unit to specify the label for the part added. Assuming a unit u with $S_u^{\text{out}} = \{d\}$, u should not be able to output an event with $S = \{t\}$. If u attempts such a call to `addPart`, the engine should reject the call and notify the unit. This design, however, implies that each unit operates at a specific initial contamination level. If a unit is instantiated at a different contamination level that its developers did not anticipate, unit execution may fail at runtime.

Contamination independence is an important property of the DEFCon API, which allows a unit to be instantiated at an arbitrary contamination level and continue to operate as expected while its initial contamination level is reflected in the labels of the events that it receives and emits. Contamination independence enables a unit a to instantiate a unit b and sandbox it, both controlling which events b receives and the propagation of any events that b emits. To achieve this, the DEFCon API adheres to a design that requires methods with labels as parameters to not impose restrictions on the calling unit, i.e. every call succeeds independently of the unit's current contamination level. DEFCon ensures that any tags in the unit's current output label are used to modify the label specified in the API call transparently. In the previous example, u's call to `addPart` with $S = \{t\}$ results in an event part labeled $S' = \{d, t\}$ instead of the call being rejected. In general, DEFCon may modify the S and I label components provided by a unit u in API calls (Table 3.1) and use S' and I' instead as follows: $S' = S \cup S_u^{\text{out}}$ and $I' = I_u^{\text{out}} - I \cap I_u^{\text{out}}$.

Freezing shared objects

DEFCon API methods allow units that execute in the same memory address space to exchange data via events efficiently. As explained in Section 3.3.4, DEFCon must ensure that only immutable objects are exchanged via events. When the type of the `data` parameter

DEFCon API method	**Description**
createEvent() $\rightarrow e$	Creates and returns a handle to a new event e.
addPart(e, S, I, *name*, *data*)	Creates a new part *name* that stores *data* and attaches it to event e. The label of the new part is (S, I).
delPart(e, S, I, *name*)	Deletes from event e any part *name*. The label of the parts that are deleted is (S, I).
readPart(e, *name*) $\rightarrow$ *(S, I, data)**	Retrieves the data from part *name* attached to event e. If the event contains more than one part with the same name, all data are returned. $S_p \subseteq S_u^{in}$ and $I_p \supseteq I_u^{in}$ must hold for every such event part.
attachPrivilegeToPart(e, *name*, S, I, t, p)	Attaches the privilege p over the tag t to an event part. The event part is referenced using its *name* and label (S, I). To succeed, the call requires that t_u^{pauth} holds. The result is a privilege-carrying event (§3.2.5).
cloneEvent(e, S, I) $\rightarrow e'$	Creates e', i.e. a copy of the event e. Every label in the resulting event is updated to reflect the contamination level of the unit that performs the operation. Therefore, (1) every tag in S is attached to the confidentiality compartment of every part's label in e; and (2) only tags in I are maintained in the integrity compartment of each part's label. The updated labels of e' ensure that e' may only be dispatched to units that can receive u's output.
publish(e)	Publishes a new event e that was created by the same unit. The call rejects events without parts because all events should contain labels that control their propagation.
release(e)	Releases an event e that was dispatched to the unit as a result of a subscription (§3.2.6).
subscribe(*filter*) $\rightarrow s$	Subscribes to events by providing a *filter* for content-based matching (§3.4.2). The filter is a set of predicates on the data stored in event parts. Successful matching occurs when the event data match the filter and $S_p \subseteq S_u^{in}$, $I_p \supseteq I_u^{in}$ holds for each accessed part. The method returns a subscription s.
subscribeManaged(*handler*, *filter*) $\rightarrow s$	Subscribes to events with a *filter* and a subscription *handler* for processing matched events. The method returns a managed subscription s (§3.4.3). An event is delivered to the handler when $S_p \subseteq S_u^{in,max}$ and $I_p \supseteq I_u^{in,min}$ hold for each part referenced by the filter. ($S_u^{in,max}$, $I_u^{in,min}$) is the most permissive input label possible, i.e. given all clearance privileges that the unit currently holds or may be delegated via the triggering event.
getEvent() $\rightarrow (e, s)$	Blocks the calling unit until an event e matches one of its subscriptions
instantiateUnit(u', S, I, $O_{u'}^{p}$, $O_{u'}^{\text{pauth}}$)	Instantiates a new unit u' at a given contamination level (S, I) and delegates to it the $O_{u'}^{p}$, $O_{u'}^{\text{pauth}}$ privileges. The call requires $O_{u'}^{p} \subseteq O_u^{pauth}$ and $O_{u'}^{pauth} \subseteq O_u^{pauth}$. Additionally, $S \subseteq O_u^{+auth}$ and $I \subseteq O_u^{-auth}$ must hold.
changeInOutLabel($\langle$S|I$\rangle$, $\langle$add|del$\rangle$, t)	Adds/removes a tag t to/from a unit's input *and* output label. The method requires that either t_u^+ or t_u^- holds.
changeOutLabel($\langle$S|I$\rangle$, $\langle$add|del$\rangle$, t)	Adds/removes a tag t *only* to/from a unit's output label. The method requires that either t_u^+ or t_u^- holds.

Table 3.1: An overview of the DEFCon API available to event processing units.

in the `addPart` method is of an immutable Java class such as `String`, the resulting event part and its content can be safely released to another unit. Limiting events to immutable JDK classes is unacceptable though. Units that wish to exchange objects of class `Date` or of collection classes such as `ArrayList<Date>` would be forced to encode and decode mutable objects into immutable counterparts. This is equivalent to serialisation and eliminates the benefits of the shared memory address space.

DEFCon avoids the cost of serialisation for mutable classes (either performed explicitly by units or implicitly during event dispatch) by limiting event content to a set of `Freezable` classes. `Freezable` is a package-private base class defined by DEFCon that contains a single method `freeze`. DEFCon calls `freeze` when it dispatches events. The method implementation ensures that a `Freezable` object can no longer change after it was "frozen". The approach avoids serialisation at the cost of inspecting an `isFrozen` flag before modifying the object during any mutating operation.

DEFCon extends `Freezable` to implement counterparts of many JDK collection classes, e.g. `ArrayList` and `HashMap`. Objects of `Freezable` collection classes must efficiently freeze their content when the container gets frozen. An iterative `freeze` implementation that visits each object stored in a collection is unacceptable because it introduces additional overheard during event dispatch. Instead, when objects of `Freezable` classes are inserted to a `Freezable` collection, they store a reference to the collection's `isFrozen` flag. This effectively makes `freeze` execute in constant time. The downside is that mutating operations on a `Freezable` object inspect the `isFrozen` flag of every collection that the object is part of.

A similar approach is used to avoid repeatedly updating the labels provided by units in each API call. Contamination independence requires that any unit-provided label parameter is modified before it is used in an event to reflect the calling unit's current contamination level. This introduces overhead at runtime for calls that require labels as parameters. To avoid this overhead, the `Label` type in DEFCon supports a `freeze` method. Units must call this method before a label is used in an API call. `Label.freeze` prevents future edits to the label and, at the same time, it modifies the label to capture the unit's current output label. Frozen labels can directly propagate to event parts without modification. A unit is able to use a particular frozen label as long as it does not apply any of its privileges—these change the unit's current output label and thus affect the labels of any subsequently emitted events.

3.4.2 Event and label matching

DEFCon supports unit communication via content-based publish/subscribe [EFGK03]. Publish/subscribe decouples the sender from the receiver, i.e. events are dispatched according to their content and not to specific recipients. Moreover, a successful `publish` call does not convey information about event delivery to the caller, thus publish/subscribe permits unidirectional communication.

DEFCon uses a modified version of a well-known content-based matching algorithm by Fab-

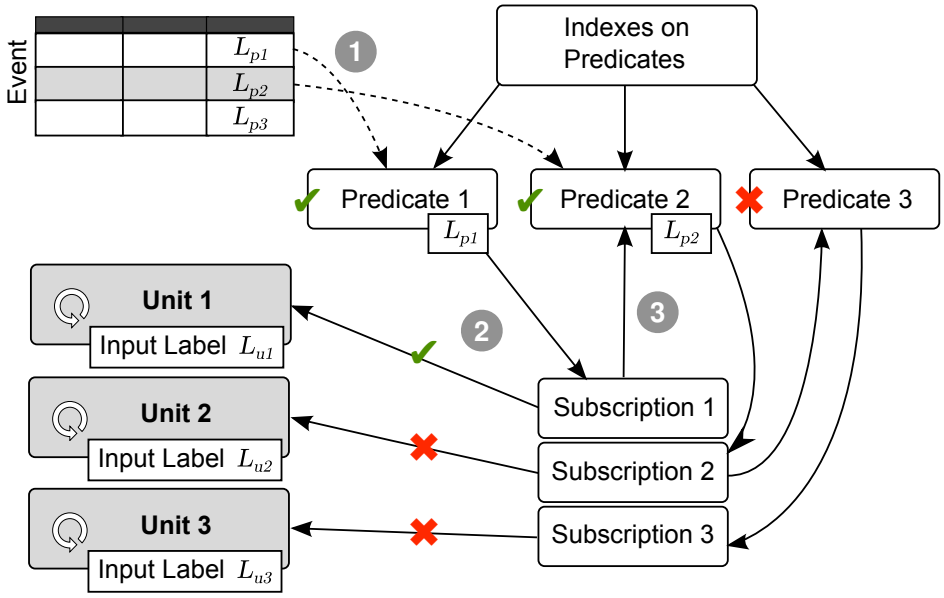

Figure 3.7: Label-aware publish/subscribe matching in DEFCon's Event Dispatcher.

ret et al. [FJL+01]. In addition to matching event content against subscriptions, the modified algorithm checks the "can-flow-to" relation between event and unit labels. Figure 3.7 shows the data structures inside the DEFCon event dispatcher while it matches an event against the subscriptions of three potential recipients.

In step 1, the event dispatcher iterates over the event parts and checks them for matches against a set of subscription predicates. In this step, the event dispatcher uses only one of each subscription's predicates and indexes them for fast retrieval. This predicate is known as the subscription's *access predicate* and, to improve performance, it must be the subscription's most selective predicate. When a predicate matches an event part (e.g. predicate 1 and predicate 2), the part's label is stored temporarily with the predicate.

In step 2, the event dispatcher fetches the corresponding subscriptions for all access predicates that matched in step 1 (i.e. subscriptions 1 and 2). Each subscription contains a reference to the unit that issued it and, as a result, to the unit's current input label. For the subscription to be considered in step 3, step 2 verifies that the label of the access predicate from step 1 and the input label of the respective unit satisfy the "can flow to" relation, e.g. $L_{p1} \prec L_{u1}$ for subscription 1.

In step 3, the event dispatcher iterates over the result subscriptions with matched access predicates from step 2. Each subscription contains references to any remaining predicates that must also match for this subscription to receive the event. For example, subscription 1 requires that predicates 2 also matches while subscription 2 requires a match for predicate 3. For each such subscription, in step 3 the event dispatcher verifies that all additional predicates matched the event in step 1. In addition, it ensures that the label stored with the predicate satisfies the "can flow to" relation when compared to the input label of the unit that issued the subscription. In the example of Figure 3.7, the event matches subscription 1 if $L_{p2} \prec L_{u1}$

and does not match subscription 2 because predicate 3 did not match in step 1.

If S is the number of subscriptions and P is the average number of predicates in each, the worst case complexity of the algorithm is $O(SP)$. In practice however each event contains few parts, and these parts only match a small subset of the access predicates registered by active subscriptions. This means that only few subscriptions match after step 1 and need to be checked in detail later (steps 2 and 3). As indicated by the performance evaluation of the DEFCon prototype in Section 3.5, the algorithm can support a large number of subscriptions registered at the same time with high event rates.

3.4.3 Managing unit life-cycle

DEFCon supports two different modes of execution for units: *unmanaged* and *managed*. The managed mode of execution enables a single unit to process multiple events and avoid cross-contamination (§2.3.3). Cross-contamination occurs when a unit processes events protected with different tags without having the privilege to remove these tags from its output label. Such tags accumulate in all subsequent events that the unit publishes and significantly restrict which units may receive them.

In unmanaged mode, which is the default mode of operation for units in DEFCon, units control their subscriptions and may stop execution at any point. A unit can initiate an arbitrary number of threads and use them to publish events or continue processing between event deliveries. Each unit issues subscriptions with the `subscribe` API method and gets notified of matching events (see Table 3.1). Units are responsible for setting up their labels via the `changeOutLabel` and `changeInOutLabel` methods.

Units may enter the managed mode of execution by issuing a *managed subscription* with the `subscribeManaged` API method. In managed mode, the engine invokes a unit-supplied subscription handler each time that there is an event that matches the subscription. The subscription handler processes the event, may publish additional events in response, and its execution stops with a call to `release` the incoming event that triggered the execution.

Managed subscriptions provide an abstraction similar to event processes in Asbestos (§2.3.3). In a managed subscription, the subscription handler is unable to maintain state between the delivery of events that are labeled differently. The engine generates new instances of the subscription handler each time a new event is delivered to a managed subscription. Thus, only the labels of the *current* event should propagate to any subsequent events published as a response—not the labels of events that the handler has previously received.

To enable different instances of the event handler whose labels permit communication to exchange data, DEFCon exposes a simple key-vale store that inspects the current contamination level of the subscription handler before servicing requests.

3.5 Evaluation

The goal of the experimental evaluation is to demonstrate the effectiveness of DEFCon in supporting high-performance event processing applications while enforcing event flow security requirements. This is achieved by implementing and evaluating a simple financial stock trading platform in DEFCon. The DEFCon implementation is compared to a similar implementation in *Marketcetera* [Asa09]. Marketcetera is an open-source trading platform written in Java and is one of the few available offerings in a market dominated by proprietary solutions. It is gaining momentum by matching the performance offered by proprietary systems [Sho09]. It also supports various features such as complex event processing, rapid prototyping of trading strategies and modules to interact with different stock exchanges.

The overhead of event flow enforcement, i.e. enforcing requirements expressed with the DEFC model, is measured in terms of reduced event processing performance and increased memory consumption. Processing performance is quantified through the two metrics from Section 3.1.1: event throughput and processing latency.

The performance results show that DEFCon, despite enforcing DEFC, performs better than Marketcetera. Marketcetera isolates the trading strategies of different traders using separate JVMs. Instead, DEFCon enforces isolation as described in Section 3.3. This allows significantly more trading strategies to be hosted on a single machine along with the stock market feed. Overall, without the increased memory consumption from multiple JVMs, DEFCon scales to 10 times the number of trading strategies while exhibiting low processing latency and high throughput. This makes DEFCon suitable for co-location (§3.1) as more traders may use a single machine for secure event processing.

3.5.1 Financial trading scenario

DEFCon and Marketcetera are evaluated using pairs trading (§3.1). We implemented pairs trading in both systems. This section describes the pairs trading implementation in Marketcetera and DEFCon.

Marketcetera implementation

In Marketcetera, the pairs trading strategy is implemented inside a Strategy Agent. Strategy Agents are components that encapsulate the trading strategies of different users and provide the facilities to interact with the rest of the platform. To isolate the processing inside different Strategy Agents, Marketcetera instantiates a separate JVM for each Strategy Agent. Marketcetera does not impose restrictions on event processing via the Strategy Agent. Therefore, third-party trading strategies may leak any information that they receive and traders must vouch for the correctness of every trading strategy. Brokering in Marketcetera is performed via the Order Routing Service (ORS), which can be reached from the trading strategies via the Strategy Agent. The ORS usually only routes orders to an external exchange for matching.

To emulate a co-location scenario, the ORS is extended to match orders locally and generate a resulting stock tick feed, which is sent to Strategy Agents. Our implementation is based on Marketcetera version 1.5.0.

DEFCon implementation

In DEFCon, trading strategies of competing traders execute in the same JVM for low communication latency. Traders express confidentiality requirements using the DEFC model and DEFCon enforces these without fear of disclosure. Figure 3.8 illustrates the trading platform as implemented in DEFCon. It introduces the following processing units:

Trader units encapsulate trading strategies. Each trader may provide multiple such units. In the given scenario, Trader units buy and sell stocks using pairs trading.

Pair Monitor units monitor the prices of two stocks and notify other units when an investment threshold is reached. They offer a basic monitoring service to Trader units to facilitate the rapid development of trading strategies development such as pairs trading.

A **Broker unit** matches bid/ask orders from Trader units. When a match can occur locally, no external stock exchange is involved.

A **Stock Exchange unit** communicates orders that cannot be matched locally to a remote stock exchange. It is also the source of the stock tick feed for trades that occur there.

A **Regulator unit** samples and inspects a subset of local trades for compliance with financial regulations. For example, it may verify that the volume of traded stock in any given transaction does not exceed a quota.

DEFCon operation

Figure 3.8 illustrates the nine steps that describe the operation of the DEFCon trading platform:

Step 1: A Trader unit initiates the monitoring of two stocks by publishing a Monitor event. As the trader's selection of stocks must be kept private independently of the processing in the Pairs Monitor, Trader 1 allocates a unique tag t_1 and uses it to protect the event. The corresponding Pairs Monitor unit is delegated the t_1^+ privilege and can read the event content to learn which stock prices to monitor. Since the Pairs Monitor unit does not possess t_1^-, DEFCon ensures that any subsequent events that it publishes include t_1 in their parts' confidentiality compartments.

Step 2: The Pair Monitor unit issues two subscriptions for Tick events that correspond to the two stocks referenced in the Monitor event. All Pair Monitor units are instantiated with the tag s in their input integrity level and therefore are confined to receive only events that

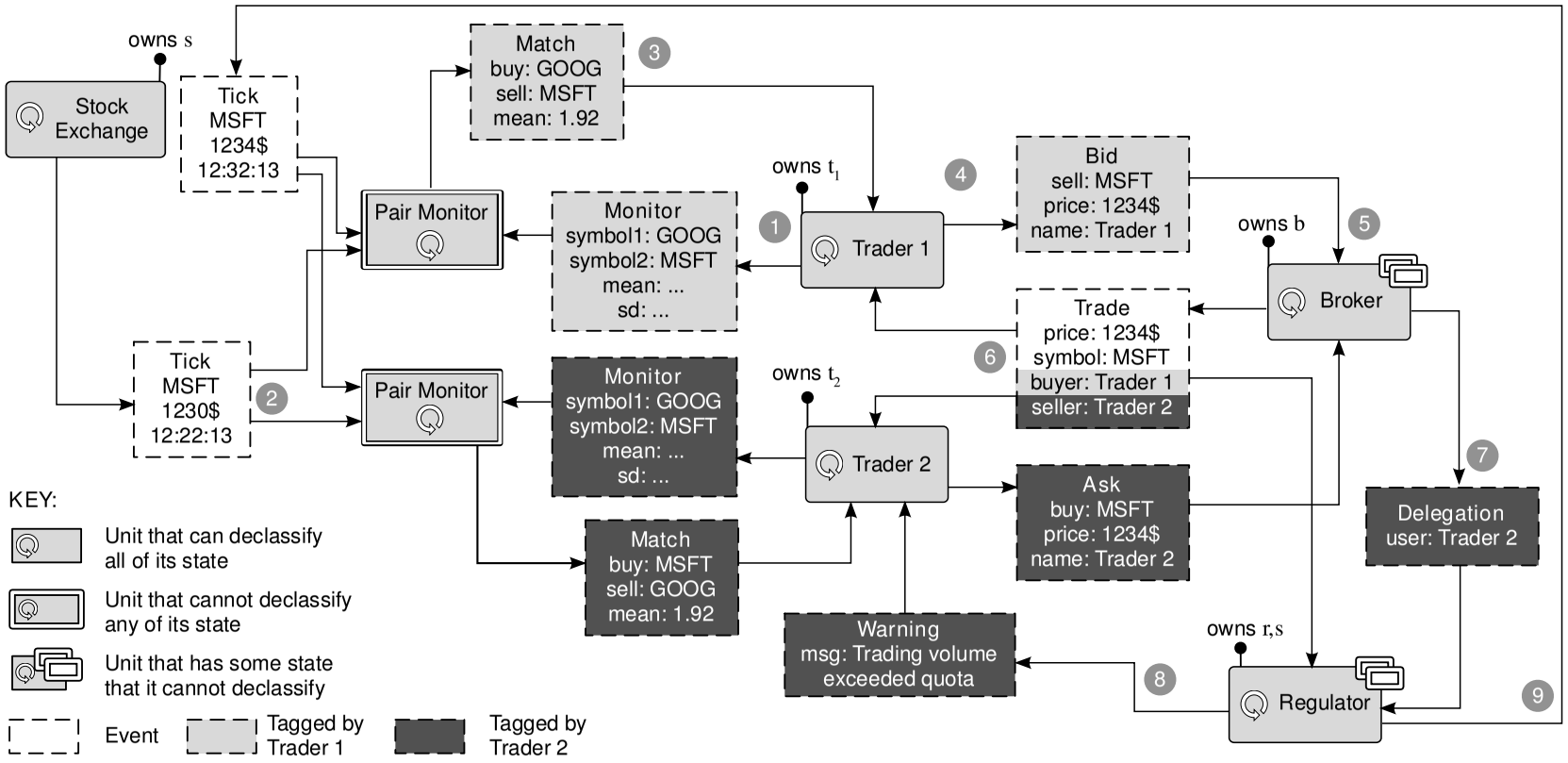

Figure 3.8: A stock trading platform in DEFCon. The figure shows the units involved in trading and the events that they exchange, highlighting the use of DEFC for controlling event flow.

contain s. Given that only the Stock Exchange unit owns s, no other unit may influence the monitoring of stock prices that the Pair Monitor performs, e.g. by publishing bogus events.

Step 3: Once the triggering price ratio according to the pairs trading algorithm occurs, the Pairs Monitor unit publishes a Match event to indicate this observation. Since the Match event is guaranteed to be protected with t_1, only Trader 1 has t_1^+, which is required to receive the event. Additionally, the Pairs Monitor attaches the tag s to the Match event's integrity label, thus proving to Trader 1 that the stock price observation is based on authoritative data.

Step 4: Given the Match event from the Pairs Monitor and trader-specific logic, Trader 1 may decide to issue an order to sell stock. It publishes a Bid event with the details of the order and the credentials of the trader involved. The order should be matched using the local brokering facilities offered by the Broker unit. However, Trader 1 has three confidentiality requirements for each published order: (1) the order must convey enough information to the Broker to allow matching with other orders; (2) only the Broker should be able to identify the trader that issued the order; and (3) no other unit should be able to correlate two different orders of Trader 1. Units correlating events with their publishers may infer and replicate the trading logic despite lacking knowledge of the publisher's identity. The Broker is also not fully trusted by Trader 1—the trader name in an order should never be disclosed to another unit by the Broker. These requirements are satisfied by using different confidentiality tags in the Bid event. The first part, symbol/price, is protected with tag b while the trader name part is protected with an additional unique tag t_r that used only for this event. The Broker is the only unit with b^+/b^-, so it is free to match orders and disclose the results. However, Trader 1 delegates to the Broker only t_r^+ via the first Bid part. The Broker may then learn the trader name but is forced to propagate t_r to any subsequent events that it publishes.

Step 5: The Broker receives the Bid event, reads the first part and waits for a matching ask order to complete the trade. The Broker uses a managed subscription (§3.4.3) to learn trader identities and thus avoids cross-contamination from the unique tags used in different Bid/Ask events.

Step 6: When two orders match, the Broker publishes a Trade event to notify the units involved in the transaction. The first part of the event (symbol/price) contains public information and is not protected. The two parts that contain the identities of the trading parties (buyer/seller) are protected with the respective unique tag that each Trader unit used in Step 4. Trader 1 may thus learn about its successful trade while DEFCon ensures that any other unit in the system is confined to public information.

Step 7+8: A Regulator unit inspects the Trade events published to verify them for legal compliance. It samples some of the events and inspects them in detail. To gain access to event parts that are protected by unique tags, the Regulator requests the respective t_r^+ privilege on-demand from the Broker (Step 7). For the Broker to be able to delegate t_r^+, t_r^{+auth} should have been included in the order in Step 4. The Regulator uses a managed subscription

to discover the trader names and avoid cross-contamination. Any regulating violations are reported to the Trader units that are referenced in the Trader event (Step 8). These Warning events are protected by the respective tag used for the trader identity in Step 4.

Step 9: The Regulator unit republishes as Tick events the public information contained in Trade events that comply with regulations. To enable Pair Monitor units to receive events from the Regulator, the Regulator owns tag s and uses in the integrity compartment of its output label.

3.5.2 Experimental results

The experimental evaluation of both DEFCon and Marketcetera is based on a synthetic workload of stock ticks derived from trading traces made in the London Stock Exchange. The prices in the original traces were modified in order to trigger periodically the pairs trading algorithm and circumvent the problem of choosing correlated pairs of stocks from real market data. To generate significant order load in both systems, the workload triggered the pairs trading algorithm once every 10 stock ticks on average.

The evaluation demonstrates the performance of DEFCon and Marketcetera when varying the number of active trading strategies. In both platforms, each trading strategy monitors a pair of stocks. The selection of stocks is made according to a Zipf distribution that provides results skewed towards specific pairs. This models the fact that certain stock pairs are known to be highly correlated, and therefore they are more likely to be selected by traders that engage in pairs trading. The experiments were run on a dual processor Intel Xeon E5540 2.53 GHz machine with 1 GiB of maximum heap memory allocated to each JVM, which was Sun's Hostspot JVM version 1.6.0_16.

DEFCon performance

To evaluate the event throughput performance of DEFCon, the Stock Exchange unit continuously replays the stock tick traces at the maximum achievable rate. The rate is measured every 100 ms. The Stock Exchange unit implementation is single-threaded to follow a design with a single point of entry for incoming market data.

Figure 3.9 shows the median event throughput when increasing the number of trading strategies (each strategy is implemented as a separate Trader unit). In the simplest configuration without any security enforcement (*no security*), the event throughput ranges from 220,000 events per second with 200 Trader units to 75,000 events with 2,000 Trader units. Next, labels are introduced that control event propagation. To evaluate the benefit of freezing objects instead of copying them (§3.4.1), the experiment is repeated while defensively copying shared data in API calls (*labels+clone*) or freezing them(*labels+freeze*). The *labels+clone* scenario results in a significant reduction of event throughout (around 30%) despite most units only including simple data structures in events. The overhead in *labels+freeze* is neg-

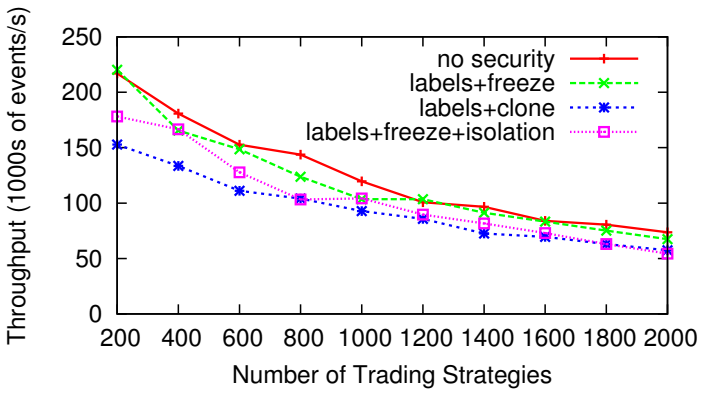

Figure 3.9: Maximum supported event throughput in DEFCon while increasing the number of trading strategies.

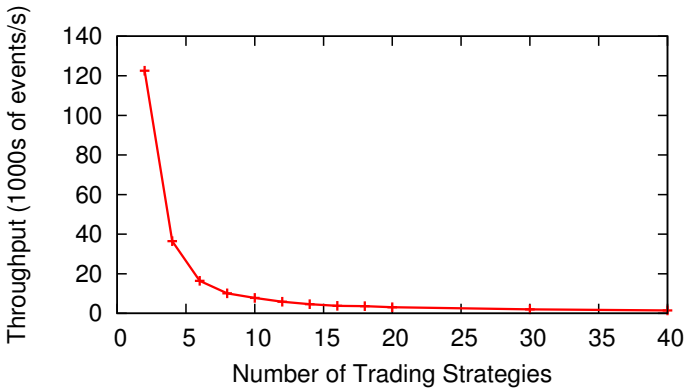

Figure 3.10: Maximum supported event throughput in Marketcetera while increasing the number of trading strategies.

ligible with throughput being close to the *no security* scenario. Adding the runtime checks to enforce isolation (§3.3) leads to an overall reduction in throughput close to 20% (*labels+freeze+isolation*) when compared to the *no security* case, which is largely independent of the number of Trader units.

Event processing latency is measured end-to-end as the timespan between the publication of a `Tick` event by the Stock Exchange unit and the publication of the resulting Trade event by the Broker. It includes the event processing time spent inside the Pair Monitor, Trader and Broker units as well as the event dispatch time between them.

Figure 3.11 shows the 70^{th} percentile of observed latencies to finalise a trade while increasing the number of active trading strategies. We ignore higher latency percentiles because they are affected by transient order queuing and by activations of the Java garbage collector. When a popular pair of stocks triggers pairs trading, the increased load in the Broker introduces transient queuing in the system, which affects the latency of order processing. Such spikes are common in trading, e.g. when markets open. The Java garbage collector is also invoked periodically and causes the preemption of processing threads. Its operation typically lasts for 20 ms and increases the latency of individual events.

Figure 3.11 shows that the baseline for event processing latency, i.e. without security enforce-

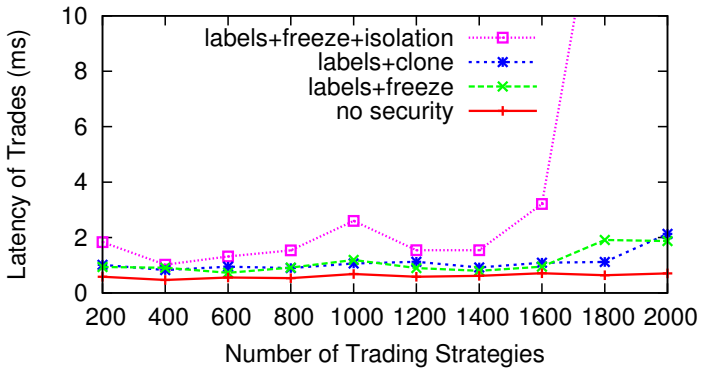

Figure 3.11: Event processing latency in DEFCon while increasing the number of trading strategies.

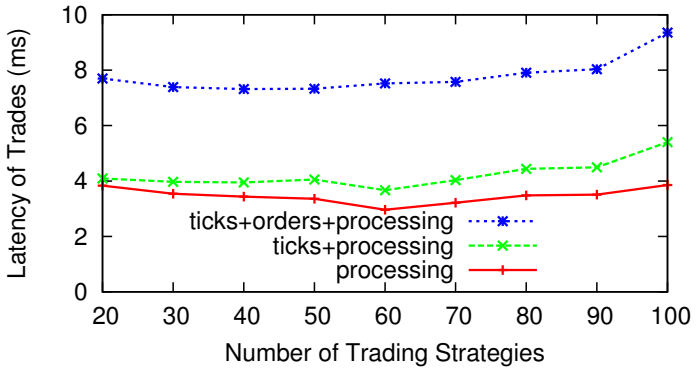

Figure 3.12: Breakdown of event processing latency in Marketcetera to its individual contributions while increasing the number of trading strategies.

ment (*no security*), is 0.5 ms, independently of the number of Trader units. Label checks that clone or freeze event parts and labels increase latency to approximately 1 ms. Isolation introduces an additional 1 ms of latency resulting in an overall latency close to 2 ms, independently of the number of Trader units. The behaviour of the system changes with more than 1500 active Trading units because it becomes overloaded.

The memory consumption while conducting these experiments is shown in Figure 3.13. Across all scenarios, about 300 MiB is the size of the stock tick traces that the Stock Exchange unit caches to maximise throughput. As expected, both the *labels+freeze* and *labels+clone* configurations require little additional memory because they do not introduce significant new data structures in memory. Instead, the AOP framework used to enforce isolation requires an additional 50 MiB for 200 Trader units and increases linearly with the number of Trader units. With more than 1500 Trader units and close to overload, the additional memory required peaks at 200 MiB.

Marketcetera performance

The results from DEFCon are compared with the performance of Marketcetera. Figure 3.10 shows the median throughput in Marketcetera. The event rate is comparable to DEFCon

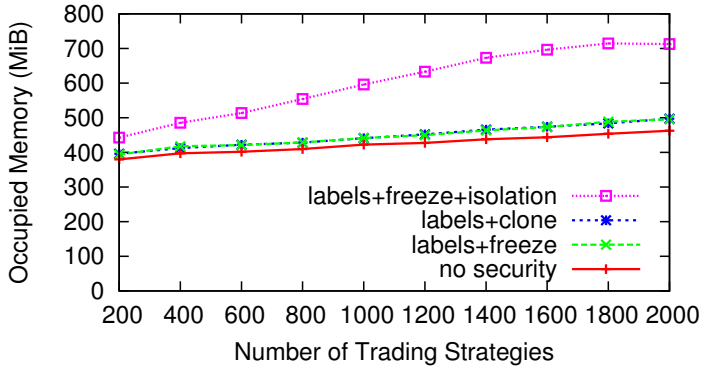

Figure 3.13: Amount of used memory in DEFCon, as a function of the number of traders.

when only two trading strategies are active but scalability suffers after that. With 10 trading strategies, throughput falls below 10,000 events per second. The design issue that causes this behaviour is the lack of a centralised event dispatcher in the Marketcetera architecture similar to the one in DEFCon (§3.4.2). Therefore each Strategy Agent has to filter market data individually.

Latency measurements in Marketcetera refer to the same timespan as DEFCon, i.e. from publishing a tick event to generating the corresponding trade event. To avoid scheduling phenomena that may affect our results, latency measurements are performed using a low stock tick event rate of 1,000 events per second. Figure 3.12 shows the 70^{th} percentile of latencies in Marketcetera when increasing the number of active trading strategies. Compared to the equivalent results for DEFCon on the same page, Marketcetera offers increased latency and does not scale as well. The latency in Marketcetera is close to 8 ms (*ticks+orders+processing*). To better understand this effect, the figure breaks it down to its individual contributions: filtering of market data and processing events according to the pairs trading algorithm in Strategy Agents (*processing*), propagating stock tick events from the ORS to individual Strategy Agents for processing (*ticks+processing*) and propagating orders when the trading algorithm is triggered (*ticks+orders+processing*). With 100 trading strategies and an increasing cost of dispatching events across JVMs, the latency due to event propagation surpasses the latency due to the actual processing performed. In contrast, DEFCon takes advantage of the shared memory address space and centralised event dispatch, staying close to 1 ms for significantly more trading strategies.

The occupied memory is also higher in Marketcetera due to the additional memory needed to instantiate multiple JVMs and filter market data separately in each JVM. For 20 trading strategies, the Marketcetera deployment requires 2 GiB of memory and reaches 6 GiB with 100 strategies. DEFCon instead supports 1,500 trading strategies with less than 1 GiB of memory.

Security comparison

DEFCon and Marketcetera have different security features. DEFCon offers pairs trading as a service to traders. DEFC guarantees that information about the stocks selected for monitoring cannot leak to any other units in the platform despite implementation errors or malicious disclosure attempts. Instead, each Strategy Agent in Marketcetera can communicate with arbitrary recipients—bugs in the pairs trading implementation may therefore disclose a trader's choices. Offering trading strategies as third-party components is not supported in Marketcetera. Similarly, bugs in the implementation of the ORS in Marketcetera may disclose the identities of the traders involved in a transaction. In contrast, Trader units in DEFCon use unique tags to protect their identities and ensure that the Broker forgoes any state that contains the trader id. The Broker is forced to comply and "forget" the trader id or it is unable to contact other units due to cross-contamination. Finally, implementing a regulatory service in Marketcetera is a challenging task that requires the integration with multiple components of the platform. Only Marketcetera's original developers may reliably implement it. In DEFCon, the Regulator is implemented without any additional changes to the platform and only requires the delegation of the privileges necessary for its operation.

3.5.3 Discussion

The experimental evaluation demonstrates that DEFCon's design has the potential to improve the performance and the security properties of existing event processing systems. However, the comparative evaluation of DEFCon and Marketcetera has some limitations.

The most significant limitation is that we lack the expertise required to build a Marketcetera trading platform that is equivalent to our DEFCon implementation. In both systems, the implementation of trading strategies, the brokering algorithm and the market data filtering logic is either identical or closely resembles each other. Marketcetera, however, may support additional features that are orthogonal to the event dispatching functionality that we evaluate. For example, Marketcetera may offer ordering guarantees while dispatching events that DEFCon's publish/subscribe communication does not provide. Such features can slow the Marketcetera implementation down, yet they may also be useful for DEFCon. If such features are implemented in DEFCon, they will reduce the performance gap of the two systems. In addition, our integration of local order brokering into Marketcetera may be lacking compared to a similar implementation by Marketcetera's original authors.

A second limitation is the lack of representative and comprehensive benchmarks to test both systems across a variety of scenarios. Our evaluation scenario shows that DEFCon has a significant advantage when high-throughput event streams are filtered by many event processing units. It does not, however, cover other use cases, in which the workload due to the task that the event processing system performs is significantly different.

3.6 Summary

High-performance event processing applications, for example, as found in algorithmic stock trading, need end-to-end data flow enforcement that does not degrade performance. In this chapter, we presented DEFCon: an event processing and runtime taint tracking system that enforces Decentralised Event Flow Control (DEFC).

DEFC meets the particular security needs of event processing by providing data flow guarantees for event data in the presence of processing bugs and intentional leaks. DEFC is a taint tracking policy that associates taint meta-data, i.e. labels, with event parts. Units receive labeled events and may process a subset of their parts without tainting all the data that the events contain. DEFC separates tag privileges from delegation privileges and enforces arbitrary event processing topologies.

DEFCon relies on isolation at the programming language level. We described a practical methodology for achieving isolation in Java with low manual effort. By isolating processing units running in the same memory address space, we tried to strike a balance between the need for isolation and efficient inter-isolate communication. The isolation methodology first prevents units from using certain Java features and, second, statically identifies code paths in the JDK that may be used by units to exchange data. Such code paths are intercepted and checked at runtime using aspect-oriented programming techniques.

The evaluation of DEFCon highlights the practicality and efficiency of the approach when compared to an open-source trading platform. DEFCon scales to significantly more trading strategies hosted on a single machine and co-located with the stock exchange. This enables more traders to react to market changes with low latency.

Overall, DEFCon demonstrates runtime taint tracking as a practical and efficient solution for supporting demanding applications with minimal performance penalty. DEFCon allows applications in event processing to introduce data flow requirements and subsequently controls data flow accordingly. Isolation of trading strategies is achieved in the same memory address space without changes to the language runtime that are hard to maintain in practice.

In the next chapter, we will present a policy language for specifying data flow policy in DEFCon. High-level data flow policy specification simplifies the interaction of developers with the system. We will also show how data flow policy can be enforced across multiple DEFCon engines.

Chapter 4

Event Flow Security Policy

In this chapter, we describe our solution for expressing and enforcing data flow policy in distributed event processing applications. When distribution is considered, data flow policy may involve multiple organisational domains, which have different requirements about data flow in their systems. We argue that data flow policy should be expressed by policy administrators in the different organisational domains using a high-level policy language. By separating data flow policy from the units that constitute the event processing application, policy administrators can focus on the high-level requirements without being overwhelmed by implementation details. The high-level data flow policy of the application may then be translated to DEFC labels, tags and unit privileges for enforcement.

This chapter presents the above ideas in the context of DEFCon, focusing on enforcement of multi-domain event flow security policy. The chapter starts by introducing the need for a policy language in distributed and multi-domain DEFCon applications and the requirements for such a policy language. Section 4.2 presents the policy language using examples of the different event flow constraints that it supports. Section 4.3 covers the additional components in the DEFCon architecture for policy management and translation to DEFC. Finally, we evaluate our approach for event flow policy with the example from healthcare (§2.1.2) in Section 4.4. The chapter finishes with a summary in Section 4.5.

4.1 Requirements

Without additional support from the event processing system, unit developers are responsible for the specification and initialisation of event flow security policy. As an example, consider the pairs trading scenario from Section 3.5. Financial event processing units are expected to allocate tags correctly, identify the units that should process published events and delegate privileges to enable event processing. Event flow policy that specifies how events are labeled is assumed to be controlled by a single trusted administrator. This is acceptable when event flow policy only involves units deployed on a single DEFCon engine. Instead, if units execute in different DEFCon engines that are under the control of different organisational domains,

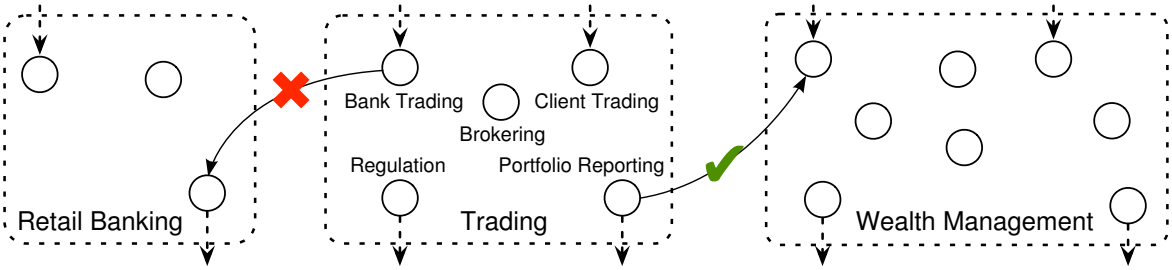

Figure 4.1: Specifying event flow policy within a bank. The figure illustrates DPL flow categories that define boundaries around data and event flow constraints that specify which units can pass data over those boundaries (see Section 4.2).

no single trusted administrator may exist. In such a *multi-domain* scenario, the assumption that event flow policy is specified using low-level tags is error-prone and, as discussed below, inadequate for enforcing event flow.

Figure 4.1 illustrates a multi-domain event processing scenario from banking. The event processing infrastructure of a bank hosts event processing applications on behalf of multiple departments: trading units identify investment opportunities and perform order brokering, retail banking units oversee credit card transactions and private wealth management units collect and report events of interest to the bank's private clients. Events are exchanged between units within the same department and across units that belong to different departments.

Each department in this bank has different event flow policies that the event processing system must enforce. For example, the trading department requires that data stored in events published there should never reach units involved in retail banking unless disclosed by units in Regulation or in Portfolio Reporting. The wealth management department accepts events from the trading department but requires that specific units receive them first. Specifying event flow policy using DEFC tags and privileges directly in such a multi-domain event processing scenario has a number of limitations:

*L*1**: Low-level policy specification.** Event flow policy specification using DEFC tags and privileges is error-prone. Unit developers must translate high-level event flow policy such as "units in Portfolio Reporting cannot send events to the group of units of the retail banking department" to tags and privileges. Event flow policy enforcement may be jeopardised by errors in the translation process, by bugs in the code that allocates tags and delegates privileges at runtime, or by unit vulnerabilities that enable other units to request and obtain privileges that they should not possess.

*L*2**: Tight coupling of units with event flow policy.** Event flow policy specification using DEFC tags and privileges makes changing policy difficult. Changes to policy require units to allocate different tags or alter the distribution of privileges between them. This can only occur via modifications to unit source code. Units involved in the same policy specification are therefore coupled by the policy that they implement. Changing the policy requires, in the worst case, modifications to every unit referenced by it.

*L*3**: No shared semantics of tags across engines.** Event flow policy specification using DEFC tags and privileges requires the translation of tags between DEFCon engines. The reason for this is that tags are opaque random bit-strings (§3.2.2) specific to the engine in which they are allocated. As such, tags cannot be used in a different DEFCon engine. A potential solution is for engines to enforce that local tags are globally unique, e.g. by communicating tag allocations or devising a structured naming scheme for tags. Any solution will require additional coordination in distributed event processing applications and must not reduce event processing performance within each engine.

*L*4**: No support for event flow enforcement across engines.** Event flow policy specification using DEFC tags and privileges cannot enforce event flow across engines without delegating privileges. Assume, for example, that units in Brokering should not directly send events to units of the retail banking department. If all units execute in a single DEFCon engine, this requires labeling all events emitted by units of the trading department with a single confidentiality tag t and delegating to units in Brokering only t^+. This policy cannot be enforced when the units are deployed in different engines. Assume that units of the retail banking department are deployed in engine A and the units of the trading department are deployed in engine B. The units in Brokering, however, are deployed in engine A instead of B. Notice that units in Brokering must still communicate with the other units of the trading department in engine B. To allow external communication, the engine A requires that units in Brokering have t^- (i.e. declassification privilege). However, given t^- for a confidentiality tag, a unit can ignore t and send events to arbitrary destinations.

*L*5**: Inconsistent understanding of policy between domains.** Event flow policy specification using DEFC tags and privileges does not protect against inconsistent policy implementation between domains. In the example of Figure 4.1, units in Portfolio Reporting are allowed to send events to units of the wealth management department. Assume, however, that the two departments use different DEFCon engines and that the trading department requires that only specific units of the wealth management department are able to publicly release event content. Units of the trading department cannot ensure that units of the wealth management department enforce this event flow policy. The reason is that the trading department is unable to control event processing in a different DEFCon engine and correctness depends on the event flow policy of the wealth management department.

Based on the above limitations (*L*1–*L*5), when specifying event flow policy in distributed and multi-domain event processing applications, we devise the following requirements:

- Event flow security policy should be separate from the label-based enforcement mechanism inside DEFCon (see *L*1).

- Event flow security policy should be decoupled from the implementation of event processing units (see $L2$).
- Event flow security policy should provide guarantees for confidentiality and integrity of event flow in distributed and multi-domain event processing applications end-to-end, i.e. across DEFCon engines that belong to different organisational domains (see $L3$, $L4$).
- Event flow security policy should support specification by multiple collaborating domains while the result is amenable to checks for inconsistencies (see $L5$).
- Event flow security policy should map to DEFC efficiently, without resulting in an unacceptable degradation of event processing performance.

4.2 The DEFCon Policy Language

The DEFCon *Policy Language* (DPL) [MPE+10b] is a language for event flow policy specification in DEFCon applications. This section presents DPL using the banking scenario from the previous section. The events in that scenario can be divided into *event flow categories*. An event flow category in DPL identifies a set of events with separate data flow requirements. Examples of potential event flow categories are (1) all the events that units of the trading department publish (see Figure 4.1); (2) the events that units in Portfolio Reporting publish that must be sent to the wealth management department; and (3) the events that units in Brokering publish. Such events must propagate amongst a particular set of units and must not be corrupted or leaked by software faults or malicious behaviour of any unit involved in their processing. Event flow categories are marked in Figure 4.1 with dashed lines.

4.2.1 Event flow constraints

DPL defines data flow policy as *event flow constraints* on event flow categories. Event flow constraints restrict how units communicate the data that they receive from events in the flow category (event flow confidentiality) and which data may be used when generating events in the flow category (event flow integrity). They have the following syntax:

flow_constraint ::= ⟨*flow_name*⟩ ':' '{' flow_part (',' flow_part)* '}' '.'
flow_part ::= ['->'] ⟨*processing_context*⟩ ['->']

As an example, consider a constraint to describe how events should flow between units in Figure 4.1. For the trading department, all units in Bank Trading should only generate events for other units in the same department and should not send data directly to the retail banking department. Units in Regulation instead are trusted to output events outside the trading department. The following DPL example specifies a flow constraint for events in the trading department:

```
1 trading_flow: {
2     −> bank_trading,
3     −> client_trading,
4         brokering,
5         regulation −>,
6         portfolio_reporting −>
7 }
```

Flow constraints start with a *name* of an event flow category (e.g. `trading_flow`, line 1) and then list *flow parts* (lines 2–6), stating whether each part can emit events inside the flow or receive events published outside the flow. Each flow part represents a *processing context*, i.e. a set of units. A processing context can be as specific as a single processing unit that receives and emits events of the particular flow or may enclose many processing units[1]. A unit that is included in a flow constraint without any additional annotation is able to receive events from and publish events in the flow. However, it is *sand-boxed* and may not receive or publish events outside the event flow. In this particular example, the `brokering` unit may receive but not disclose events from the `trading_flow`. Therefore, it resembles the operation of the DEFCon Broker unit in Section 3.5.

Flow parts in DPL constraints may be annotated using the `->` operator. If `->` is used as a prefix operator (lines 2 and 3), the annotated unit is an *input unit* for that event flow. Input units, such as `bank_trading` and `client_trading` for `trading_flow`, are restricted to publish events in the flow but there are no restrictions for their input. They can receive events from arbitrary sources and publish them "inside" the flow. If `->` is used as a postfix operator instead (lines 5 and 6), the annotated unit is an *output unit* for that event flow. Output units, such as `regulation` and `portfolio_reporting` in `trading_flow`, are restricted to receive events from within the flow only, and they are able to publish events without restrictions. If `->` is used as both a prefix and postfix annotation, the corresponding unit is free to receive and publish events ignoring the flow constraint. Input and output units therefore have a degree of freedom when receiving and publishing events, respectively; input units may choose to receive events either "inside" or "outside" the event flow and output units may similarly publish events "inside" or "outside" the flow.

A single event flow constraint restricts event processing to guarantee the confidentiality and the integrity of events in a flow. For example, the `trading_flow` constraint specifies that only the `bank_trading` and `client_trading` units may mark events as having `trading_flow` integrity. These two units protect other units in the flow from receiving bogus events. Similarly, only the `regulation` and `portfolio_reporting` units are privileged to communicate freely events published by units in `trading_flow`. Again, these two units prevent information disclosure that may potentially occur via other units in the `trading_flow` constraint.

[1]To simplify the discussion, a processing context is assumed to be equivalent to a single unit; this assumption is relaxed in Section 4.2.5, in which the general case is addressed. The rest of the section switches to lower-case names for units to highlight that, in reality, the discussion refers to processing contexts.

Sand-boxed units have their input and output restricted by the flow constraint and, due to the enforcement of DPL policies with DEFC (presented later in Section 4.3), they cannot receive or emit events in violation of the constraint. As a consequence, the unit code that has to be trusted to maintain the integrity of the event flow is limited to input units and, for the confidentiality of the event flow, to output units.

There are two ways that policy specification can distinguish flows of information by applying flow constraints: *vertical* and *horizontal* flow separation. Vertical flow separation is applied in flow constraints that require end-to-end restrictions on event processing from input to output. Horizontal flow separation is used to separate individual stages of event processing from each other, effectively guaranteeing that transformations on events are applied in the correct order.

4.2.2 Vertical flow separation

Each flow constraint in DPL describes the propagation of events with different security requirements. The policy administrator may provide arbitrary event flow constraints that reference non-overlapping sets of units. Such flow constraints separate event processing vertically, i.e. each constraint is enforced independently of other constraints. If, however, two flow constraints reference overlapping set of units, the constraints—when enforced collectively—may cut off units from input events or prevent units from publishing events. Overall, vertical flow separation should never prevent the event flow to reach all units referenced by the constraints.

As an example, consider the units referenced by the `trading_flow` constraint from the previous section. Assume that the `regulation` unit should only monitor the trading activity of the bank and not its clients, while the `portfolio_reporting` unit should only report to units of the wealth management department the results of client trading activity. These event flow requirements may be specified in DPL using two constraints as follows:

```
1 bank_trading_flow: {
2   -> bank_trading,
3   -> client_trading,
4       brokering,
5       regulation ->
6 }
```

```
1 client_trading_flow: {
2   -> bank_trading,
3   -> client_trading,
4       brokering,
5       portfolio_reporting ->
6 }
```

The intersection of two flow constraints may introduce conflicting flow requirements for events received or published. For example, the `bank_trading_flow` and `client_trading_flow` constraints both reference three units: `bank_trading`, `client_trading` and `brokering` (Figure 4.2(a)). Some events that `brokering` publishes should be part of one flow (e.g. events that mark successful trades of the bank published to `regulation`) while others should be part of both flows (e.g. events exchanged between `brokering`, `bank_trading` and `client_trading` units that issue, handle and respond to orders). Events published by units at the intersection

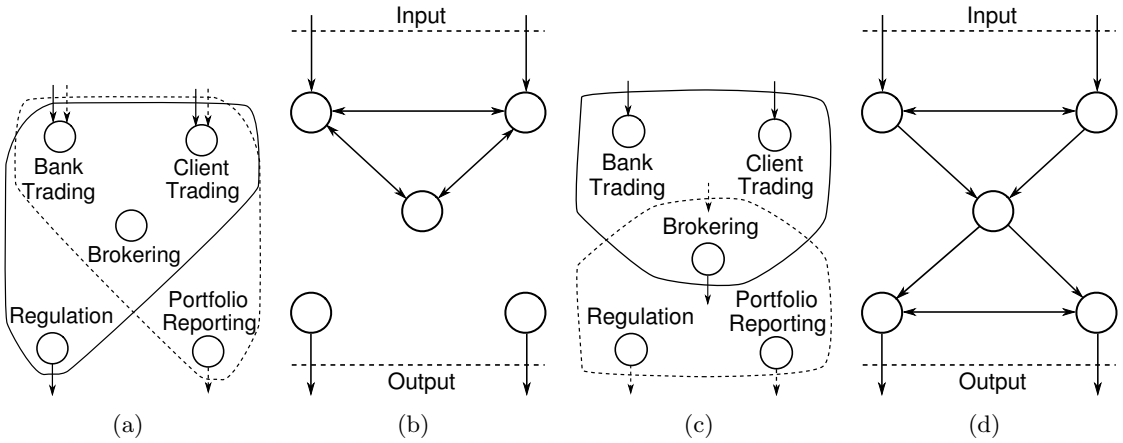

Figure 4.2: Vertical (a) and horizontal (c) flow separation in DPL. Each type of flow separation permits different communication flows, presented in (b) and (d) respectively.

of the two event flows (e.g. by `brokering`) may be received by other units at the intersection or by units in one of the two flows that are also input units for the other flow. This, however, is not the case for either the `regulation` or the `portfolio_reporting` units. Therefore these units cannot receive events from the other three units. They are, in effect, cut off from input. Figure 4.2(b) illustrates the possible interactions between units from Figure 4.2(a) and highlights this problem. DEFCon checks DPL policies for such consistency errors before activation and alerts the administrator (see Section 4.3.1).

There are two ways to address the above issue and allow events to reach `regulation` and `portfolio_reporting` units. The first is to split every shared unit into two units, one for each flow constraint. This acknowledges the fact that the taint tracking system cannot track separate event flows *within a single unit*. Therefore splitting a unit enables DEFCon to track event flow in each and thus enforce constraints between different units. The second solution is to relax the two flow constraints to allow a communication pattern that was not possible before, for example, to enable `brokering` to act an input and output unit for the `bank_trading_flow` and `client_trading_flow` constraints. The downside of this is that the `brokering` unit will have to be trusted to categorise correctly the events that it publishes in the two different flows that it is part of.

4.2.3 Horizontal flow separation

A second use of flow constraints is to control event propagation between categories. The goal is to "chain" different constraints together and control how events propagate from one set of units to the next. When flow constraints separate event processing in stages, horizontal flow separation occurs.

With horizontal flow separation, a policy administrator enforces a specific ordering in event processing. In contrast to vertical flow separation that separates the processing of events with different security requirements, horizontal flow separation ensures that only particular

units may communicate data from the events in one flow constraint to the events in another constraint. In addition, units that are assigned to this role are forced not to disclose data from the events that they process.

Consider the `trading_flow` constraint from Section 4.2.1. An alternative policy would require that all trades are audited for regulatory or reporting purposes. This can be expressed with the horizontal separation of `trading_flow` into two new flow constraints as follows:

```
1 brokering_flow: {
2   −> bank_trading,
3   −> client_trading,
4       brokering −>
5 }
```

```
1 auditing_flow: {
2   −> brokering,
3       regulation −>,
4       portfolio_reporting−>
5 }
```

As seen in Figure 4.2(c), the two flow constrains share references to a single unit, i.e. `brokering`. In contrast to vertical flow separation in Figure 4.2(a), these flow constraints neither cut off a unit from input events nor preclude a unit from outputting events (Figure 4.2(d)). Instead, the intersection of the two flows benefits event flow enforcement. The shared unit, `brokering`, may only receive events from the `bank_trading` or `client_trading` units but, as an output unit for `brokering_flow`, it is able to communicate the events that it outputs freely. The `auditing_flow`, however, references `brokering` as an input unit thus `brokering` may only output events that are covered by the `auditing_flow` constraint. Overall, data propagate from events in the `brokering_flow` to events in the `auditing_flow`, event processing occurs in two consecutive steps and units that are not referenced by the two constraints may not affect the process or identify the data it involves.

4.2.4 Parameterisation

Some event flow policies require that multiple flow constraints with the same structure are separated vertically, e.g. to isolate the processing of different clients. DPL provides a simple method to define multiple such constraints using *parameters*. For example, the `client_trading_flow` constraint may be parameterised with the parameter `client_name` to ensure that all units involved in order processing for a given client are not used for trading on behalf of other clients:

```
1 client_trading_flow[client_name]: {
2   −> bank_trading[client_name],
3   −> client_trading[client_name],
4       brokering[client_name],
5       portfolio_reporting[client_name] −>
6 }
```

Parameters may appear both in flow category names and next to processing units. When one or more parameters appear next to the flow category name, for each value of these parameters, a different flow constraint is generated. If instead the flow category name is

not parameterised (while at least one of the units in the constraint is), a single constraint is assumed that references each unit independently of the parameter's value. For example, a single flow constraint that forces all trading activity to reach `portfolio_reporting` units via `brokering` is:

```
1 reporting_flow: {
2   -> brokering[client_name],
3       portfolio_reporting[client_name] ->
4 }
```

As with horizontal and vertical flow separation, the communication patterns that arise due to parameterised constraints must not cut off units from input or prevent output.

4.2.5 Event processing contexts

Up to this point, all DPL examples are presented assuming a deployment scenario in a single domain. A single policy administrator who understands the requirements of every unit specifies the event flow policy. In such scenarios, the policy administrator can link the policy constraints to the units and events that they refer to. This assumption, however, is unrealistic for multi-domain deployments for two reasons: (1) it places a significant burden on the policy administrator who may not be in a position to known and understand in detail the event flow policy of multiple domains; and (2) when the event processing application involves units administered by different domains, no single domain has control over the details of event processing performed in other domains.

Key to specifying event flow policy for event processing applications that span multiple organisational domains is the abstraction of the relationship between flow constraints and units. DPL achieves this with the concept of event processing contexts. Event processing contexts are hierarchical names that capture the organisational structure of domains involved in event processing. The hierarchical structure of contexts facilitates policy specification for multi-domain applications: domains provide policy constraints at a level that administrators are familiar with and then rely on a federated naming service (analogous to the domain name system (DNS) [Moc87]) for policy specification in other domains (see Section 4.3.1).

Processing context names provide a common, consistent and structured naming scheme to correlate the policies provided by different organisations with units that execute on multiple DEFCon engines. Flow constraints reference contexts—not units—and contexts are mapped to units at a later stage. This relaxes the assumption made in Section 4.2.1 and allows for multiple units referenced in a flow constraint via a single processing context.

By referencing a context, a flow constraint restricts event processing not only for units that are directly part of this context but also for units that are part of any of its sub-contexts.

Processing contexts are illustrated with two examples. The first example further refines the event flow inside the `portfolio_reporting` context (Figure 4.1) with an additional constraint:

```
1 anon_portfolio_reporting: {
2   −> portfolio_reporting.anonymiser,
3       portfolio_reporting.stats −>
4 }
```

The `anon_portfolio_reporting` constraint introduces `anonymiser` and `stats` as two sub-contexts of `portfolio_reporting`. It ensures that all events are anonymised before they are used for calculating statistics in reports. Every unit that is instantiated inside these two sub-contexts is restricted not only from the `anon_portfolio_reporting` constraint but also from other policy flow constraints that reference `portfolio_reporting` (e.g. the `trading_flow` constraint).

As a second multi-domain example, consider an alternative version of the `trading_flow` constraint defined as follows:

```
1 policy uk.co.bank
2 trading_flow: {
3   −> bank_trading,
4   −> .uk.co.hedge−fund.trading,
5       brokering,
6       .uk.co.fsa.regulation.bank −>,
7       portfolio_reporting −>
8 }
```

Identifiers that start with a dot (e.g. `.uk.co.hedge-fund.trading`) mark fully-qualified context names. These may refer to arbitrary contexts. Unqualified context names, such as `bank_trading`, are appended to the context prefix that is specified in each policy's header (e.g. `uk.co.bank`).

The example constraint introduces two new contexts: a hedge fund that the bank trades with (i.e. `uk.co.hedge-fund`) and the UK's Financial Services Authority (i.e. `uk.co.fsa`). Each organisation authorises the bank to introduce DPL flow constraints that restrict event processing in its contexts (here `trading_flow`). The actual units that are instantiated in these contexts and the events that these units publish, however, are under the control of the respective organisation. Additional flow constraints that further specify event flow policy in these contexts are managed directly by the respective organisations.

4.3 Policy enforcement

DEFCon engines enforce DPL policies by translating them into local constraints on event processing units. The process involves three new components implemented in DEFCon engines, which are presented in Figure 4.3:

Policy manager. The policy manager instantiates, authorises and checks DPL policies that refer to contexts local to an engine. It also coordinates with other policy managers in remote DEFCon engines to instantiate multi-domain policies.

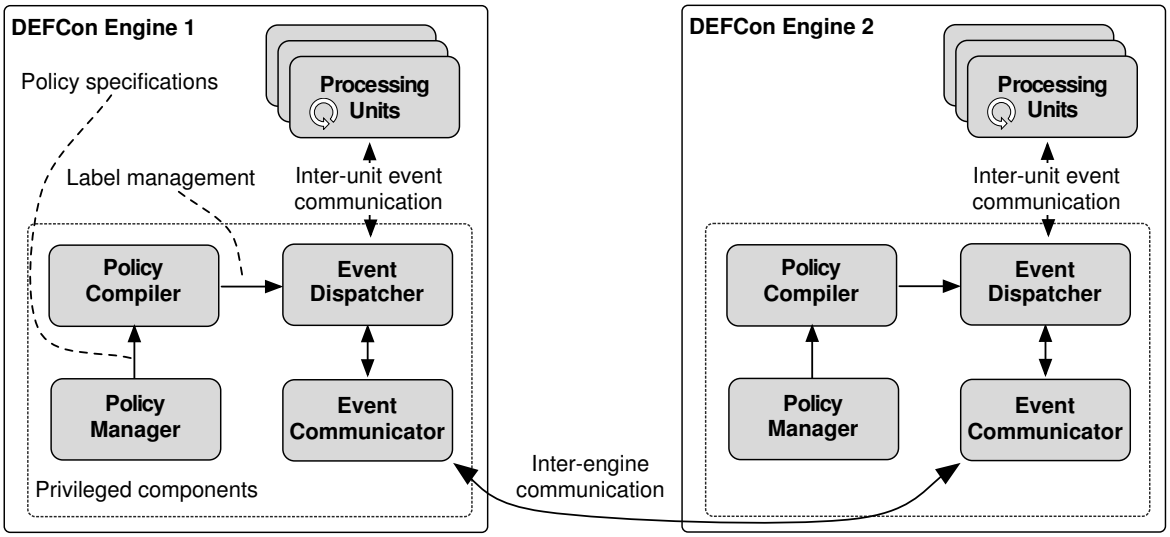

Figure 4.3: Overview of the DEFCon architecture with DPL support.

Policy compiler. The policy compiler translates DPL policies that have been checked by the policy manager into DEFC labels and privileges for enforcement.

Event communicator. Event communicators propagate events securely between DEFCon engines. They ensure that the local tags that are attached to events in each DEFCon engine correspond to the same DPL flow constraints.

The operations that these components perform are presented in detail in the following three sections.

4.3.1 Distributed policy management

To support large multi-domain deployments, DEFCon needs to handle many DPL processing contexts deployed across different engines. The policy manager is the central component that handles the deployment of such policies. It identifies the DEFCon engines that should participate in the enforcement of a new DPL policy, ensures that all engines authorise the policy and finally verifies that the policy does not result in inconsistencies such as those presented in Section 4. The following paragraphs describe these steps.

Step 1: Context to engine resolution

After the submission of a new DPL policy, the first step of the policy manager is to resolve each DEFCon engine responsible for contexts mentioned in the policy's flow constraints. A distributed directory service is used for this process. The directory service may also be federated to allow participating organisations to own part of the namespace. DNS fulfills these requirements.

Step 2: Engine trust verification

The second step of the policy manager is to verify that the local administrator trusts that all remote engines referenced by the policy enforce event flow. This is an important step because the actual DEFCon engines referenced in a policy are only known after context resolution. For example, the context `.uk.co.fsa.regulation.bank` from the `trading_flow` constraint in Section 4.2.5 may map to a local engine hosted by the bank while the context `.uk.co.hedge-fund.trading` may map to a remotely-hosted engine at the `defcon.hedge-fund.co.uk` domain. The policy manager has to verify that a remote engine and the organisation that manages it are trusted for event flow enforcement.

Each domain, such as `bank.co.uk`, specifies a set of engines that it trusts for policy enforcement. There are multiple possible choices for this, e.g. specifying trusted engines per organisation, per individual DPL policy or even per individual flow constraint. It is assumed that a trusted DEFCon engine correctly enforces event flow across all the units that it hosts, and therefore no additional trust is required per unit.

Step 3: Policy deployment and authorisation

With trust in all engines verified, the policy manager communicates the DPL policy to remote engines. At this step, it is the policy managers of the remote engines that must authorise the operation. The remote policy managers should not only be able to authenticate the engine that initiates the policy deployment but also they should determine whether the engine is allowed to place constraints on the contexts that the policy references. An authorisation scheme using PKI may be used for this purpose. For example, an SPKI-based [EFL+99] solution could use authorisation certificates [Gol99] that specify which DEFCon engines may place constraints on another engine's contexts. Identity certificates [Ell96] may then map DEFCon engines to organisations. Certificates can be integrated with the directory service from Step 1.

Step 4: Policy checking

The last step of the policy manager before instantiating a new policy is to verify that the policy does not lead to inconsistencies, as those described in the vertical separation example in Section 4.2.2. An inconsistent policy may violate liveness properties leading to units that are isolated from input or from output due to policy constraints.

The policy manager recursively fetches all policies that reference contexts from the new policy. It then performs a graph traversal to ensure that every context has at least a single event flow from the outside world that provides it with input (reachability) and at least a single output event flow towards the outside world (observability). While this strategy does not prevent all policy mis-configurations (e.g. a unit may require a different event flow policy

than the one imposed), it ensures the bare minimum: input and output are possible to and from every context referenced in the policy.

4.3.2 Translation to DEFC

After the policy was distributed to engines, the policy compiler in each engine sets up policy enforcement. DEFC is used for this purpose. The policy compiler translates DPL constraints to DEFC using a simple algorithm. For each flow constraint f, a tag pair (c_f, i_f) is created. The policy compiler chooses the labels for units affected by the constraint according to their context. It assigns privileges over c_f and i_f according to the position that a context appears in the DPL flow constraint:

- *Sandboxed and output units* are initialised with c_f and i_f in the confidentiality and integrity components of their *input* label, respectively. They are given c_f^+ and therefore can receive every event published inside the flow constraint. They are, however, limited to receive events that contain i_f, i.e. they can *only* receive events from inside the flow constraint.
- *Sandboxed and input* units are initialised with c_f and i_f in the confidentiality and integrity components of their *output* label, respectively. They are constrained to have their emitted events contain c_f, i.e. they publish inside the flow constraint.
- *Input units* are additionally given i_f^+ and therefore can produce events with i_f, even without having i_f in the integrity component of their input label. They are also given i_f^-. With these two privileges, they can receive events from outside of the flow and publish them inside the flow.
- *Output units* are given c_f^- to produce events without c_f, even if the confidentiality component of their input label contains c_f. Therefore, they are able to receive events from inside the flow constraint and publish events outside the constraint.

In the multi-domain example from Section 4.2.5, assume that the three sub-contexts of `uk.co.bank`, i.e. `bank_trading`, `brokering` and `portfolio_reporting`, are resolved to the the DEFCon engine at `defcon.bank.co.uk`. The policy compiler of this engine creates the c, i tags to represent the `trading_flow` constraint. Processing units are instantiated as follows:

- in `bank_trading`, with empty input label and $L^{out} = (\{c\}, \{i\})$;
- in `brokering`, with $L^{in} = L^{out} = (\{c\}, \{i\})$; and
- in `portfolio_reporting`, with $L^{in} = (\{c\}, \{i\})$ and an empty output label.

The policy compiler also reserves i^+ and i^- for units in `bank_trading`, c^+ for units in `brokering` and c^- for units in `portfolio_reporting`. Delegation privileges such as c^{-auth} and i^{+auth} are not allocated to units.

When a context is mentioned in multiple flow constraints, the labels and privileges of units instantiated in that context are the union of the corresponding labels and privileges as a result of each individual constraint. For example, a unit in a context sand-boxed by two constraints, f_1 and f_2, has $L^{in} = L^{out} = (\{c_{f1}, c_{f2}\}, \{i_{f1}, i_{f2}\})$.

Instantiating a unit at an arbitrary contamination level is possible due to the contamination independence property of the DEFCon API (§3.4.1). Units may process events at an arbitrary contamination level without having to adapt manually to the policy enforced.

4.3.3 Inter-engine communication

Units that execute in different DEFCon engines need to exchange events according to event flow policy. As described in Section 4.1, this requires additional support from the engine. DEFC only permits a unit to receive low integrity data from external sources and send sensitive data to arbitrary destinations if its input and output labels are empty. With empty input/output labels, a unit may not participate in the processing of sensitive events. Alternatively, a unit may process sensitive events and communicate with the external environment as long as it owns each tag in its input and output label. Allocating tag privileges, however, is unacceptable if an event flow policy is to constrain a unit—with declassification and endorsement privileges, units can ignore event flow enforcement.

DEFCon addresses this problem with a trusted proxy unit, the event communicator. The event communicator accumulates privileges for each tag used for DPL enforcement in a DEFCon engine. Therefore it can transfer every event published to another engine. Using the services of the event communicator in a remote DEFCon engine, the tags that protect an event locally are replaced by their equivalents in the remote engine. The mapping of tags across engines is agreed on-demand when a request to transfer an event is made. It relies on Step 3 of policy deployment (§4.3.1), which ensures that the relevant constraints are known to all engines with contexts referenced by a policy.

The multi-domain `trading_flow` constraint from Section 4.2.5 is an example in which inter-engine communication is required. The policy involves both the `defcon.bank.co.uk` and `defcon.hedge-fund.co.uk` engines (assuming such a resolution of the `.uk.co.bank` and `.uk.co.hedge-fund.trading` contexts). In order to enforce the `trading_flow` constraint locally, the policy manager at `defcon.hedge-fund.co.uk` first allocates the tags c, i and then ensures that units in `.uk.co.hedge-fund.trading` are instantiated with $L^{out} = (\{c\}, \{i\})$ and the i^+ privilege. These units cannot use the network to send events to `brokering` units in `defcon.bank.co.uk` directly. The reason is that their output events are sensitive according to the `trading_flow` constraint; only units in `.uk.co.fsa.regulation.bank` or `portfolio_reporting` may disclose data.

The two policy managers in `defcon.bank.co.uk` and `defcon.hedge-fund.co.uk` contact their local event communicators to delegate to them i^+, i^-, c^+ and c^-. When a unit in the `.uk.co.hedge-fund.trading` context sends an event to units in `defcon.bank.co.uk`, it

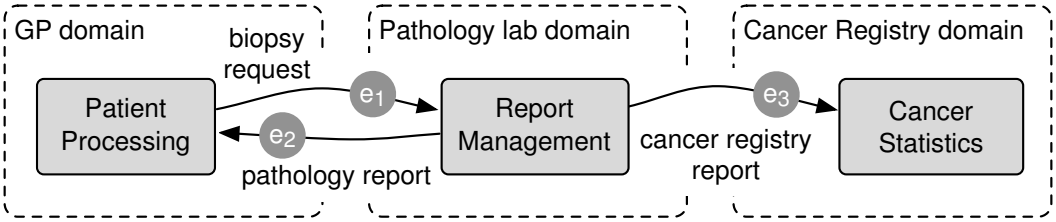

Figure 4.4: A multi-domain healthcare scenario for the evaluation of DPL. Different domains provide domain-specific DPL constraints while an NHS policy limits event flow.

routes the message through the local event communicator. It is the event communicator in `defcon.hedge-fund.co.uk` that exercises the c^- privilege to declassify, encrypt and transmit the message to the event communicator in `defcon.bank.co.uk`. The `defcon.bank.co.uk` event communicator uses its own local i^+ and c^+ privileges to ensure that the published message is received only by units in the contexts referenced in the `trading_flow` constraint.

4.4 Evaluation

The goal of the experimental evaluation is to demonstrate the ease-of-use and the effectiveness of DPL from a software developer's standpoint. DPL is used to specify event flow security policy for a multi-domain event processing application in healthcare. The section presents how DEFCon enforces DPL policy and compares event flow policy specification in DPL to manipulating DEFC labels and privileges directly.

4.4.1 Healthcare case study

The healthcare scenario used for the evaluation has been presented before in Section 2.1.2—it is repeated in Figure 4.4 for easy reference. The workflow begins when a patient visits a GP and the GP requests a biopsy. A Pathology Laboratory analyses the biopsy request (event e_1) and responds with a report to the GP (event e_2). A Cancer Registry must receive a report (event e_3) each time that the a Pathology Laboratory identifies a cancer incident. The event flow policy should provide the following three guarantees:

*G*1 Pathology reports are only sent to the GP that requested them or to the Cancer Registry.

*G*2 The Cancer Registry only receives pathology reports that involve cancer incidents.

*G*3 In the Pathology Lab, sensitive patient data are released only to doctors that perform biopsies.

Listing 4.1 shows an extract of the policy that enforces these guarantees. The extract consists of three different sections:

```
1  policy uk.nhs
2  sensitive[gp]: {
3    −> GP[gp].sensitive −>,
4    −> lab.doc[gp] −>,
5         lab.sensitive[gp],
6         cancer_registry.sensitive −>
7  }
8
9  policy uk.nhs.GP
10 patient_data[gp]: {
11   −> sensitive.patient_data,
12        sensitive.patient_data.anonymiser −>,
13        sensitive.pathology.patient_data −>,
14   −> sensitive.pathology.incoming_reports
15 }
16 anonymised_data[gp]: {
17   −> sensitive.patient_data.anonymiser,
18        statistics.anonymised_data −> ,
19        performance.anonymised_data −>
20 }
21 path_request[gp]: {
22   −> sensitive.pathology.test_requests,
23   −> sensitive.pathology.patient_data,
24        .uk.nhs.lab.doc[gp].pathology.request −>,
25        .uk.nhs.lab.sensitive[gp].pathology.patient_data −>
26 }
27
28 policy uk.nhs.lab
29 path_report[gp]: {
30   −> doc[gp].pathology.report,
31   −> sensitive[gp].pathology.patient_data,
32        .uk.nhs.GP[gp].sensitive.pathology.incoming_reports −>
33        sensitive[gp].pathology.cancer_registry_reporting −>
34 }
35 tumour_report: {
36   −> sensitive[gp].pathology.cancer_registry_reporting,
37        .uk.nhs.cancer_registry.sensitive.pathology.incoming −>
38 }
```

Listing 4.1: Extract of the DPL policy specification for the healthcare scenario [MPE+10b]. Constraints not related to the scenario of Figure 4.4 are omitted.

policy uk.nhs The first seven lines specify an NHS policy that constraints data propagation. It ensures that the sensitive data of GPs are only handled by specific contexts inside Pathology Laboratories and the Cancer Registry. The parameter `gp` is used to avoid introducing multiple identical policies for individual GPs. The `lab` context is subdivided into two sub-contexts, `lab.doc` and `lab.sensitive`, with only the first, which represents doctors, being trusted for input/output (lines 4–5). Finally, the `cancer_registry.sensitive` context allows the Cancer Registry to output reports for computing tumor statistics (line 6).

policy uk.nhs.GP GPs specify individual policies to refine or extend the constraints imposed by the global NHS policy (lines 9–26). This particular policy specifies three constraints that focus on (1) handling of patient data and incoming lab reports, (2) anonymisation of data for statistical purposes and (3) controlling the propagation of data in pathology requests. The first constraint (`patient_data`) states in line 12 that an `anonymiser` context processes events first, i.e. before units designed to operate on unclassified data (e.g. units that measure performance or calculate statistics in lines 16–20). Similarly, the `patient_data` constraint allows patient data to be used in pathology requests (line 13) while the `path_request` constraint restricts sensitive data propagation in other domains (lines 21–26). Line 14 states that incoming pathology reports may be combined with sensitive patient data.

policy uk.nhs.lab The lab specifies how units in its contexts interact when generating reports. The first constraint (lines 29–34) ensures that reports may combine data from the `lab.doc` and `lab.sensitive` contexts. The resulting reports may be disclosed only by units in the GP context (line 32) or to the Cancer Registry (line 33). The second `tumor_report` constraint in lines 35–38 restricts event processing when generating tumor reports.

4.4.2 Policy enforcement with DEFC

The specification of event flow policy in DPL is translated to DEFC tags and privileges using the process described in Section 4.3. Each flow constraint is enforced with a single tag pair (i_x, c_x). The x subscript in the tag symbols indicates the line number in Listing 4.1, in which the corresponding constraint is defined.

The enforcement of guarantee $G1$, which restricts the propagation of pathology reports, occurs via the c_{29}^{gp} and c_{35} tags. The tag c_{29}^{gp} protects the reports of a particular GP. Only units in two contexts possess the declassification privilege c_{29}^{gp-}, which is required to declassify reports: `GP[gp].sensitive.pathology.incoming_report` and `lab.sensitive[gp].pathology.cancer_registry_reporting`. The first context represents units of the requesting GP, and thus is the correct destination for reports. The second context restricts unit output to contain c_{35} due to the constraint in line 35. $G1$ is therefore enforced because only units in `cancer_registry.pathology.incoming` may remove the tag c_{35}.

The second guarantee, $G2$, which restricts the Cancer Registry to receive only cancer pathology reports, is enforced with tag i_{35}. Units in the `cancer_registry.pathology.incoming` context—tainted by i_{35}— may only receive events from units in the `lab.sensitive[gp].pathology.cancer_registry_reporting` context that possess the i_{35}^{+} privilege (line 33). While the actual identification of tumor reports depends on the correctness of units in `lab.sensitive[gp].pathology.cancer_registry_reporting`, DEFCon ensures afterwards that only events that have been identified as tumor reports may flow to the Cancer Registry.

The third guarantee, $G3$, which restricts patient data disclosure to doctors, is enforced using tag c_2^{gp}. Units in `GP[gp].sensitive` generate sensitive events protected with tag c_2^{gp}. The only units in the Pathology Laboratory that can declassify c_2^{gp} and reveal sensitive patient data to doctors are those in `lab.doc[gp]`. Due to GP policy, however, this does not occur directly: the `patient_data` constraint (line 10) requires that sensitive data published by units in `GP[gp].sensitive.patient_data` are also protected with tag c_{10}^{gp}. Units in `lab.doc[gp].pathology.request` belong to a sub-context of `lab.doc[gp]`. Therefore they possess the c_2^{gp-} privilege but lack c_{10}^{gp-}. Nevertheless, the flow constraint in line 21 allows units in `GP[gp].sensitive.pathology.patient_data` to swap c_{10}^{gp} with c_{21}^{gp} (horizontal flow separation), which units in `lab.doc[gp].pathology.request` can declassify.

The DPL policy fragment consists of 38 lines. If n is the total number of GPs in the system, the policy fragment generates $10n+2$ tags. These correspond to one pair of tags for each GP in every flow constraint in lines 1–34 and one pair of tags for `tumor_report`. Assuming that one unit is instantiated in each context, at least $14n+2$ units must be initialised with correct taints. The policy fragment also results in the distribution $34n+1$ privileges according to the procedure in Section 4.3. To provide just the initial set-up of labels manually, a programmer would have to call the low-level DEFCon API (§3.4.1) at least $24n+4$ times, i.e. one call to `changeInOutLabel()` and `changeOutLabel()` per unit. Instead, these calls, the creation of tags and the distribution of privileges are carried out by DEFCon automatically.

4.5 Summary

Event processing applications can be distributed across multiple administrative domains. Enforcing event flow in such scenarios is challenging because policy specification using DEFC primitives is prone to developer errors, couples policy with unit implementations and requires privileged unit code to transfer events between engines.

In this chapter, we presented how DEFCon achieves practical end-to-end enforcement of distributed event flow security policy based on the DEFCon Policy Language (DPL). DPL captures policies that control the propagation of events in DEFCon independently of the actions of the units involved. It decouples event flow policy from event processing.

This chapter provided details on DPL and sketched its semantics. DPL expresses event flow policy through the definition of constraints on event flow categories. Flow constraints are used to control the processing of events with similar event flow requirements. Event flows can

be separated vertically in order to isolate processing events with different security requirements from each other. Horizontal flow separation instead controls processing within a single flow by introducing specific ordering in how units process events. Flow constraints, however, reference processing contexts—not units. Processing contexts are hierarchical names that enable a domain to specify additional event flow restrictions between the controlled units.

DPL policies are checked for consistency and are translated to DEFC for enforcement. Three new components were introduced in the DEFCon architecture: (1) a policy manager coordinates engines to setup policies and checks that every processing context referenced is observable and reachable; (2) a policy compiler translates policies to DEFC; and (3) event communicators propagate events between engines and translate tags to their local representations at runtime.

The evaluation of DPL with a real-world healthcare workflow from the NHS demonstrates the practicality of the approach. Event flow security policy using DPL is easier to develop compared to manipulating DEFC labels and privileges directly, enables different domains to refine and enforce policy collaboratively, and reduces the number of DEFCon API calls that unit developers must perform.

In the next chapter, we shift our focus from the back-end of an organisation (in which event processing is often performed) to its front-end. Web applications may display a significant subset of the data that an organisation maintains internally, and thus they are an attractive target for attackers. PHP Aspis is a runtime taint tracking system that protects web applications from injection attacks while, similar to DEFCon, avoids modifications to the execution platform.

Chapter 5

Partial Taint Tracking for Protection against Injection Attacks

This chapter presents PHP Aspis [PMP11], our solution for practical protection of PHP web applications from injection attacks. PHP Aspis improves on past research on runtime taint tracking by combining three important ideas: (1) *partial* taint tracking, i.e. it propagates taint meta-data only to parts of a web application, which significantly minimises the performance overhead of the approach; (2) configurable taint categories, i.e. the administrator adapts the taint tracking policy to match the sanitisation operations of the web application and thus avoids redundant sanitisation operations; and (3) source code transformations, i.e. taint tracking is performed by transforming the source code of a web application without changes to the language interpreter.

PHP Aspis augments PHP values with taint meta-data to track their origin in order to detect and prevent injection attacks while they occur. To improve performance, it performs taint tracking only in an application's most vulnerable parts: third-party plugins. It then rewrites the source code of the web application to invoke additional methods that propagate taint meta-data explicitly during processing. Overall, taint tracking via source code transformations enables administrators to harden the security of web applications without the need to maintain a custom interpreter. Partial taint tracking reduces the performance overhead of the approach while we show that it effectively prevents real-world injection exploits.

This chapter starts with an introduction of why partial taint tracking is a suitable defence against injection attacks in web applications. Section 5.2 presents an overview of PHP Aspis as a runtime taint tracking system, and Section 5.3 describes the taint meta-data that it maintains. Section 5.4 covers the core of the transformations that PHP Aspis applies. Transformations on web application source code, however, create incompatibilities with parts of the application that are not transformed due to partial taint tracking. Section 5.5 explains how interoperability is achieved between these two parts of a transformed application.

PHP Aspis is evaluated using Wordpress, the popular open source weblog platform, and the chapter is summarised in Section 5.7.

5.1 The case for partial taint tracking

PHP is the most popular web development language, as indicated by web surveys[1], and its gentle learning curve often attracts less experienced developers. At the same time, PHP lacks fully automated features that protect web applications from injection attacks (§2.3.3). This increases the likelihood for less experienced developers to create web applications that are vulnerable to injection attacks [FW11].

Less experienced developers frequently extend mature web applications through third-party code in the form of *plugins*. Such extensibility is a popular feature for applications but leads to a significant security threat from plugins. In 2009, the CVE database [MIT12] reported that the Wordpress platform suffered from 15 injection vulnerabilities, out of which 13 were introduced by third-party plugins and only 2 involved the core platform. In 2010, the breakdown was similar: 10 vulnerabilities were due to plugins and only 2 due to Wordpress.

Not all application code is equally prone to injection vulnerabilities. For example, Wordpress spends much of its page generation time in initialisation code, setting up the platform before handling a user request. This involves time-consuming steps such as querying the database for installed plugins, setting them up and generating static parts of the response with theme-aware headers and footers. Injection vulnerabilities tend to exist in code that handles user-generated content: CVE-2010-4257, an SQL Injection vulnerability, involves a function that handles track-backs after a user published a post; CVE-2009-3891, an XSS vulnerability, involves a function that validates uploaded files; and CVE-2009-2851 and CVE-2010-4536, again XSS vulnerabilities, involve functions that display user comments. Overall, injection vulnerabilities are more likely to occur when less experienced developers handle user data.

Partial taint tracking exploits this observation to improve performance by propagating taint meta-data only in the most vulnerable parts of an application. We suggest a simple heuristic to decide when to propagate taint meta-data in a web application: focus on *parts of third-party plugins that handle user-generated data*. This restricts the propagation of taint meta-data to a small fraction of the codebase of a web application. As a consequence, the large performance penalty that exhaustive taint tracking at the source code level would incur is mitigated.

5.2 Overview

Figure 5.1 presents an overview of how PHP Aspis transforms applications. First, it identifies specific *data entry points*, in which the application receives data from users (e.g.

[1]Programming Languages Popularity Website, `http://langpop.com/`, last accessed: 19/9/2012

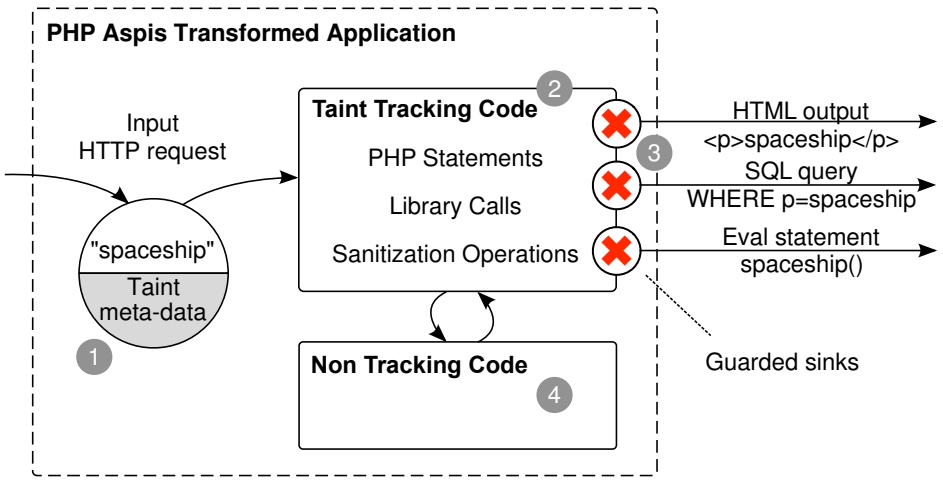

Figure 5.1: Overview of partial taint tracking using PHP Aspis.

through input HTTP requests) and transforms the application code to mark data as user-generated (label 1). Second, it divides the application's codebase into two parts: *tracking* and *non-tracking* code. Instead of propagating taint meta-data everywhere, it focuses on parts of the codebase that are more likely to contain code injection vulnerabilities (label 2). In tracking code, PHP Aspis records the origin of data and filters data at output statements when an injection vulnerability could exist (label 3). These output statements, in which data filtering occurs, are called *guarded sinks*. For non-tracking code, PHP Aspis does not perform taint tracking, trusting the code not to be vulnerable (label 4).

PHP Aspis prevents injection attacks transparently in existing applications. It achieves this via *character-level* taint tracking, i.e. it tracks the taint meta-data of each string character individually. As explained in Section 2.3.3, variable-level taint tracking implementations such as Ruby's taint mode are inadequate if the system must secure existing web applications without developer input and without introducing false positives. Consider an application that concatenates a user-provided value with a static HTML template, stores the result in variable `$v` and returns `$v` to the client as a response. Inferring that variable `$v` is tainted is not enough for transparent protection because, with this information alone, a runtime taint tracking system can only stop the operation. This will prevent normal users from using the application even if XSS is not performed. Instead, with character-level taint meta-data, PHP Aspis can sanitise only the user-generated parts of `$v` and secure the web application in the presence of XSS attacks transparently.

Figure 5.2 shows an application transformed with PHP Aspis in the context of the runtime taint tracking model from Section 2.3.1. PHP Aspis performs source code transformations in the web application to intercept operations on variables and to invoke tracking operations that propagate taint meta-data per string character. PHP Aspis stores and propagates one bit of taint meta-data per character and per taint category (explained next in Section 5.3). When the application invokes specific functions, in which an injection vulnerability may manifest (also referred to as *sink* functions), PHP Aspis imposes checking operations, i.e. *sink guards* (see Section 5.3). Sink guards inspect the taint meta-data of

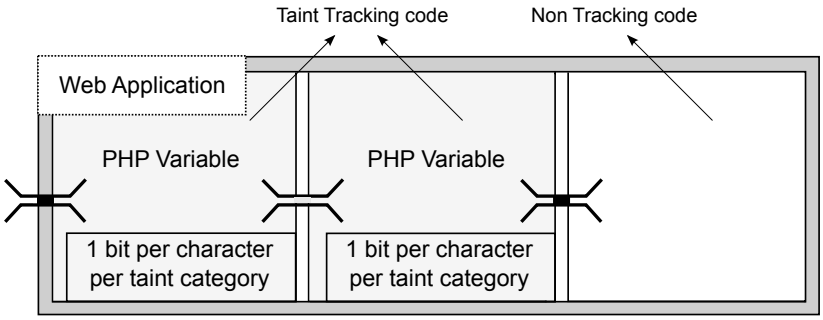

Figure 5.2: PHP Aspis as a runtime taint tracking system. PHP Aspis uses character-level tracking granularity that propagates taint meta-data within parts of an application.

the sink function's parameters and abort the invocation or transparently prevent a potential attack. As explained next, PHP Aspis allows the administrator of a web application to control the actions and the placement of checking operations via the definition of taint categories.

5.3 Taint meta-data representation

The taint tracking policy that PHP Aspis enforces depends on the targeted web application. First, only some types of injection vulnerabilities are feasible in a given web application. For example, Eval injection can only occur if the application uses the `eval` statement. Second, user data sanitisation depends on the business logic of a web application. For example, the administrator of a weblog may be allowed to add Javascript code in a post's comments while unregistered users should be unable to do so.

PHP Aspis adapts its taint tracking policy to propagate taint meta-data only for injection attacks that are possible in a given web application. It also monitors the application's sanitisation efforts to avoid sanitising data that the application considers secure. The next two sections describe how the application administrator controls the taint tracking policy in PHP Aspis and how the flexibility that PHP Aspis offers in specifying a taint tracking policy affects the taint meta-data that it propagates.

5.3.1 Taint categories

PHP Aspis tracks multiple independent and user-provided *taint categories*. A taint category is a generic way of defining how an application is supposed to sanitise data and how PHP Aspis should enforce that the application always sanitises data before they are used.

A taint category consists of a *set of sanitisation functions* and a *set of guarded sinks* (i.e. sink functions along with corresponding sink guards). Sanitisation functions can be PHP library functions or may be defined by the application. A sanitisation function is called by the application to transform untrusted user data so that they cannot be used for a particular type of injection attack. Commonly, sanitisation functions either transform unsafe character

Sanitisation functions	`htmlentities()` `htmlspecialchars()`

Guarded sinks	`echo()` ⇒ `AspisAntiXss()`
	`print()` ⇒ `AspisAntiXss()`
	...

Table 5.1: Excerpt of the definition of an XSS taint category in PHP Aspis.

sequences to safe equivalents (e.g. `htmlentities`) or filter out a subset of potentially dangerous occurrences (e.g. by removing `<script>` but not `<b>`). Calls to sanitisation functions by the application are intercepted, and PHP Aspis untaints the corresponding data to avoid sanitising them again.

Sink guards are functions that protect data flow to sensitive sink functions. When a call to a sink function is made, PHP Aspis invokes the sink guard with references to the parameters passed to the sink function. The sink guard is a user-provided function that has access to the relevant taint category meta-data and typically invokes one or more sanitisation functions for that taint category.

For example, Table 5.1 shows an excerpt of an XSS taint category definition. It specifies that a user-provided string can be echoed safely to the user after `htmlentities` or `htmlspecialchars` has been invoked on it. The second part lists all functions that can output strings to the user (e.g. `echo`, `print` etc.) and guards them with an external filtering function (`AspisAntiXss`). The sink guard either aborts the print operation or sanitises any remaining characters. The administrator can change the definitions of taint categories according to the requirements of the application.

By listing all the sanitisation functions of an application in the relevant taint category, PHP Aspis can monitor the application's sanitisation efforts and adapt its tracking operations accordingly. When applied to a well-designed application, PHP Aspis untaints user data as they get sanitised by the application, but before they reach the sink guards. Thus, sink guards can apply a simple, application-agnostic sanitisation operation (e.g. `htmlentities`) acting as a "safety net" for cases, in which the application developer has omitted a call to a sanitisation function. Some applications may not define explicit sanitisation functions or these functions may be omitted from the relevant taint category. In such cases, sink guards have to replicate the sanitisation logic of the application.

A different taint category must be used for each type of injection vulnerability. PHP Aspis tracks different taint categories independently from each other. For example, when a sanitisation function of an XSS taint category is called on a string, the string is still considered unsanitised for all other taint categories. This ensures that a sanitisation function for handling one type of injection vulnerability is not used to sanitise data for another type.

String	Taint meta-data
`$s='Hello'`	`array(0=>false)`
`$n='John'`	`array(0=>true)`
`$r=$s.$n`	`array(0=>false, 5=>true)`

Table 5.2: Representation of taint meta-data for a single taint category in PHP Aspis.

5.3.2 Storing taint meta-data

It is challenging to represent taint meta-data so that it supports arbitrary taint categories and character-level taint tracking. This is due to the following properties of PHP:

P1 PHP is not object-oriented. Although it supports objects, built-in types such as `string` cannot be augmented transparently with taint meta-data. This precludes solutions that rely on altered class libraries (as suggested for Java [CW09]).

P2 PHP does not offer direct access to memory. Any solution must track PHP references because variables' memory addresses cannot be used [XBS06].

P3 PHP uses different assignment semantics for objects ("by reference") compared to other types including arrays ("by copy"). This property of PHP prevents the substitution of a scalar type with an object without manually copying objects to avoid aliasing.

P4 PHP is a dynamically typed language, which means that there is no generic method to identify all string variables statically.

Due to these properties, PHP Aspis uses arrays to store taint meta-data. Table 5.2 shows how PHP Aspis encodes taint meta-data for a single taint category. For each string, it keeps an array of character taints, with the index representing the first character that has a given taint. In the example, string `$s` is untainted and `$n` is tainted. Their concatenation `$r` is untainted from index 0 to 4 and tainted from index 5 onwards. Numerical values use the same array-based approach for representing taint meta-data but only store a common taint for all digits.

Taint meta-data must remain associated with the value that they refer to. As shown in Table 5.3, this is achieved by enclosing the value and the taint meta-data inside an additional array. First, all scalars such as `'Hello'` and `12` are replaced with arrays (rows 1 and 2), which is referred to as the value's *Aspis*[2]. The Aspis contains the original value and an array of the taint meta-data for all currently tracked taint categories (`TaintCategories`). Similarly, scalars within arrays are transformed into Aspis-protected values.

According to property P4, PHP lacks static variable type information. Moreover, it offers type identification functions at runtime. When scalars are replaced with arrays, the system must be able to distinguish between an Aspis-protected value and a regular array. For this,

[2] "Aspis" is the name of the circular wooden shield used by soldiers in Ancient Greece.

	Original value	Aspis-protected value
1.	`'Hello'`	`array('Hello',TaintCategories)`
2.	`12`	`array(12,TaintCategories)`
3.	`array()`	`array(array(),false)`

4.	`array('John')`	`array(`
		` array(`
		`  array('John',TaintCategories)`
		` )`
		`,false)`

5.	`array(13=>20)`	`array(`
		` array(13=>`
		`  array(20,ContentTaintCategories,`
		`  KeyTaintCategories)`
		` )`
		`,false)`

Table 5.3: Augmenting values with taint meta-data in PHP Aspis.

the resulting arrays are themselves enclosed in an Aspis-protected value, albeit without any taint (`false` in rows 3 and 4). The original value of a variable can always be found at index 0 when Aspis-protected. Objects are handled analogously. Aspis-protected values can replace original values in all places except for array keys: PHP arrays can only use the types `string` or `int` as keys. To circumvent this, the key's taint categories are attached to the content's Aspis (`KeyTaintCategories`) and the key retains its original type (row 5).

Overall, this taint representation is compatible with the language properties mentioned above. By avoiding storing taint meta-data inside objects, an assignment cannot lead to two separate values referencing the same taint meta-data (P3). By storing taint meta-data in-place, variable aliasing correctly aliases taint meta-data (P2). Finally, by not storing taint meta-data separately, the code transformations that enable taint propagation can be limited to handling the original, Aspis-protected values correctly. As a result, the structure of the application in terms of functions and classes remains unchanged, which simplifies interoperability with non-tracking code, as explained in Section 5.5.

5.4 Taint tracking transformations

Based on the taint representation, PHP Aspis modifies an application to support taint tracking. The source code transformations employed to achieve this, i.e. the *taint tracking transformations*, are applied to the part of the codebase that is marked as tracking code (§5.2). These transformations achieve the first three steps from Figure 5.1: marking data as untrusted at data entry points, propagating taint meta-data and invoking sink guards at sink functions.

Original expression	Transformed expression
`$s.$t`	`concat($s,$t)`
`$l = &$m`	`$l = &$m`
`$j = $i++`	`$j = postincr($i)`
`$b+$c`	`add($b,$c)`
`if ($v) {}`	`if ($v[0]) {}`
`foreach ($a as $k=>$v) {...}`	`foreach ($a[0] as $k=>$v) {restoreTaint($k,$v) ...}`

Table 5.4: Example source code transformations in PHP Aspis. Transformations are used to propagate taint meta-data and preserve the original program semantics in the presence of Aspis-protected values.

5.4.1 Data entry points

Taint tracking transformations must mark any user-generated data as fully tainted, i.e. all taint meta-data for each taint category in an Aspis-protected value should be set to `true`. Any input channel, such as the incoming HTTP request, that is not under the direct control of the application may potentially contain user-generated data.

In each transformed PHP script, PHP Aspis inserts initialisation code that (1) scans the arrays that contain HTTP request data and identifies any values submitted to the script by the user; (2) replaces all submitted values with their Aspis-enclosed counterparts; and (3) marks user-submitted values as fully tainted (i.e. all taint meta-data for every taint category have the value `true`). All constants defined in the script are also Aspis-protected, but are marked as fully untainted (i.e. all taint meta-data for each taint category have the value `false`). As a result, all initial values are Aspis-protected in the transformed script.

5.4.2 Taint propagation

Next all statements and expressions are transformed to (1) operate with Aspis-protected values, (2) propagate taint meta-data correctly and (3) return Aspis-protected values.

Table 5.4 lists representative transformations for common operations supported by PHP Aspis. Functions in the right-hand column are introduced to maintain the original semantics and/or propagate taint meta-data. For example, `concat` replaces operations for string concatenation in PHP (e.g. double quotes or the concat operator ".") and returns an Aspis-protected result. Control statements are transformed to access the enclosed original values directly. All these transformations do not introduce additional statements—only the `foreach` transformation requires an extra call to `restoreTaint` to restore the taint meta-data of the key for subsequent statements in the loop body. Notice that, before the call to `restoreTaint` in the transformed expression, `$k` is a scalar value and not Aspis-protected. The meta-data of `$k` are stored with the content `$v` (see row 5 of Table 5.3). The call to `restoreTaint` ensures that `$k` is Aspis-protected for subsequent statements in the loop body.

PHP function library

Without modifications, built-in PHP functions cannot operate on Aspis-protected values and do not propagate taint meta-data. Since these functions are typically compiled for performance reasons, PHP Aspis uses *interceptor functions* to intercept calls to them and attach wrappers for taint meta-data propagation.

By default, PHP Aspis uses a generic interceptor for built-in functions. The generic interceptor reverts Aspis-protected parameters to their original values and wraps return values to be Aspis-protected again. This default behaviour is acceptable for library functions that do not propagate taint meta-data (e.g. `fclose`). However, the removal of taint meta-data from result values may lead to false negatives in taint tracking. PHP Aspis therefore provides custom interceptor functions for specific built-in functions (e.g. for `sort` and `trim`), which ensure correct propagation of taint meta-data to the functions' output. By increasing the number of intercepted functions, the accuracy of taint tracking is improved and this reduces the number of false negatives.

The purpose of a custom interceptor is to propagate taint meta-data from the input parameters to the return values. Such interceptors rely on the well-defined semantics of library functions. When possible, the interceptor computes the meta-data of the return value based on the meta-data of the inputs (e.g. `substr`). It then removes the taint meta-data from the input values, invokes the original library function and attaches the computed meta-data to the result value. Alternatively, the interceptor compares the result value to the value of the function's input parameters and infers the taint meta-data of the result.

As an example, consider the case of `stripslashes`, a function that removes backslashes from a string. Assume that the interceptor for `stripslashes` receives a string with a taint category `array(0=>false,5=>true)`. The comparison of the original to the result string identifies the actual character indices that `stripslashes` removed from the original string; assume here, for simplicity, that only index 2 was removed. To calculate the taint meta-data of the result, the interceptor subtracts from all taint category indices the total number of characters removed before each index. Thus, it returns `array(0=>false,4=>true)` in the given example. In total, 66 of the interceptors provided with the current prototype of PHP Aspis use this method.

For other functions, the interceptor can use the original function to obtain a result with the correct taint meta-data automatically. For example, `usort` sorts an array according to a user-provided callback function and thus can sort Aspis-protected values without changes. If the callback is a library function, the function is unable to compare Aspis-protected elements and calls to `usort` would fail. When callbacks are used, custom interceptors introduce a new callback replacing the old. The new callback calls the original callback after removing taint meta-data from its parameters. In total, 21 of the interceptors provided with the current prototype of PHP Aspis use this method.

For cases in which the taint meta-data of the result cannot be determined, such as for `sort`,

an interceptor provides a separate, taint-aware version of the original PHP library function. 19 library functions are re-implemented in this way.

The current PHP Aspis prototype provides custom interceptors for 106 library functions. These include most of the standard PHP string library, which is enough to propagate taint meta-data in Wordpress effectively (see Section 5.6).

Dynamic features

PHP has many dynamic features, such as variable variables, variable function calls and the `eval` and `create_function` functions (§2.2.1). These are not compatible with Aspis-protected values, but PHP Aspis must nevertheless maintain correct semantics.

Variable variables allow dynamic access to a variable based on the string stored in another variable. Their transformation only requires access to the enclosed string. Thus a dynamic access to a variable named `$v` is transformed from `$$v` to `${$v[0]}`.

Variable function calls allow a script to call a function that is statically unknown. They are supported in PHP using variables when invoking functions (e.g. `$v()`) or via specific library functions (e.g. `call_user_func_array`). PHP Aspis transforms these calls and inspects them at runtime. When a library call is detected, PHP Aspis generates an interceptor, as described in the previous section, at runtime.

The functions `eval` and `create_function` are used to execute code generated at runtime. Since application-generated code does not propagate taint meta-data, PHP Aspis must apply the taint tracking transformations at runtime. To avoid a high runtime overhead, PHP Aspis uses a caching mechanism, such as Zend Cache[3], when available.

5.4.3 Guarded sinks

PHP Aspis protects web applications from injection vulnerabilities transparently using the taint meta-data and the defined taint categories. As described in Section 5.3.1, sink guards (specified as part of the taint categories) are invoked before the calls to their respective sink functions. Sink guards use PHP's sanitisation routines or define their own and transparently sanitise untrusted user data before they are used in an unsafe way. For example, an SQL filtering routine rejects queries with user-provided SQL operators or keywords [NTGG+05].

5.5 Partial taint tracking

The taint tracking transformations used by PHP Aspis require extensive changes to the source code, which has an adverse impact on execution performance. To preserve program

[3]Zend Framework, `http://framework.zend.com/`, last accessed: 20/10/2012

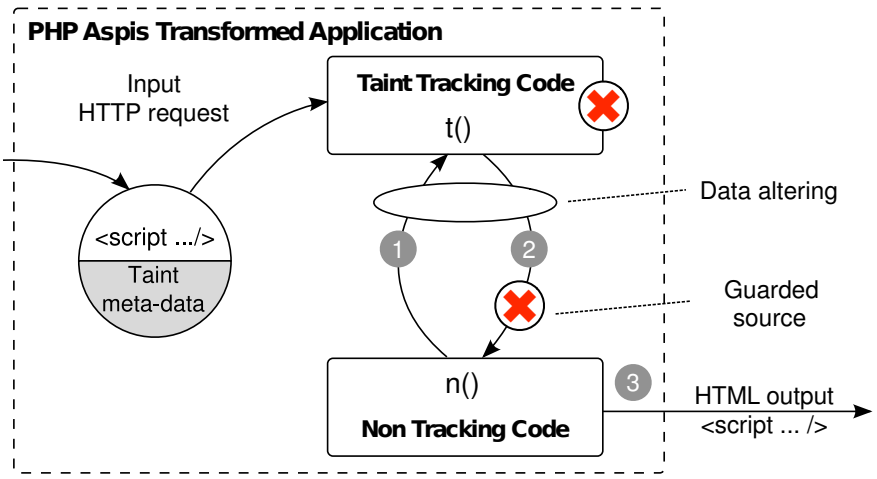

Figure 5.3: Example of an XSS vulnerability manifesting in non-tracking code.

semantics, transformations frequently require the replacement of efficient low-level operations by slower, high-level ones (see Table 5.4).

Partial taint tracking aims to improve execution performance by limiting taint tracking to the parts of the application, in which injection vulnerabilities are more likely to exist. Partial taint tracking can be applied at the granularity of *contexts*: functions, classes or the global scope. The administrator can assign each of these to be of the following *types*: *tracking* or *non-tracking*. The decision to classify application contexts into tracking or non-tracking is based on the trust that the application administrator has in the developers of a given part of the codebase.

The next two sections discuss the implications of partial taint tracking in an application transformed by PHP Aspis. First, the presence of non-tracking code reduces the ability of PHP Aspis to prevent injection attacks and, second, it requires additional transformations to preserve program semantics. The following two sections describe how these problems are overcome.

5.5.1 Detecting vulnerabilities in presence of non-tracking code

With partial taint tracking, PHP Aspis must maintain the ability to detect and mitigate injection vulnerabilities in tracking code. When parts of the codebase are classified as non-tracking code, injection vulnerabilities within these parts cannot be detected. However, in the presence of non-tracking code, vulnerabilities that exist in tracking code may result in injection attacks that only manifest themselves in non-tracking code. PHP Aspis must detect and prevent such attacks.

Assume, for example, that the non-tracking function `n` in Figure 5.3 calls the tracking function `t` (step 1). It receives a user-provided value `$v` from function `t` (step 2) and prints this value (step 3). If `t` fails to escape user input, `n` is unable to sanitise the data transparently. This is because taint meta-data are not available in non-tracking code and calls to sensitive

sinks, such as `print`, are not intercepted. From the perspective of `n`, `t` acts as the source of user generated data that must be sanitised before they leave a tracking context.

To address this issue, PHP Aspis takes a conservative approach. It can sanitise data at the boundary between tracking and non-tracking code. Taint categories are augmented for this purpose. Non-tracking application functions must be added as sinks, and the administrator should specify guard functions that sanitise user data passed as function parameters when non-tracking functions are called from tracking code.

PHP Aspis adds another type of guards, *source guards*, to each taint category in order to provide sanitisation functions when non-tracking code invokes a tracking function. A source is a tracking function and the guard is the sanitisation function applied to its *return value* when called from non-tracking code. In the example from Figure 5.3, the tracking function `t` can act as a source of user-generated data when called from the non-tracking function `n`. A guard for `t` would intercept `t`'s return value and apply `htmlentities` to any user-generated characters. Sink and source guards ensure that user data are sanitised properly before they can be used in non-tracking code.

Note though that this early sanitisation is an additional operation introduced by PHP Aspis. Thus, if the non-tracking context that received the data attempts to sanitise them again, the application would fail. Moreover, there is no generic sanitisation routine that can always be applied because the final use of the data is unknown. Instead, early sanitisation is only suitable for cases when both the final use of the data is known and the application does not perform any additional sanitisation. This often holds for third-party plugin APIs, which are typically well-documented.

5.5.2 Compatibility transformations

The taint tracking transformations in Section 5.4 generate code that handles Aspis-protected values. For example, a tracking function that changes the case of a string parameter `$p` expects to find the actual string in `$p[0]`. Such a function can no longer be called directly from non-tracking code with a simple string for its parameter. Instead, PHP Aspis requires additional transformations to intercept this call and automatically convert `$p` to an Aspis-protected value, which is marked as fully untainted. *Compatibility transformations* are a set of additional transformations that aim to preserve program semantics when PHP Aspis is configured for partial taint tracking.

Compatibility transformations make changes to both tracking and non-tracking code. These changes alter the data that are exchanged between a tracking and a non-tracking context, i.e. data exchanged between functions, classes and code in the global scope. They strip Aspis-protected values when passed to non-tracking contexts and restore Aspis protection for tracking contexts.

Function calls

A function call is the most common way of passing data across contexts. PHP Aspis transforms all cross-context function calls: a call from a tracking to a non-tracking context has its taint meta-data removed from parameters and the return value Aspis-protected again. The opposite happens for calls from non-tracking to tracking contexts. This also applies to method calls.

Transforming parameters and return values is similar to using the default interceptor function from Section 5.4. User code, however, can share objects of user-defined classes. Instead of transforming each internal object property, PHP Aspis uses proxy objects that decorate the objects that are passed as parameters or returned as results. Consider an object `$o` of class `c` and assume that `c` is marked as a tracking context. When `$o` is passed to the non-tracking context of function `f`, `f` is unable to access `$o`'s state directly or call its methods. Instead, it receives the decorator `$do` that points to `$o` internally. `$do` is then responsible for transforming the parameters and the return values of method calls when such calls occur. It also handles reads and writes of public object properties.

PHP also supports call-by-reference semantics for function parameters. Since changes to reference parameters by the callee are visible to the caller, these parameters effectively resemble return values. Compatibility transformations handle reference parameters similarly to return values—they are transformed to match the calling context (i.e. to Aspis-protected values if the calling context is tracking and to normal values if the calling context is non-tracking) after the function call returns.

This behaviour can lead to problems if a single variable is aliased by contexts of different types, i.e. if a tracking class internally stores a reference to a variable also stored in a non-tracking class. In such cases, PHP Aspis can no longer track these variables effectively across contexts because references to them are not passed explicitly from one context to the other (e.g. as a function call parameter) right before the variables are used. This forces the administrator to mark both contexts as tracking or non-tracking. Since shared references to internal state make it hard to maintain class invariants, they are considered bad practice [Blo08], and a manual audit did not reveal any occurrences in Wordpress.

Accessing global variables

PHP functions can access references to variables in the global scope using the `global` keyword. These variables may be Aspis-protected or not, depending on the type of the current global context and previous function calls. The compatibility transformations rewrite `global` statements: when the imported variable does not match the context of the function, the variable is altered so that it can be used by the function. After the function returns, all imported global variables must be reverted to their previous forms—`return` statements are preceded with the necessary reverse transformations. When functions do not return values, reverse transformations are added as the last function statement.

Accessing superglobal variables

PHP also supports the notion of *superglobals*: arrays that include the HTTP request data and can be accessed from any scope without a `global` declaration. Data in these arrays are always kept Aspis-protected; removing their taint meta-data would effectively stop taint tracking everywhere in the application. The reason is that if non-tracking code modifies the data stored in superglobal arrays, the taint meta-data stored there would be invalidated, and therefore the taint meta-data would have to be discarded. As a result, only tracking contexts are permitted to access superglobals directly. In addition, compatibility transformations enable limited access from non-tracking contexts when access can be statically detected (for example, a direct read to `$_GET` but not an indirect access through an aliased variable). This is because PHP Aspis does not perform static alias analysis to detect such indirect accesses [JKK06b].

Include statements

PHP's global scope includes code outside of function and class definitions and spans across all included scripts. Compatibility transformations can handle different context types for different scripts. This introduces a problem for variables in the global scope: they are Aspis-protected when they are created by a tracking context but have their original value when they are created by a non-tracking context.

To address this issue, PHP Aspis transforms temporarily all variables in the global scope to be compatible with the current context of an included script, before an `include` statement is executed. After the `include`, all global variables are transformed again to match the previous context type. In PHP, two different files may serve different types of HTTP requests or may be combined (i.e. via `include`) in order to serve the same type of requests. In the latter case, transforming all variables in the global scope every time `include` is called significantly reduces performance. To mitigate this performance overhead, code in the global scope of different files but used to handle the same type of requests should be of the same context type (i.e. tracking or non-tracking).

Dynamic features

Compatibility transformations intercept calls to `create_function` and `eval` at runtime. PHP Aspis rewrites the code provided as an input parameter to these functions based on the context type of the caller: when non-tracking code calls `eval`, only the compatibility transformations are applied and non-tracking code is generated. Moreover, `create_function` uses a global array to store the context type of the resulting function. This information is used to adapt the function's parameters and return value in subsequent calls.

5.6 Evaluation

The goal of the evaluation section is to explore the effectiveness of PHP Aspis in preventing real-world vulnerabilities and to measure the performance penalty for transformed applications. To achieve this, PHP Aspis is used to secure an installation of Wordpress, a popular open source web logging platform, with known vulnerabilities.

The section first describes how an administrator sets up PHP Aspis to protect a Wordpress installation. It then discusses the vulnerabilities observed and shows how PHP Aspis addresses them. Finally, the performance penalty that PHP Aspis incurs is measured for multiple applications, and the evaluation finishes with a short discussion about the practicality of the approach.

5.6.1 Securing Wordpress

The extensibility of Wordpress relies on a set of hooks defined at certain places during request handling. User-provided functions can attach to these hooks and multiple types are supported: *actions* are used by plugins to carry out operations in response to certain events (e.g. sending an email when a new post is published), and *filters* allow a plugin to alter data before they are used by Wordpress (e.g. a post must receive special formatting before being displayed).

A plugin contains a set of event handlers for specific actions or filters and their initialisation code. Plugin scripts can also be requested directly via HTTP requests from a client (i.e. via direct requests to `.php` files of the plugin). The web server executes these plugin scripts and returns their response to the client. In such cases, plugin scripts execute outside of the main Wordpress page generation process.

Each plugin is secured from injection vulnerabilities using PHP Aspis as follows.

1. The functions, classes and scripts defined by the plugin are listed and the relevant contexts are marked as *tracking*.

2. The plugin is automatically inspected for possible sensitive sinks, such as print statements and SQL queries. This step involves deciding upon the taint categories to be used in order to avoid irrelevant tracking (i.e. avoid maintaining taint meta-data for Eval injection if no `eval` statements exist).

3. A list of event handlers is obtained from the `add_filter` statements used by the plugin. The taint category definitions are augmented with these handlers as guarded sources because return values of filters are subsequently used by Wordpress (§5.5).

4. Any plugin initialisation code is classified as non-tracking because it is less likely to contain injection vulnerabilities (§5.1).

5.6.2 Security

Table 5.5 lists all injection vulnerabilities reported in Wordpress plugins in 2010 and in the first quarter of 2011. For each vulnerable plugin, the vulnerability is verified using the attack vector described in the CVE report. The same attack vector is then replayed on an installation protected by PHP Aspis. The table shows whether the attack is prevented in the installation protected with PHP Aspis. The table also presents the total number of guarded sources specified as part of the taint category to secure the event handlers that each plugin registers as filters (§5.6.1).

The experiments are done on the latest vulnerable plugin version, as mentioned in a CVE report, running on Wordpress 2.9.2. PHP Aspis manages to prevent most vulnerabilities, which can be summarised according to three different categories:

Direct request vulnerabilities

The most common type of vulnerability involves direct requests to plugin scripts. Many scripts do not perform sanitisation for certain parameters (e.g. CVE 2010-4518, CVE 2010-2924, CVE 2010-4630, CVE 2010-4747 and CVE 2011-0740). Others do not anticipate invalid parameter values and neglect to sanitise when printing error messages (e.g. CVE 2010-4637, CVE 2010-3977 and CVE 2010-1186).

PHP Aspis manages to prevent all attack vectors described in these CVE reports by propagating taint meta-data correctly from the HTTP parameters within the plugin scripts. Before printing or script termination statements such as `die` and `exit` are called, PHP Aspis invokes sanitisation functions and prevents the injection attacks from succeeding.

Action vulnerabilities

Some of the plugins tested (e.g. CVE 2010-4402, CVE 2011-0641 and CVE 2011-1047) introduce a vulnerability in an action event handler. Similarly, a few other plugins (e.g. CVE 2010-2924, CVE 2010-1186 and CVE 2011-1047) only exhibit a vulnerability after a direct request but explicitly initialise Wordpress before servicing such a request. The common characteristic of these two scenarios is that Wordpress, upon initialisation, transforms the `$_GET` and `$_POST` superglobal arrays (these arrays store the parameters of the HTTP request) by applying some preliminary functions to their values in `wp-settings.php`. As the Wordpress initialisation code is classified as non-tracking, these functions effectively remove all taint meta-data from the HTTP parameters and introduces a false negative for all plugins that execute after Wordpress has loaded.

Since this behaviour is common for all plugins, taint tracking is used in a limited set of Wordpress contexts—the functions `add_magic_quotes`, `esc_sql` and the `wpdb` class—which are invoked by the initialisation code of Wordpress. As the assignment statements that alter the superglobal arrays are in the global scope of the `wp-settings.php` file, taint tracking is

CVE	Type	Description	Guarded Sources	Prevented
2010-4518	XSS	Plugin script `wp-safe-search-jx.php`, when called directly, uses `$_GET` parameters to construct the response.	1	✓
2010-2924	SQLI	Plugin script `myLDlinker.php`, when called directly, uses the `$_GET` parameter `url` to concatenate a query. The parameter is only partially sanitised and it can contain SQL keywords.	2	✓
2010-4630	XSS	Plugin script `create.php`, when called directly, echoes the `$_REQUEST` parameter `action` to the client without any sanitisation.	0	✓
2010-4747	XSS	Plugin script `popup.php`, when called directly, echoes the `$_GET` parameter `pluginurl` to the client without any sanitisation.	1	✓
2011-0740	XSS	Plugin script `magpie_slashbox.php`, when called directly, echoes the `$_GET` parameter `rss_url` without any sanitisation.	1	✓
2010-4637	XSS	Plugin script `handler_image.php`, when called directly, calls a third party library with the `$_GET` parameter `i`. The library echoes the parameter if it does not have a valid value.	2	✓
2010-3977	XSS	Plugin script `lib_ajax.php`, when called directly with an invalid `$_GET` parameter `rs`, echoes the invalid value to the client.	5	✓
2010-1186	XSS	Plugin script `media_rss.php`, when called directly with an invalid `$_GET` parameter `mode`, echoes the invalid value to the client.	15	✓
2010-4402	XSS	Plugin handler for the `register_form` action prefills the Wordpress registration page with multiple `$_GET`/`$_POST` parameters.	6	✓
2011-0641	XSS	Plugin function `iriStatPressSearch` concatenates multiple `$_GET` parameters into an SQL query without sanitisation and echoes that query.	2	✓
2011-1047	SQLI	Plugin scripts `wpf.class.php`, `wpf-post.php` use `$_GET`/`$_POST` parameters to concatenate SQL queries without sanitisation. Plugin script `feed.php` behaves similarly when called directly.	1	✓
2010-4277	XSS	Plugin handler for the `the_content` filter edits a post before display and attaches video embedding code.	3	✓
2011-0759	XSS	Plugin handlers for various filters echo to the client configuration fields stored in the local database that are not sanitised.	9	✗
2011-0760	XSS	Plugin handler for the `the_content` filter echoes to the client data fields stored in the local database that are not sanitised.	1	✗
2010-0673	SQLI	The plugin was not publicly available.	–	–

Table 5.5: Injection vulnerabilities in Wordpress plugins; as reported by the CVE database in 2010 and the first quarter of 2011. PHP Aspis prevents the majority of attacks from succeeding.

performed in this context as well to ensure that taint meta-data are not lost.

Unfortunately, this file is central to the initialisation of Wordpress: enabling taint tracking there leads to substantially reduced performance. To avoid this problem, a small function is introduced that encloses the assignment statements and its context is marked as tracking. This change required an extra three lines of code to define and call the function but it improved performance significantly.

Filter vulnerabilities

From all tested plugins, only one (CVE 2010-4277) introduces a vulnerability in the code attached to a filter hook. Although the behaviour described in the CVE report is verifiable, it is likely intended functionality of Wordpress: the Javascript injection can be done only by a user who is permitted to post Javascript-enabled text. PHP Aspis correctly marks the post's text as untainted before it is passed to the filter hook and avoids a false positive.

This plugin, however, was the only one with a reported vulnerability in code attached to a filter hook. To test the associated source guard despite the discussion in the previous paragraph, the plugin was edited to receive the text of posts from a tainted `$_GET` parameter instead of the Wordpress hook. This indeed introduces an XSS vulnerability. With this change, however, PHP Aspis properly propagates taint meta-data and correctly escapes the dangerous Javascript in the source guard applied to the filter hook.

False positives and negatives

The taint tracking performed by PHP Aspis may result in false negatives. By propagating taint meta-data accurately (see §5.6.4), the problem of false positives is largely avoided. False negatives, however, can occur because they are introduced (1) by built-in library functions that do not propagate taint meta-data; (2) by calls to non-tracking contexts; and (3) by data flows that involve the file system or the database. In these cases, taint meta-data are removed from data, and when that data are subsequently used, vulnerabilities cannot be prevented. PHP Aspis' current inability to propagate taint meta-data in the database is the reason why the XSS vulnerabilities CVE 2011-0760 and CVE 2011-0759 are not prevented.

To reduce the rate of false negatives, interceptors are used that perform precise taint tracking for all built-in library functions called by the tested plugins. In addition, classifying the aforementioned set of Wordpress initialisation routines as tracking contexts is sufficient to prevent all other reported injection vulnerabilities. Note that the last vulnerable plugin (CVE 2010-0673) was withdrawn and thus not available for testing.

5.6.3 Performance

To evaluate the performance impact of PHP Aspis, the page generation time is measured for:

App.	Tracking	Page generation	Penalty
Prime	Off	44.9 ms	-
Prime	On	466.8 ms	10.4×
DB	Off	0.4 ms	-
DB	On	1.3 ms	3.4×
WP	Off	65.6 ms	-
WP	On	394.4 ms	6.0×
WP	Partial	144.3 ms	2.2×

Table 5.6: Performance overhead of PHP Aspis in terms of page generation time.

- a simple prime number generator that tests each candidate number by dividing it by all smaller integers (Prime);
- a typical script that queries the local database and returns an HTML response (DB);
- Wordpress (WP) with the vulnerable Embedded Video plugin (2010-4277). Wordpress is configured to display a single post with a video link, which triggers the plugin on page generation.

The measurements were taken on a 3 Ghz Intel Core 2 Duo E6850 machine with 4 GiB of RAM, running Ubuntu 10.04 32-bit. The machine had PHP 5.3.3 and Zend Server 5.0.3 CE installed, with Zend Optimizer and Zend Data Cache enabled. For each application, tracking was enabled for two taint categories, XSS and SQL Injection.

Table 5.6 shows the 90^{th} percentile of page generation times over 500 requests for various configurations. Overall, performing taint tracking in the whole application (Tracking: On) has a performance impact that increases page generation between 3.4× and 10.4×. The overhead of PHP Aspis depends on how CPU intensive the application is: DB is the least affected because its page generation is the result of a single database query. On the other hand, Prime has the worst performance penalty of 10.4×, mostly due to the replacement of efficient mathematical operators with function calls.

Performing taint tracking everywhere in Wordpress (WP) results in a 6.0× increase of page generation time. With partial taint tracking configured only on the installed plugin (Tracking: Partial), page generation overhead is reduced significantly to 2.2×. Given that Wordpress uses globals extensively, the main source of performance reduction for the partial taint tracking configuration are the checks on global variable access as part of the compatibility transformations.

Although performing taint tracking everywhere in an application at the source code level incurs a significant performance penalty, partial taint tracking can reduce the overhead considerably. In practice, a 2.2× performance overhead when navigating Wordpress pages with partial taint tracking is acceptable for deployments, in which security has priority over performance.

5.6.4 Discussion

PHP Aspis aims to be a practical tool for hardening existing PHP web applications against injection attacks. It follows a design philosophy that assumes data are trusted when the taint tracking system is unable to track taint propagation. In practice, this means that if PHP Aspis cannot assess fully how to update taint meta-data (e.g. when accessing the `i`-th element of an array `a`, in which `i` is tainted but the value stored in `a[i]` is not—should `a[i]` be considered tainted?), it considers the result not to be tainted. The alternative philosophy—assuming such data are untrusted—may result in false positives and affect application semantics [SB09]. PHP Aspis favours a reduced ability to track data propagation on some execution paths, leading to false negatives. For this, PHP Aspis should not be trusted as the sole mechanism for protection against injection attacks.

Partial taint tracking is suited for applications, in which a partition between trusted and untrusted components is justified, e.g. in applications that employ third-party code. In addition, interactions across such components must be limited because, if data flow from a tracking to a non-tracking context and back, taint meta-data may be lost. PHP Aspis also does not propagate taint meta-data in file systems or databases. However, techniques for this have been presented earlier [DC10, YWZK09].

Although the performance penalty of PHP Aspis can multiply the page generation time several times, if taint tracking is limited only to a subset of a web application, the performance penalty is reduced significantly while many real world vulnerabilities are mitigated. The evaluation with Wordpress shows that PHP Aspis can offer increased protection for a Wordpress installation when a moderate increase in page generation time is acceptable.

The evaluation of partial taint tracking with PHP Aspis, however, does not show the broad applicability of partial taint tracking across different taint tracking systems and applications. The 63% speedup of partial taint tracking compared to performing taint tracking everywhere in Wordpress is indicative but does not prove that other taint tracking systems would benefit in the same way. The reason is that the performance gains observed may be significant only because performing taint tracking in PHP Aspis using source code transformations is slow. Nevertheless, a similar system for applications written is C [CSL08] also demonstrates significant performance gains. Exploring the applicability and the performance impact of partial taint tracking as a generic method for reducing the overhead of variable-level runtime taint tracking systems is left for future work.

PHP is a language without formal semantics. Available documentation is imprecise regarding certain features (e.g. increment operators and their side effects), and there are behavioural changes between interpreter versions (e.g. runtime call-by-reference semantics). Although the source code transformation approach in PHP Aspis requires changes when the language semantics changes, this cost is smaller than the maintenance of a third-party runtime implementation that requires updates even with maintenance releases.

The taint tracking transformations in PHP Aspis support most common PHP features, as

specified in the online manual[4]. Support for newer features from PHP5, such as namespaces or closures, has not been implemented yet. This precludes an evaluation of the current PHP Aspis prototype with applications that take advantage of these features (e.g. Drupal) without additional engineering effort.

5.7 Summary

Injection vulnerabilities are a result of web applications using untrusted user data without sanitisation. Manually ensuring the lack of injection vulnerabilities is challenging because it requires that developers comprehensively track and sanitise user-provided data before they are used in sensitive operations.

In this chapter, we presented PHP Aspis, a tool that applies partial taint tracking at the source code level, with a focus on third-party extensions. PHP Aspis avoids the need for taint tracking support in the PHP runtime. Instead it transforms application source code in order to propagate taint meta-data explicitly as the application processes data.

PHP Aspis tracks taint meta-data for multiple taint categories. Taint categories allow it to adapt to the sanitisation operations of different web applications and not replicate sanitisation effort. At a high level, a taint category describes how PHP Aspis propagates taint meta-data and how it uses these to protect against a single type of injection attacks. Taint categories also specify the sanitisation functions that the web application is supposed to use. Sanitisation functions are trusted to operate correctly by PHP Aspis. Therefore any data that they return have their meta-data removed. Taint categories also define guard functions, i.e. additional functions invoked automatically when calls to functions that can be used in injection attacks are detected. Taint categories correspond directly to additional bits of taint meta-data that PHP Aspis maintains per string character.

PHP Aspis divides application source code into tracking and non-tracking code. In tracking code, all PHP values are transformed to Aspis-protected values, which are arrays that also contain taint meta-data. Taint tracking transformations applied on tracking code ensure that operations on Aspis-protected values maintain their semantics and additionally propagate taint meta-data. A second set of compatibility transformations is applied to both tracking and non-tracking code. These ensure the interoperability of tracking and non-tracking parts of the application codebase.

The evaluation of PHP Aspis with Wordpress highlights the effectiveness of the approach in preventing injection attacks. Partial taint tracking focused on Wordpress plugins reduces the performance penalty of the approach by several times while still mitigating many injection vulnerabilities.

Despite these contributions, PHP Aspis is limited by the lack of support for taint tracking in the PHP interpreter. The source code transformations of PHP Aspis make the resulting

[4]PHP Online Manual, `http://www.php.net/manual`, last accessed: 18/1/2012

applications easy to deploy (especially when compared to maintaining a third-party PHP interpreter) but deployments in organisations with significant performance requirements are challenging. In the next chapter, we present SafeWeb, a runtime taint tracking system that combines design choices from both DEFCon and PHP Aspis to control sensitive data dissemination, end-to-end, across an organisation. We show that the Ruby language allows developers to collect and maintain meta-data with minimal overhead at runtime, and this forms the basis of a practical and efficient runtime taint tracking system.

Chapter 6

Protecting Data Confidentiality in Ruby-based Web Applications

In this chapter we leverage runtime taint tracking to provide a practical solution for protecting patient data confidentiality in a cancer registry at the NHS. Our key ideas towards a system that can readily be deployed given the constraints of the cancer registry are: (1) the handling of web requests is decoupled from the processing of sensitive data for security; (2) a runtime taint tracking system ensures that data flow is accurately tracked, end-to-end, across the organisation's systems; and (3) we leverage specific dynamic features of the Ruby programming language, such as the ability to redeclare existing library methods, to propagate taint meta-data and invoke checking operations without modifications to the language interpreter. These three design choices result in a system that is easy to implement, efficient and can be deployed easily because it requires few changes to existing infrastructure.

Our system, SafeWeb [HMP+11], is a Ruby-based runtime taint tracking system for protecting against accidental data disclosure in web applications. When data leave an organisation's central database, SafeWeb ensures that they are labeled correctly and processed in an application-specific manner using event processing. Labels are used while processing web requests. Taint tracking guarantees that developer errors cannot disclose sensitive information in violation of application security policy. This reduces the need for extensive security audits of new applications. Compared to previous taint tracking systems, SafeWeb improves maintainability, performance and ease-of-use because its implementation builds on standard features of Ruby.

This chapter presents SafeWeb starting with the functional and security requirements for the cancer registry application. Section 6.2 provides a high-level overview of SafeWeb, focusing on the separation between sensitive data processing in the back-end of an organisation and the presentation of sensitive data in the front-end. These two parts of SafeWeb are described in detail in Sections 6.3 and 6.4, respectively. SafeWeb is evaluated in Section 6.5 by implementing a web application for a cancer registry in the NHS. The chapter finishes with a summary in Section 6.6.

6.1 Requirements

Many organisations in the public and private sectors collect, store and process sensitive data. Maintaining data confidentiality is crucial for such organisations. Two things must be ensured at all times: (1) the organisation's infrastructure and applications must not contain vulnerabilities that may be exploited to disclose or modify sensitive data, and (2) the data processing performed must be compliant with the organisation's security policy.

Guaranteeing data confidentiality is hard, costly and time-consuming when following current industry best practices. Organisations typically employ multiple security measures such as risk assessments, internal security code reviews and external consultations with security experts. All these measures introduce a significant overhead during development and can be seen as the primary reason for limited development of new services. For example, organisations within the NHS that handle sensitive patient data struggle to create new medical applications with a potential to improve patient care (see Section 6.1.1).

Runtime taint tracking can minimise the auditing effort required and therefore reduce the cost of developing new applications. SafeWeb's goal is to implement taint tracking within the constraints of a production environment, leverage taint tracking to guarantee data confidentiality in applications and shift the auditing effort to the taint tracking system instead. This approach minimises the need for code audit of each new application because the same taint tracking system can guarantee data confidentiality across multiple applications.

6.1.1 The case for a data quality assurance application in ECRIC

The Eastern Cancer Registry and Information Centre (ECRIC) is an NHS organisation that acts as a registry for cancer cases in eastern England. It collects data to provide a comprehensive picture of cancer incidents and cancer treatment within the NHS. ECRIC uses many data sources: Multidisciplinary Teams of doctors (MDTs) that oversee patient care, Patient Administration Systems (PAS) in hospitals, the Office of National Statistics (ONS) and pathology laboratories.

Most applications in ECRIC are built using the Ruby programming language to leverage the availability of rich application frameworks and local developer expertise. ECRIC maintains a *main database*, which stores structured data about tumors, patients and treatments. It is hosted in a secure private network isolated from the public Internet. Data are collected from the different sources, processed using domain knowledge of staff and then imported to the main database. ECRIC staff has the ability to access the main database remotely using a web application server that has undergone extensive security audit.

Discussions with staff at ECRIC identified one new application that they want to introduce: a web portal to improve the collection of data from MDTs. MDTs at hospitals send reports about cancer incidents and patient treatment to ECRIC through paper-based reports or

NHSmail[1], i.e. a government-accredited email service for the transmission of patient data. ECRIC receives and processes these reports but often identifies errors, e.g. entries that miss particular information or discrepancies between the data provided by different MDTs. Currently, ECRIC staff is involved in a laborious process of manually generating information about data quality and then disclosing it to the relevant MDTs. Instead, an *MDT portal application* would allow MDTs to see their own data after they have been processed by ECRIC, compare data quality to their peers and notify ECRIC about any changes required.

The MDT portal application should satisfy the following *functional* requirements:

F1 Members of an MDT may access the MDT portal application using a web browser, log in and view the detailed ECRIC records for the patients whom they treat. They should have the option to provide feedback, which is handled externally in compliance with the current workflow, e.g. using secure NHSmail.

F2 Members of an MDT may access various metrics that ECRIC generates from their data, e.g. how complete MDT-provided data actually are in regard to the data that they are supposed to provide to ECRIC.

F3 Members of an MDT may compare their own data with statistics generated from the data of their peers, e.g. relevant to data completeness.

ECRIC has the following two *security* requirements for each application that it deploys internally (they must also be satisfied by the MDT portal application):

S1 External users should only be able to access a fixed subset of ECRIC's confidential data that is decided statically and never be in a position to modify any of the data that the organisation stores internally.

S2 Data should be protected from inadvertent disclosure throughout the workflow of an application. Implementation errors must be contained and not result in violations of requirement *S1*.

In particular, ECRIC has the following *application-specific security requirement* for the MDT portal application:

AS1 Members of any MDT may only consult the details of the patients treated by their team. All MDTs in a given region may consult MDT-level aggregates within the same region. Every MDT may consult regional-level aggregates.

ECRIC has ruled out a standard design for the MDT portal application, in which the web application server directly accesses the main ECRIC database. First, such a design would

[1]NHSmail, `http://http://www.connectingforhealth.nhs.uk/systemsandservices/nhsmail/`, last accessed: 18/11/2012

violate the security requirement $S1$ for static, one-way access of data. If the application server is ever compromised, attackers may gain access to ECRIC's internal network and target the main database hosted there. Second, such a design would violate the security requirement $S2$: implementation errors in the MDT portal application may result in confidential data disclosure.

The application-specific requirement $AS1$ is also challenging to enforce with a traditional web application design because it is an end-to-end requirement with different provisions for different types of data. The MDT portal application fetches information from the main ECRIC database, processes it in an application-specific manner and finally presents the results to individual members of MDTs. While the propagation of regional aggregates is relatively permissive, the MDT portal application must carefully restrict access to detailed patient records. Any developer error in either data processing or access control may result in violations of security policy. ECRIC would have to perform security audits repeatedly during the application's life-cycle to prevent such errors.

If a security mechanism is to satisfy ECRIC's requirements, it should both be able to enforce end-to-end data flow guarantees and to control the propagation of different data in compliance of different policies. As this thesis demonstrates, runtime taint tracking is such a mechanism.

6.1.2 Practical runtime taint tracking for web applications

Runtime taint tracking can enforce the application-specific requirement $AS1$, independently of implementation errors in the MDT portal application. $AS1$ must first be translated into a taint tracking policy for enforcement (§2.3.1). A simple option is to associate taint meta-data that take the form of unique tags to confidential patient information and then allow only users with declassification privileges over those tags to receive the associated information (i.e. similar to confidentiality labels in DEFC). Each user of the MDT portal application should be allocated the declassification privilege (§3.2.3) for a subset of possibly different tags. If data labeling and the assignment of declassification privileges to users is correct, the application-specific requirement $AS1$ will be enforced.

A taint tracking system can also ensure that application bugs do not disclose confidential patient data (security requirement $S2$). First, the propagation of taint meta-data is transparent to the application and the invocation of checking operations occurs automatically before any data are released to external parties. Second, since application code is written by in-house developers at ECRIC, the benevolent developer assumption (§2.3.1) holds: code does not attempt to evade taint tracking actively. For these two reasons, the taint tracking system has an accurate view of the confidentiality of the data that the application processes. Therefore it can prevent inadvertent data disclosure.

The application-specific security requirement $AS1$ introduces restrictions for patient data aggregates at the MDT and regional levels. Aggregates are challenging for taint tracking

due to cross-contamination: a single component that inspects confidential data on behalf of multiple patients should generate output as confidential as all of its inputs combined. No user would be able to receive such cross-contaminated data because no single user may receive confidential data for patients treated by other MDTs.

The cross-contamination problem can be avoided with a component that is trusted to declassify patient data. Such a component would (1) receive the aggregate data and remove all tags; (2) identify a single tag that captures the confidentiality of the aggregate data and attach it to them; and (3) ensure that all users that should be able to receive the aggregate data possess the declassification privilege for the tag used. Overall, the MDT portal's security policy $AS1$ can be enforced using taint tracking with two different types of tags: MDT-level and region-level.

Introducing taint tracking, however, in the production environment of ECRIC is challenging. ECRIC prefers not to replace existing infrastructure with research prototypes because their staff do not possess the necessary resources or expertise to maintain them. For example, while unit isolation in DEFCon incurs minimal performance overhead, verifying future versions of the JDK may require familiarity with some JDK internals. Similarly, for taint tracking to be practical its impact on performance must be minimal—partial taint tracking as suggested in PHP Aspis achieves good performance but the cost of relaxing policy enforcement is not acceptable to ECRIC.

SafeWeb improves the state-of-the-art in taint tracking systems in two aspects, which enable a production deployment at ECRIC. First, SafeWeb does not require changes to the underlying Ruby interpreter. The implementation leverages Ruby's meta-programming and security primitives, resulting in a system that is easy to understand and maintain. The performance overhead of the approach is en par with taint tracking implementations in other interpreted languages that require custom interpreter modifications. Second, SafeWeb suggests a system design that combines tracking at different granularities, i.e. at variable-level and at unit-level, to track data flow across ECRIC's systems. Unit-level taint tracking ensures that the processing performed when extracting data from ECRIC's main database is monitored, i.e. right from the point when confidential data are first accessed. As explained in the next section, the generated taint meta-data are then used at the variable level by the web application to ensure that data are never disclosed in violation of security policy.

6.2 End-to-end tracking of confidential data

SafeWeb separates the processing of confidential patient data from their presentation and uses taint tracking to enforce data flow constraints end-to-end in an organisation's systems. Figure 6.1 illustrates the two parts of SafeWeb: the *event processing back-end* which hosts application-specific processing, and the *web front-end*, which exports the result of the processing to clients via a web interface. Confidential data processing in the back-end occurs in event processing units that consume and emit events. The architecture satisfies the security

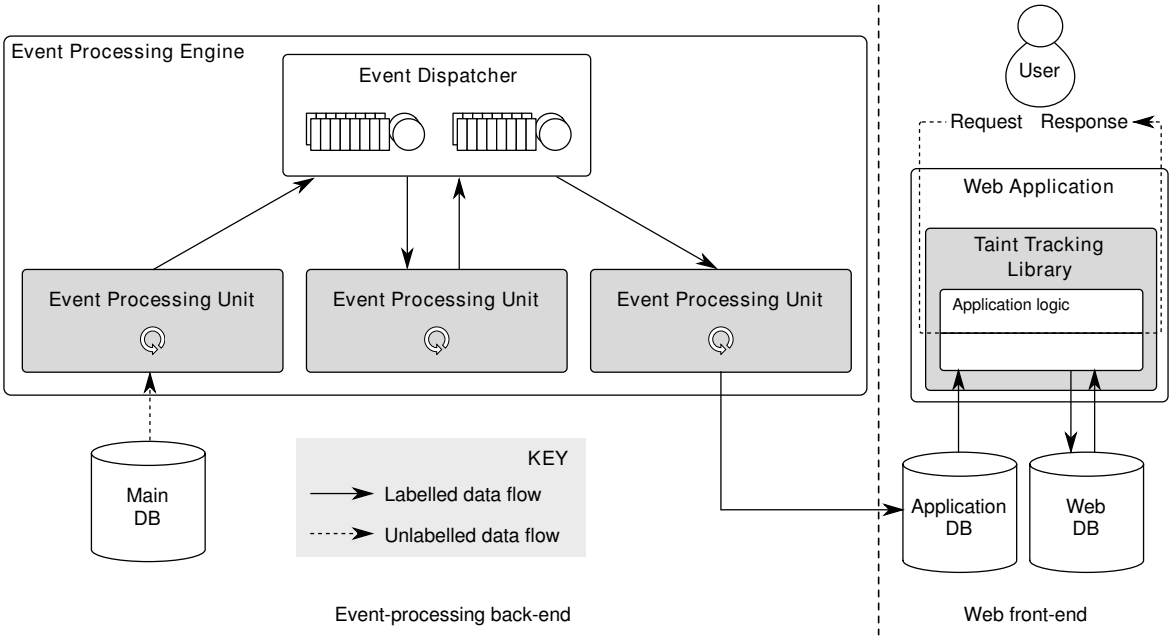

Figure 6.1: Overview of the SafeWeb architecture.

requirement $S1$ because it decouples the handling of web requests from confidential data processing [YJ10]. It ensures static and one-way access to data because a compromised web server cannot affect event processing.

The event processing back-end hosts business logic for processing confidential data stored in the main database. Its design is similar to a DEFCon engine (§3.4). *Event Processing Units* generate, process and store events according to the functional requirements of the application. Units that access ECRIC's main database (*Main DB*) are responsible for correctly labeling the events that they publish. The *Event Dispatcher* ensures that labeled events only propagate to units that are eligible to receive them. The *Event Processing Engine* acts as the runtime execution platform for units: it controls unit access to the environment to enforce isolation and invokes checking operations to inspect labels. Result events are stored together with their associated labels in an application-specific database (*Application DB*), which is subsequently exported to the web front-end.

The web front-end serves client web requests and is restricted to confidential data stored in the local application database. A second database, the *Web Database*, stores other management data that the web application requires. SafeWeb's *Taint Tracking Library* is responsible for taint meta-data propagation at the front-end. It fetches the labels stored with the data in the application database and propagates them during page generation. The security requirement $S2$ for end-to-end protection from accidental disclosure is therefore satisfied: labels are associated with data throughout processing and label checks before sending responses to clients ensure that data are never disclosed in violation of the application security policy.

Figure 6.2 uses the taint tracking model from Section 2.3.1 to illustrate the design of SafeWeb. The event processing back-end tracks taint meta-data at the granularity of individual

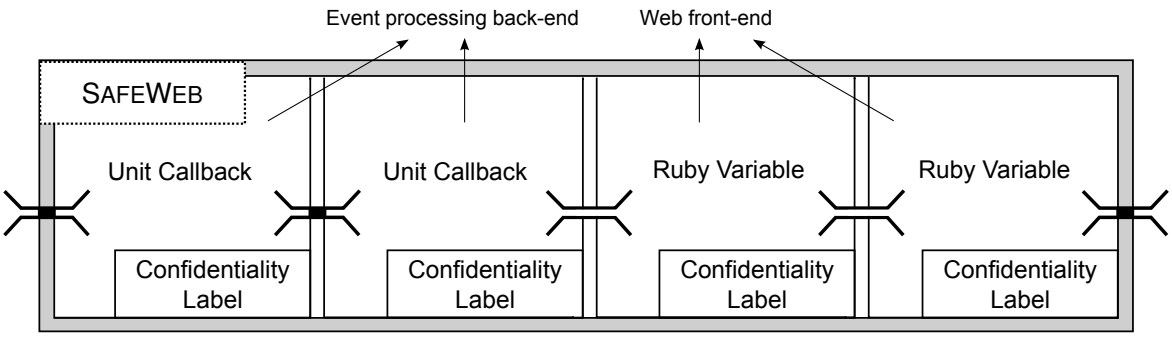

Figure 6.2: SAFEWEB as a runtime taint tracking system. SAFEWEB combines two tracking granularities, the unit- and the variable-level granularities, to satisfy the security requirements of web applications in ECRIC.

event subscription callbacks that units register. SAFEWEB uses a simplified version of the DEFC label model (§3.2) for taint meta-data focused on confidentiality only. The tracking granularity of the system changes at the web front-end in which confidentiality labels are associated with individual Ruby variables. Tracking operations at the front-end propagate confidentiality labels transparently to the application. Labels are used as input for checking operations, which are invoked when the web application generates HTTP responses.

6.3 Event processing back-end

The SAFEWEB back-end houses the event processing engine. The event processing engine supports and controls unit execution as follows: it associates confidentiality labels with events and units (§6.3.1), it dispatches events between units (§6.3.2) and it restricts unit access to the environment (§6.3.3).

6.3.1 Taint tracking policy

The design of the taint tracking policy in SAFEWEB focuses on simplicity and ease-of-use. Compared to DEFCON, the SAFEWEB engine tracks event flow using a single label per event, it only enforces confidentiality (not integrity) data flow constraints and the data flow policy enforced is assumed to be known statically (and does not change at runtime).

Events in SAFEWEB are key-value attribute pairs with an optional data payload. Both the event attributes and the data payload are untyped strings. Additionally, each event is protected with a single confidentiality label that consists of a set of tags. Tags are arbitrary strings with application-specific semantics, e.g. the URI `ecric.org.uk/mtdp/patient/56321` may be used as a tag to protect the data of a single patient.

In the MDT portal application a unit periodically inspects the main ECRIC database and identifies new patient records. It then publishes events that correspond to new records protecting each event with a label specific to an MDT (i.e. a label that contains the MDT's ID). This is always possible because adding tags to a label of an event does not require

```
1  subscribe '/mdt_update', 'type=deletion' do |event|
2    list = get 'mdt_list_' + event[:mdt_id]
3    list delete_at event[:del_index]
4    set 'mdt_list_'+ event[:mdt_id], list
5  end
6  subscribe '/update_mdt_portal' do |event|
7    event[:mdt_total] times do | mdt_id |
8      list = get 'mdt_list_' + mdt_id
9      publish '/mdt_report', list, :remove => $LABEL,
10       :add => ['label:conf:ecric.org.uk/mdt/' + mdt_id]
11   end
12 end
```

Listing 6.1: An example of a unit in SafeWeb.

additional privileges. Aggregates calculated from confidential data of multiple MDTs are labeled with region-specific labels following the process described in Section 6.1.2.

Unit access to events is controlled by allocating clearance and declassification privileges. As in DEFCon (§3.2.3), the clearance privilege over a tag allows a unit to receive an event protected by the tag but the tag propagates to the unit's output. The declassification privilege enables the unit to remove the tag from any subsequent events that it publishes. Privileges are assigned to units statically using a policy configuration file.

SafeWeb does not support data flow policies that change dynamically at runtime. Units cannot allocate new tags or delegate privileges over existing tags. Unit developers reference tags directly, i.e. tags can be communicated freely and knowledge that a specific tag is being used does not convey any useful information. This design decision limits the generality of SafeWeb but results in a system that is simple for developers to use.

6.3.2 Event dispatch

The *Event Dispatcher* enables units to communicate by exchanging events. It supports topic-based publish/subscribe communication, i.e. each event is matched to subscriptions on a given topic, with optional content filtering on event attributes within a topic [EFGK03]. Events are filtered according to their attributes and only delivered to units if labels permit communication. SafeWeb's checking operations inside the dispatcher compare the label of the event to the clearance privileges of the subscriber unit and only permit dispatch if the tags in the event label are a subset of the tags that the unit has clearance for (i.e. according to the "can-flow-to" relation).

Listing 6.1 shows part of a unit that is responsible for updating the MDT portal application. The unit issues two subscriptions, one to receive notice when a patient record has been deleted (`mdt_update` events in line 1) and one for events that trigger an update of the application database at the front-end (`update_mdt_portal` events in line 6). Each subscription is associated with a callback that is invoked when an event matches the subscription

(lines 2 to 4 and 7 to 12). The event dispatcher records the unit's clearance privileges when registering a subscription for use in checking operations during event dispatch.

Units may store state that persists between event deliveries in a unit-specific key-value store. The key-value store associates labels with its keys and allows different callbacks to share state in a secure fashion (explained next in Section 6.3.3). For example in Listing 6.1, the key-value store is used by the first callback to keep track of deleted patient files (lines 2 and 4) and by the second callback to publish cumulative results per MDT (lines 8 and 10).

The event dispatcher relies on the *Streaming Text Oriented Message Protocol (STOMP)*[2], which it extends for transferring additional meta-data. STOMP is a simple yet extensible message protocol similar to HTTP with existing Ruby bindings[3]. Its requests consist of a command field (e.g. `CONNECT`, `SEND` or `SUBSCRIBE`), optional headers and a body. The `destination` header is used for the topic of events published while additional SafeWeb-specific headers store labels. An optional header stores the subscription filter that is specified by units as an SQL-92 selector.

The SafeWeb event dispatcher implementation builds upon the Ruby open-source server implementation of STOMP[2], in which it adds support for SSL at the transport layer. While matching events to subscriptions, the dispatcher calculates the "can-flow-to" relation and rejects messages, which recipients are not eligible to receive. The STOMP client library used by units to communicate with the event dispatcher is implemented using the EventMachine I/O library[4].

6.3.3 Controlling unit execution

The SafeWeb engine effectively enforces confidentiality constraints by maintaining taint meta-data about the confidentiality of the data that unit callbacks are allowed to receive (callback label tracking) and by monitoring all communication channels available to units (callback isolation).

Label tracking

The engine monitors data flow by associating a single label with the execution of a subscription callback. Before an event is delivered to a subscription callback for processing, the callback's label is initialised to the label of the incoming event. The current callback label is accessible to unit code though the variable `$LABEL`. When a new event is published by the callback, the value of `$LABEL` is assigned to a new label used to protect the confidentiality of the published event. A unit can change the label of the events that it publishes by specifying tags to add or remove in `publish` calls (Listing 6.1; lines 9 and 10)—the latter is only possible if the unit has the declassification privilege for each tag removed.

[2]STOMP Protocol, `http://stomp.github.com`, last accessed: 20/9/2012
[3]StompServer, `http://stompserver.rubyforge.org/`, last accessed: 20/9/2012
[4]EventMachine, `http://rubyeventmachine.com/`, last accessed: 20/9/2012

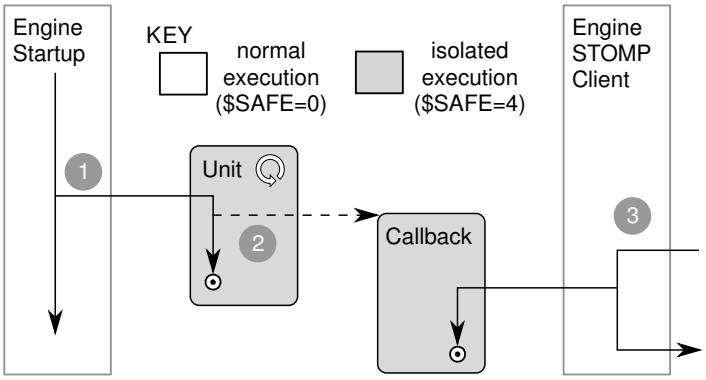

Figure 6.3: Isolation of units and callbacks performed by the event processing engine.

For values stored in the key-value store, the engine maintains labels per key. When a unit callback reads a value from the store, `$LABEL` is updated to reflect the confidentiality of the data that the unit receives. For this, the tags stored in the value's key are added to `$LABEL`. Similarly when a unit stores a value in the store, the current value of `$LABEL` becomes the key's label (except for any tags added or removed, as in `publish` calls).

Note that such implicit tainting of unit state upon receiving events or when reading from the key-value store is acceptable because SafeWeb only prevents accidental data disclosure. SafeWeb does not protect data confidentiality for cases, in which units maliciously attempt to disclose information (i.e. in contrast to DEFCon; see §3.1.2).

Isolation

The SafeWeb engine must isolate unit callbacks to ensure that they cannot use I/O channels or variables outside the local scope, exchanging data in violation of the application security policy (i.e. through global variables, instance variables and local variables of enclosing scopes).

SafeWeb leverages Ruby's *safe levels* to avoid unit access to I/O functions and to prevent communication via variables outside of the local scope. The variable `$SAFE` is a global variable that may have a different value in each thread. Setting `$SAFE` to a value between 1 and 4 activates a hard-coded taint tracking policy that the Ruby interpreter enforces (§2.3.3). When `$SAFE` is set to 4, the following restrictions apply: the thread cannot perform I/O operations, every object that the thread creates is flagged using a single "taint" bit and the thread can only modify objects that are not marked as tainted[5] (e.g. it cannot modify global variables).

Figure 6.3 shows how unit callbacks are isolated using the Ruby safe level 4. Before the engine initiates a unit, it launches a new thread and sets `$SAFE=4` (step 1). The new thread invokes unit code that initialises unit variables and registers subscription callbacks (step 2). Having unit threads start with `$SAFE=4` guarantees that unit initialisation routines cannot

[5]For another example of a less restrictive taint tracking policy available, see the Ruby taint mode (i.e. `$SAFE` level 1) presented in Section 2.3.3.

circumvent isolation: at safe level 4, a thread may not write to untainted global variables and thus it has no way to share references that point to mutable objects with other unit threads (which are also created with `$SAFE=4`). When a new event is received from the event dispatcher through the engine's STOMP client, a new thread is instantiated (again at `$SAFE=4`) and executes the callback (step 3). Callbacks may only store data in the key-value store, in which the engine maintains labels.

Despite safe level 4, callbacks of the same unit can exchange data via shared variables in their enclosing scopes. Such variables are, for example, those declared by unit initialisation code. Due to `$SAFE=4`, the variables created by unit intialisation code are tainted, and Ruby does not prevent modifications from other threads at `$SAFE=4`. To prevent callbacks from using such shared and mutable variables, the engine creates a separate variable instance per callback. New variable instances are created transparently to unit code when a subscription callback is activated. The implementation leverages the programmatic access to variable scopes in the Rubinius interpreter[6].

Safe level 4 prevents unit access to I/O channels but is overly restrictive for *privileged* units that import and export data between the event processing engine and external databases (see Figure 6.1). The engine allows units with dual clearance and declassification privileges for a set of tags to operate at `$SAFE=0` and perform I/O. It only restricts event processing for such privileged units as follows: if a unit does not possess the clearance privilege for a tag, the event dispatcher does not deliver to the unit events protected by that tag.

6.4 Web front-end

The SafeWeb front-end displays the result of event processing to remote users over the web. It requires no modifications to web application source code: developers only have to import SafeWeb's taint tracking library to prevent accidental data disclosure. SafeWeb's taint tracking library propagates the labels generated at the back-end to capture the confidentiality of the data in the front-end. When the web application attempts to return a response to a client, the library inspects the confidentiality of the data stored in the response and aborts the operation if the application security policy would be violated.

At the front-end, SafeWeb associates labels with individual Ruby variables. As an example, if a variable *ctype* stores the type of cancer for an individual patient, the variable also carries a label with tags that protect the confidentiality of that particular patient's data.

SafeWeb switches to tracking taint meta-data at the granularity of Ruby variables in order to protect typical database-backed web applications. Typical database-backed web applications receive HTTP requests from a client, fetch data from a local database and return the data in HTTP responses. Web applications handle requests on behalf of multiple users, and they need access to the confidential data of *every* user of a particular application, i.e. access

[6]Rubinius, `http://rubini.us/`, last accessed: 21/9/2012

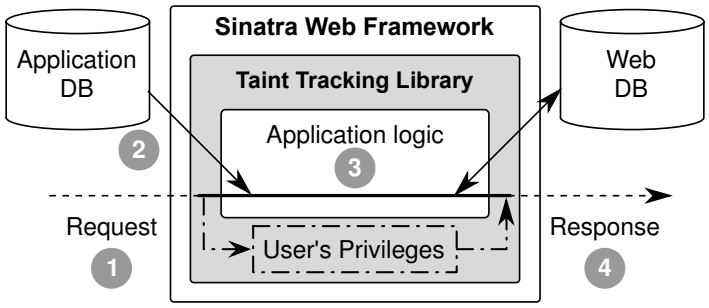

Figure 6.4: Variable-level taint tracking granularity in the SafeWeb web front-end.

to *all* confidential data stored in the database. Such a non-compartmentalised design clearly violates the security requirement $S2$ for preventing accidental disclosure of confidential data in the presence of implementation errors. The reason is that without variable-level taint tracking, once a web application is given the privilege to communicate confidential data on behalf of multiple users, the application could disclose private data of one user to another.

SafeWeb invokes checking operations to inspect the confidentiality of the data in HTTP responses before the responses are sent to clients. The checking operations ensure that the set of tags in the response's confidentiality label is a subset of the tags that the user is privileged to receive—otherwise the response is suppressed. This is sufficient to prevent accidental data disclosure, assuming that the propagation of taint meta-data inside the web application is precise. No further modifications to the web application's architecture are required.

Figure 6.4 illustrates how a web request is served by the front-end when the SafeWeb taint tracking library is active. In step 1, a new HTTP request arrives. Control is given to the taint tracking library, which authenticates the user and identifies their privileges. In step 2, the web application fetches confidential data from the application database. SafeWeb's taint tracking library intercepts this operation. It inspects the data retrieved and ensures that the corresponding labels (as set in the application database) are attached to the Ruby object returned to the application. The application then continues with data processing in step 3 to generate a response.

The SafeWeb taint tracking library modifies the semantics of Ruby statements and library functions to propagate taint meta-data. The rules for meta-data propagation are simple: if the result of a statement or function call contains data from its input, all tags in the labels of the input are attached to the label of the output. In step 4, the taint tracking library inspects the label of the response—unless the label contains a subset of the tags that the user is privileged to receive the operation is aborted.

The SafeWeb taint tracking library redefines the `String` class and subclasses of `Numeric` so that: (1) each instance stores a SafeWeb label and (2) each method propagates labels according to the method's semantics. As an example of label propagation, consider string concatenation. When two strings are concatenated, the result string should be as confidential as the original two strings. To achieve this, the SafeWeb taint tracking library declares a new method `nconcat()` in the `String` class. It aliases the existing concatenation method `'+'`

to call `nconcat()`, which, in addition to string concatenation, also propagates the labels of the input to the result. All subsequent invocations of `'+'` correctly propagate taint meta-data. Assuming non-malicious developers, who do not attempt to circumvent this mechanism at runtime (§6.1.2), this change effects correct taint meta-data propagation for string concatenation. Similar redefinitions are introduced for all `String` methods.

The implementation of meta-data propagation in the SafeWeb taint tracking library outlined above leverages the pure object-oriented foundations and standard meta-programming features of Ruby. All types in Ruby are classes (there are no primitive types), all language operators are implemented as class methods and all class method definitions can change at runtime. This flexibility (compared to languages such as Java and PHP) eliminates the need for changes to the execution platform and results in a practical taint tracking implementation that is easy to maintain.

SafeWeb requires that web applications are written using the Sinatra web framework[7] and execute on the Rubinius interpreter. Sinatra is a simple web framework for rapid web application development in Ruby. It is used by SafeWeb because it offers well-defined interception points for incoming HTTP requests and outgoing responses. The taint tracking library intercepts execution and attaches code to either fetch user privileges or invoke checking operations. The Rubinius interpreter allows the SafeWeb taint tracking library to manipulate the regular expression variables (`$~`, `$1` etc). These are special variables set in the calling scope when a regular expression function is invoked. With Rubinius, the SafeWeb taint tracking library can modify the regular expression variables and ensure their correct labeling.

SafeWeb does not introduce additional functionality to protect web applications from code injection attacks such as XSS or SQL Injection. This is considered an orthogonal problem, and Ruby already offers some support for it. As presented in Section 2.3.3, when `$SAFE` is set to 1, Ruby activates a simple taint tracking policy that can be used as the basis of a variable-level taint tracking system. Such a system would intercept functions that perform sensitive operations to check the value of the `taint` bit before proceeding.

6.5 Evaluation

The experimental evaluation of SafeWeb has two goals: to explore the system's effectiveness in preventing inadvertent disclosure of confidential data and to measure the performance overhead while doing so. The security properties of SafeWeb and its impact on performance are evaluated with a prototype of the MDT portal application.

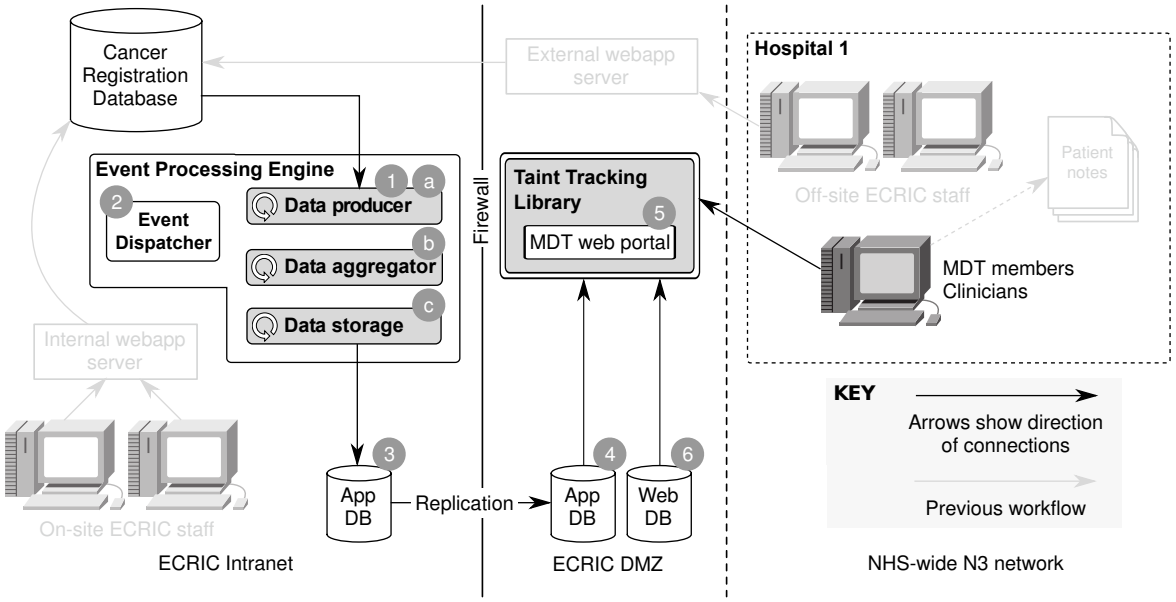

Figure 6.5: Deployment of the MDT portal application using SafeWeb at ECRIC.

6.5.1 Implementation of the MDT portal application

Figure 6.5 shows the MDT portal application and its deployment using SafeWeb at ECRIC. The MDT portal application consists of three units in the event processing back-end (1a–1c) and a Sinatra-based web application (*MDT web portal*) in the front-end (5). These are deployed into the two separate zones of ECRIC's network: the *Intranet* and the *demilitarised zone* (*DMZ*). A firewall separates the Intranet from the DMZ permitting only unidirectional connections from the Intranet to the DMZ. Critical infrastructure such as the main database is only accessible from within the Intranet. MDTs may access the web front-end in the DMZ using the HTTP protocol and the NHS-wide *N3 network* that connects NHS organisations[8].

The MDT portal application is implemented using three units in the back-end:

1. The *data producer* accesses the main ECRIC database for input. It reads patient data from multiple tables using ECRIC's framework for data access, labels the data according to the MDT that is responsible for each patient and finally publishes the result as labeled events using the event dispatcher. The data producer only uses MDT-level tags (not unique tags per individual patient) because they are sufficient to satisfy the application-specific requirement *AS1*.

2. The *data aggregator* continuously listens for events published locally that require additional processing before they are sent to the MDT web portal. It receives events on behalf of multiple patients and calculates MDT-level and region-level aggregates to satisfy the functional requirements *F1–F3*. The events that the data aggregator generates are labeled correctly despite implementation errors because the data aggregator

[7]Sinatra, `http://www.sinatrarb.com/`, last accessed: 20/9/2012
[8]N3, `http://n3.nhs.uk/`, last accessed: 20/9/2012

executes without declassification privileges over the tags used for individual MDTs.

3. The *data storage* unit is responsible for data persistence. It ensures that labels are correctly associated with the data that they protect inside the application database. The data storage unit executes with clearance and declassification privileges for each tag used in the MDT portal application.

The MDT web portal in the front-end is implemented as a Sinatra application that imports SafeWeb's taint tracking library. As described in Section 6.4, SafeWeb's taint tracking library authenticates users transparently to the web application. In particular, it adds hooks to the MDT web portal, which intercept every incoming HTTP request. The current prototype of the SafeWeb taint tracking library uses HTTP Basic Authentication and relies on Transport Layer Security (TLS) for the secure transmission of user credentials. There are plans at ECRIC to add support for authentication using smartcards.

The different components of the SafeWeb prototype and the MDT portal application are deployed within ECRIC as follows. The event processing engine, denoted with (1) in Figure 6.5, provides a shared execution environment in the back-end for event processing on behalf of multiple applications. A single event dispatcher (2) dispatches events between units using the "can-flow-to" relation to compare labels. The units of the MDT portal application in particular (1a–1c) use a database to store labeled output (3). ECRIC's firewall only allows unidirectional connections from the Intranet to the DMZ. Therefore two instances of the application database exist: one inside the Intranet (3) and one in the DMZ (4). The Intranet instance is transmitted periodically to the DMZ using push replication. The MDT portal in the front-end (5) does not have write access to the DMZ instance. This prevents a compromised web application from changing how data are exported, as per security requirement *S1*. The web database (6) stores data specific to the MDT web portal (such as session data).

6.5.2 Security properties

The MDT portal application is used as a case study to explore SafeWeb's effectiveness in preventing information disclosure. Specific instances of well-known errors are introduced in the MDT portal. SafeWeb should detect these errors and prevent information disclosure.

The errors introduced to the MDT portal application are representative of typical developer errors. We inspect the CVE database, studying previous occurrences of vulnerabilities classified as "Information Disclosure", "Access Control" or "Design Error". We group such vulnerabilities according to their underlying cause. We then model the errors that we introduce after these groups. The following paragraphs report whether SafeWeb manages to prevent information disclosure for each type of error.

Omitted access control checks. A common developer error that leads to information disclosure is an omitted access control check (e.g. CVE reports 2011-0701, 2010-2353 and

```
1 require 'sinatra'
2 require 'safeweb−tracking'
3 get '/records/:mid' do
4   content_type :json
5   return nil if !check_privileges(params[:mid])
6   r = Records.by_mid(:key => params[:mid])
7   process r
8   r.to_json
9 end
```

Listing 6.2: An example of a rule in the web front-end of the MDT portal application.

2010-0752). Listing 6.2 shows a method in the MDT web portal that returns patient records to an MDT after an access control check (line 5) has succeeded. To emulate the effect of an omitted access control check in an application executing without SafeWeb, lines 2 (which enables the SafeWeb taint tracking library) and 5 (which invokes the access control check) are commented out. After this change and when a user requests the record of a patient that is treated by another MDT, confidential patient data are disclosed in line 8 in violation of application security policy. If the SafeWeb taint tracking library is enabled (line 2), it results in a JSON string (line 8) labeled with tags specific to the MDT that treats the patient. The checking operation invoked by the taint tracking library inspects the result's label, detects the problem and prevents information disclosure by displaying an error message to the user.

Errors in access checks. Even when an access control check is invoked, there is no guarantee that it enforces the application security policy correctly and thus prevents information disclosure (e.g. CVE reports 2011-0449, 2010-3092 and 2010-4403). Errors inside access control checks are hard to discover during development because they only manifest with specifically-crafted input. To emulate such problems, the access control check presented in Listing 6.3 is modified to ignore the case of the username (line 4). With this change, a user may get assigned the privileges of another MDT if their usernames only differ in the case of some characters. To test the setup, two new accounts were created with usernames `mdt1` and `MDT1`, with different privileges. User `MDT1` is able to access patient records that only user `mdt1` should be able to receive. With SafeWeb's taint tracking library enabled, the patient records are labeled correctly and data disclosure is prevented.

Inappropriate access checks. Developers who are asked to implement complex security policies may not share the same understanding of the requirements. This category of application vulnerabilities covers cases, in which access control checks are invoked consistently yet the policy that they enforce is not the one expected (e.g. CVE reports 2010-4775 and 2009-2431). Such scenarios are emulated by removing the check in the `check_privileges` method that compares the clinic the user is member of with the

```
1 def check_privileges id
2   m = Measurement.find(id)
3   @is_admin or Privileges.count( :conditions => {
4     :u_id => User.find_by_name(@username).id,
5     :hospital => m.hospital_id,
6     :clinic => m.type
7   }) > 0
8 end
```

Listing 6.3: An access control check used by the MDT portal application.

type of the stored data (Listing 6.3; line 6). With SafeWeb's taint tracking library disabled, the missing condition allows a user to see patient records of other MDTs at different clinics of the same hospital. Again, such inadvertent information disclosure is prevented with taint tracking because the labels of the result records capture the true confidentiality of the data.

Design errors. The last group of vulnerabilities are due to design errors in the business logic of an application (e.g. CVE reports 2011-0899 and 2010-3933). In such cases, an application processes data in an unexpected way, and while doing so, it discloses confidential information. No action is typically taken to prevent confidential information disclosure because developers did not anticipate the particular use case. To emulate such design errors, the data aggregator unit from Figure 6.5 is modified to ignore the hospital of origin when calculating aggregates on behalf of a specific MDT user named `mdt1`. The result aggregates for user `mdt1` thus mix patient records from multiple different MDTs. Without SafeWeb's end-to-end label propagation from the back-end to the front-end, the MDT web portal discloses the aggregate records to `mdt1`. With SafeWeb enabled, the aggregate records are cross-contaminated with multiple MDT tags, one per MDT whose data are included in each record. Confidential data disclosure is prevented because no user in the front-end possesses the necessary privileges to access cross-contaminated aggregate records.

SafeWeb enforces data confidentiality but it still requires trust in the SafeWeb infrastructure and in important components of the MDT portal application. First, the taint tracking library in the front-end must correctly propagate labels, authenticate users, fetch user privileges and invoke checking operations before an HTTP response is returned. Second, the event processing back-end must isolate unit callbacks, keep track of their current label and privileges, propagate labels in the key-value store and invoke checking operations while dispatching events. Third, the two privileged units in the back-end, i.e. the data producer and the data storage unit, must correctly assign labels to the events that they publish and correctly store these labels in the application database. Finally, the policy file that assigns privileges to units (in the back-end) and to users (in the front-end) must not contain errors.

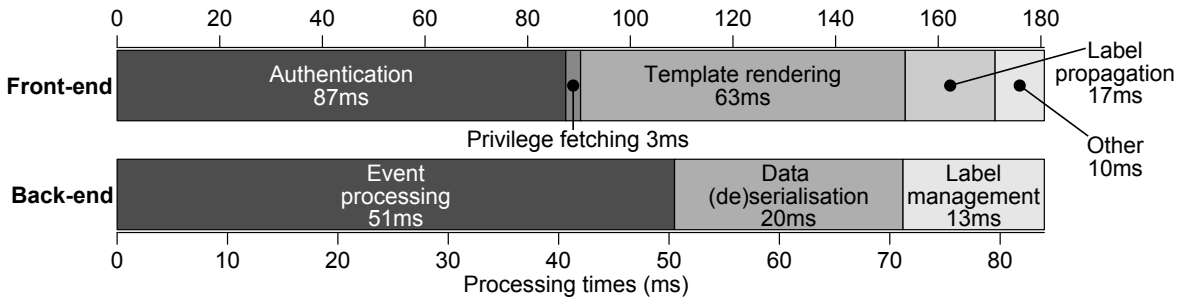

Figure 6.6: Processing latency within the MDT portal application [HMP+11].

SafeWeb therefore does not eliminate the need for a code audit. Instead, it greatly reduces the effort required because the audit can focus on a few components, which can be reused in future applications. The current implementation of SafeWeb consists of 1943 LOC in the taint tracking library and 1908 LOC in the event processing engine. After this code has been audited, the data confidentiality in each additional application only depends on a small trusted part that controls labels and privileges. In the MDT portal application, for example, such code is found only in the two privileged units in the back-end (138 LOC). The remaining 2841 LOC of the MDT portal application need not to be audited because the confidentiality of patient data does not depend on their correctness.

6.5.3 Performance overhead

To measure the performance overhead of SafeWeb, the metrics of processing latency and event throughput are used (§3.5). The measurements are taken on an AMD Opteron 6136 2.4GHz machine with 16 GiB of RAM running Ubuntu 10.04 and Rubinius 1.2.3. The 95% confidence interval for each of the measurements reported lies at most 5% of the value on each side.

Processing latency in the front-end is measured as the page generation time of the MDT portal's main page. We issue 10.000 requests, with SafeWeb's taint tracking library enabled or disabled. Due to label propagation and checking operations, the taint tracking library increases the page generation time by 14%, from 158 ms to 180 ms.

Processing latency in the back-end is measured as the time between the publication of an event at the data producer and the reception of the corresponding event at the data storage unit. Due to the additional operations required to propagate labels, invoke checking operations and enforce isolation between unit callbacks, the latency to process a single event increases by 15%, from 73 ms to 84 ms.

Figure 6.6 illustrates the impact of various operations on processing latency when SafeWeb is enabled. In the front-end, HTTP basic authentication and the transformation of the MDT web portal's `ERB`-based templates[9] to HTML are the most time-consuming operations, taking

[9]ERB–Ruby Templating, `http://ruby-doc.org/stdlib-1.9.3/libdoc/erb/rdoc/ERB.html`, last accessed: 20/9/2012

87 ms and 63 ms to complete, respectively. Fetching the privileges of the current user takes 3 ms while label propagation adds 17 ms. "Other" includes access to the databases and the rest of the processing to generate a response; together they account for the remaining page generation time. In the back-end, event processing and data serialisation when units communicate with the event dispatcher are the most time-consuming operations, taking 51 ms and 20 ms to complete, respectively. Label management, which includes label serialisation and checking operations at the dispatcher, adds another 13 ms.

The event throughput performance of the event processing engine exceeds the relatively modest requirements of the MDT portal application. Therefore, the maximum achievable throughput is measured with a simple microbenchmark of two units: an event producer that sends events continuously to an event consumer. Throughput is sampled at a frequency of one measurement per second for 1000 seconds. Without callback isolation in the event processing engine and without label checks at the event dispatcher, the observed throughput is 4455 events/second. It decreases by 17% to reach 3817 events/second when isolation and label tracking are activated. During the experiment, memory consumption remains stable, which indicates that the rate achieved is not due to queuing of events in the system.

Overall the performance overhead of SafeWeb is small and comparable to approaches that implement taint tracking by modifying the language interpreter [YWZK09]. In practice, the impact of SafeWeb on processing latency and event throughput is acceptable, considering the security requirements of the MDT portal application.

6.6 Summary

Organisations in the public and private domains that handle sensitive user data often find it challenging to deploy new applications accessible over the Internet. In this chapter, we presented SafeWeb, a runtime taint tracking system for creating secure web applications in Ruby. It enforces end-to-end confidentiality guarantees and minimises the need for code reviews, security consultations and risk assessment, while integrating with existing web development practices.

The strict data security requirements of an organisation within the UK National Health Service (NHS) provided us with a set of real-world design constraints. The sensitivity of healthcare data required careful consideration of the parts of the taint tracking system that push and pull data. The back-end requirements suited an event processing system, whereas the front-end is a typical web application.

The event processing back-end of SafeWeb is deployed within the organisation's Intranet and performs taint tracking at the granularity of event subscription callbacks in units. Units publish confidential data as events labeled according to a security policy. Units specific to each web application may receive and process these events, yet they are constrained by their privileges. Labels propagate to result data and are stored inside an application-specific database.

The web front-end is isolated at the network level from the back-end. It performs taint tracking at the granularity of Ruby variables in a web application. SafeWeb's taint tracking library intercepts requests and associates privileges with the user that issued each request. It fetches the labels stored in the application database along with the data and propagates them transparently. The labels are used to abort responses that would disclose confidential data in violation of the application's security policy.

In contrast to other taint tracking systems, SafeWeb is implemented by taking advantage of standard meta-programming features of the Ruby language and the Rubinius runtime. This simplifies the maintenance of the taint tracking system because it does not require expert knowledge of the internals of the Ruby interpreter or extensive transformations to application source code.

SafeWeb was evaluated by implementing a web application for assisting Multidisciplinary Teams (MDTs) in hospitals. SafeWeb significantly reduces the part of the application's codebase that must be audited for security, while it effectively prevents many types of well-know vulnerabilities from disclosing confidential data.

Chapter 7

Conclusion

An important problem in application security is the lack of practical methods to enforce security policy that controls the propagation of sensitive data. Once data have been communicated to an application, the application is free to process the data and use them in arbitrary operations. As such, any developer error inside the application may lead to the disclosure of confidential data or introduce application-level vulnerabilities.

Runtime taint tracking is a method for controlling data propagation. It intercepts the operations of an application and associates meta-data with the data that it processes. By studying previous runtime taint tracking systems suggested in the literature, we identified common design properties and limitations. We observed that fundamental operations in a taint tracking system, i.e. the isolation of application components and the interception of an application's operations to propagate taint meta-data, are challenging to achieve in practice and reduce performance at runtime. Past research has tackled these problems mainly by suggesting novel operating systems and modified interpreters. These demonstrated the applicability of taint tracking in various scenarios but render taint tracking hard to adopt.

This thesis focused on the practicality and efficiency of runtime taint tracking. We considered three important requirements for a taint tracking system: minimal performance overhead at runtime, the ability to configure the data flow policies that are being enforced and an implementation that eliminates the need to modify existing infrastructure.

Our key observation to satisfy these requirements was to only support a particular class of applications each time. We considered three different types of applications: event processing applications written in Java, web applications written in PHP and Ruby web applications deployed at an organisation in the NHS. Focusing on a particular type of application allowed us to leverage unique language features and make certain assumptions that facilitate the operation of the taint tracking system.

DEFCon, our event processing system, exposed a subset of the Java Developement Kit (JDK) class library to applications and employed bytecode transformation techniques to weave runtime checks at specific code paths. This allowed DEFCon to isolate application components effectively without extensive changes to the Java Virtual Machine or to JDK classes. To min-

imise the overhead of runtime taint tracking while controlling the propagation of events we suggested DEFC, a taint tracking policy that uses labels as taint meta-data. DEFCon also enforces policies specified in the DEFCon Policy Language (DPL). DPL enables high-level specification of data flow policy and efficient enforcement via translation to DEFC.

PHP Aspis, our tool for protecting PHP web applications from injection attacks, introduced partial taint tracking, an attempt to propagate taint meta-data only in the application's most vulnerable parts. With partial taint tracking, the runtime overhead of updating taint meta-data is reduced to an acceptable level while still preventing many real-world injection attacks. PHP Aspis performs taint tracking by transforming web applications at the source code level both statically and at runtime (i.e. when code is generated dynamically). This eliminates the need for a proprietary PHP interpreter. Finally, taint categories provide a way to adapt the taint tracking policy to each web application and to avoid sanitising data that applications consider secure.

SafeWeb, our taint tracking system for guaranteeing data confidentiality in Ruby web applications, demonstrated the applicability of Ruby for implementing runtime taint tracking. Specific features of Ruby, i.e. safe levels and redefinitions of library methods using meta-programming, enable isolation and invocations of tracking and checking operations without modifying the language interpreter. The resulting taint tracking system is easy to maintain because it is itself implemented using standard features of Ruby. SafeWeb incurs minimal performance overhead. It facilitates the development of new applications in an NHS organisation because it minimises the need for security audits.

Overall, this thesis has demonstrated that an efficient and practical implementation of a runtime taint tracking system is possible and can readily benefit applications in the web and in event processing.

7.1 Thesis summary

We begun this thesis with an overview of security challenges in two domains, the web and event processing. We observed that applications there would benefit from a method to enforce data flow policies that restrict their processing. In web applications, the lack of a practical method to enforce data flow results in injection vulnerabilities. In event processing, application scenarios that have the potential to benefit multiple users (e.g. low-latency trading via co-location with the stock exchange) are considered insecure without the ability to guarantee how sensitive data flow in the application.

Past research on data flow enforcement followed either a static or a dynamic approach. We presented three static methods: taint analysis, symbolic execution and security-typed languages. While static methods do not incur runtime overhead and enforce data flow even if the application code actively tries to evade the analysis, they typically suffer from false positives and support for dynamic features of most programming languages is limited.

Runtime taint tracking controls data flow at runtime by associating taint meta-data with the data that an application processes. We presented a model that identifies the common design choices available to taint tracking systems: (1) the tracking granularity; (2) the type of taint meta-data; and (3+4) the tracking and checking operations required to propagate and check taint meta-data at runtime. We used this model to present various taint tracking systems and argued that, despite the different design choices explored, many systems make assumptions that render their adoption impractical. For example, systems such as Asbestos [EKV+05] and HiStar [ZBWKM06] assume fundamentally redesigned operating system kernels to support data flow enforcement in arbitrary applications. Instead, we suggested to focus on specific types of applications, minimising any modifications to the underlying execution platform.

DEFCon was the first taint tracking system presented in this thesis. DEFCon supports the execution of multi-domain event processing applications and provides data flow guarantees for exchanged events. To minimise the runtime overhead of the approach, the system uses DEFC, a taint tracking policy with label-based taint meta-data attached to events. DEFC supports partial event processing, dynamic privilege propagation and the enforcement of specific processing topologies. Event processing units execute in the same address space for efficiency. We achieved isolation between them by prohibiting access to certain Java features and by identifying dangerous code paths in the JDK where runtime checks are invoked. The performance overhead of DEFCon is small and we showed that DEFCon is capable of supporting demanding event processing applications such as low-latency trading.

DEFCon supports DPL, a policy language to decouple data flow policy specification from unit implementation and to avoid the complexity of specifying data flow policy by manipulating DEFC labels. DPL policies consist of event flow constraints that isolate events with different security requirements. These constraints involve processing contexts, i.e. hierarchical names that map to specific units at runtime and enable administrators in different organisations to collaboratively specify data flow policy. DEFCon was extended to check DPL policies for consistency and to translate them to DEFC for enforcement. We evaluated DPL with an example workflow for cancer reports in the NHS that demonstrates how DPL facilitates specification of data flow policy.

PHP Aspis was the second taint tracking system presented in this thesis. PHP Aspis prevents injection attacks in web applications written in PHP by applying partial taint tracking, i.e. tracking taint meta-data only inside part of a web application. We motivated this design with the observation that injection vulnerabilities often occur in application plugins that handle user-generated data. Therefore, by performing taint tracking only in a small part of an application, the performance overhead at runtime can be reduced significantly. This is of particular importance for PHP Aspis because the system uses source code transformations to avoid modifications to the PHP interpreter—these transformations reduce performance significantly. PHP Aspis transforms both tracking and non tracking application source code and it monitors web applications at runtime to support dynamic features of PHP. We evaluated PHP Aspis with Wordpress plugins and we showed that PHP Aspis protects the

application in presence many injection vulnerabilities previously discovered in plugins.

The final taint tracking system presented in this thesis was SafeWeb. SafeWeb provides confidentiality guarantees for web applications that handle sensitive data such as those found in healthcare. Based on discussions with NHS staff at ECRIC, we identified a set of data flow requirements for patient data and we designed SafeWeb to satisfy them. In the back-end of an organisation, SafeWeb enables web applications to export data for display in the front-end. Taint tracking at the back-end ensures that exported data are labeled correctly. In the web front-end, SafeWeb propagates labels on each application statement and uses these labels to prevent unauthorised data disclosure due to developer errors. We implemented SafeWeb in Ruby and we noticed that Ruby's safe levels and meta-programming features simplify significantly the implementation of a taint tracking system. SafeWeb demonstrates that taint tracking is a viable alternative to security audits because it effectively prevents patient data disclosure in various scenarios.

7.2 Lessons learned

The design and implementation of DEFCon, PHP Aspis and SafeWeb enabled us to identify important lessons for building practical and efficient taint tracking systems. These lessons can benefit future security and systems researchers that wish to build upon this work.

Method interception is important. The single most important property that an execution platform should have in order to facilitate the implementation of a runtime taint tracking system is the ability to intercept method invocations at runtime efficiently. This should cover both invocations of methods provided by the platform and invocations of methods provided by applications. The existence of a flexible interception platform for Java (§3.3.3) enabled our practical isolation strategy for threads that permits efficient sharing of data. In Ruby, meta-programming features simplified the implementation of a practical taint tracking system compared to our solution for PHP, which transforms source code.

High-level languages facilitate taint tracking. High level-languages that rely on an execution platform (e.g. an interpreter) simplify the implementation of a taint tracking system. They facilitate method interception and provide libraries with well-defined semantics.

It easy to implement method interception for most high-level languages. Ruby supports it out of the box, and numerous projects offer AOP support for Java applications. Even for languages that do not support method interception such as PHP, it is possible to implement method interception by extending the language interpreter[1]. The argument against supporting method interception at runtime, i.e. reduced performance due to another level of indirection, is less important in high-level languages because most execution platforms

[1]PHP-AOP extension, `http://code.google.com/p/php-aop/`, last accessed: 19/2/2013

already invoke methods indirectly using pointers for other reasons. In PHP, for example, the interpreter does not allow applications to intercept method calls (i.e. without third-party extensions) although the interpreter supports method interceptions internally to facilitate its extensibility.

Libraries bundled with an execution platform simplify taint tracking because they have well-defined semantics. The taint tracking system can leverage their semantics and avoid tracking data propagation within the implementation of library methods. For example, a taint tracking system can avoid analysing code that implements a common data structure. Instead, it may calculate easily the taint meta-data for the return values of functions that perform well-known operations on the data structure. This is a significant advantage compared to taint tracking systems at the operating system or binary level.

The taint tracking policy should be configurable. If a taint tracking system is used to enforce data flow and not just for data flow analysis, the data flow policy that it enforces is likely to have to change during its lifetime. Ruby's default safe levels demonstrate this: they successfully prevent specific types of attacks but cannot be used easily to protect against the newer types of injection attacks that later became popular in the web. Recent systems such as GuardRails (§2.3.3) and SafeWeb completely bypass the existing mechanism and end-up replicating the operations of the taint tracking system in the language interpreter.

Another example that demonstrates the need for configurable taint tracking policies comes from recent research in the context of protection against XSS attacks [WSA+11]. Weinberger et al. show that in order to protect against many types of XSS attacks, the taint tracking system has to perform different sanitisation operations according to *the place untrusted data are used in an HTML document*. This requires intercepting tainted data at a later stage, i.e. when the full HTML document is available, compared to where most systems presented in this thesis intercept HTML output (GuardRails [BMW+11] is a notable exception). Without support for configurable taint tracking policies, this discovery can limit significantly the applicability of existing taint tracking systems.

The taint tracking policy should also be configurable so that the taint tracking system can satisfy custom data flow requirements of individual applications. A default taint tracking policy—despite being useful to enforce a particular data flow with limited or no configuration—is seldom enough. In Chapter 5 we showed that Wordpress had its own sanitisation rules that a taint tracking system must adhere to. Unless a taint tracking system is configurable and can adapt to these rules, it will not be practical.

7.3 Limitations in runtime taint tracking system design

Performing the work presented in the thesis has highlighted many important limitations that arise when designing a runtime taint tracking system. Often, such limitations are overlooked in relevant research despite the fact that they affect the capabilities of the resulting systems.

In this section we discuss some of these limitations. We hope that the discussion will inspire and help future work on runtime taint tracking systems.

The unit tracking granularity is restrictive. The unit tracking granularity, as used in DEFCon, SafeWeb and similar systems (§2.3.3), significantly constraints the processing that a unit can perform. It does not permit units to combine data of different security classes, and it increases the need for trusted code.

The unit tracking granularity considers all the data that a single unit processes to be of the same security class. This reduces the overhead of taint tracking at runtime because intra-unit tracking and checking operations are not invoked; but imposes important limits on the types of data flow policies that can be enforced. For example, a data flow policy may specify that data belonging to different users must not be disclosed to each other. Such a policy cannot be enforced if a single unit has to combine data on behalf of different users, even if the unit does not disclose data of one user to the other. Without access to the data of different security classes, many applications cannot be implemented. For example, inspecting the social circles of two users to dispatch messages between them is hard if a single unit cannot access the confidential list of friends of other users [PME+10]. Such requirements are common in practice, and we had to find solutions for the applications that we built both in DEFCon (§3.5.1) and in SafeWeb (§6.1.2).

Unfortunately, there is no strategy to avoid the constraints that the unit granularity imposes without weaknesses if access to data of different security classes is required. The only option is to relax the data flow policy, i.e. to allow a single unit to process data on different security classes and trust the unit that performs the processing not to introduce errors. This increases the trust a developer has to place on the codebase and defeats the purpose of using taint tracking to enforce data propagation. However it may be an acceptable solution if only a small part of the original unit has to be trusted. Managed subscriptions in DEFCon (§3.4.1) and event processes in Asbestos (§2.3.3) allow a single unit to efficiently process data of different security classes while avoiding cross-contamination. The downside here is that no code in such a unit can access data of a different security class because that would enable data flow that the taint tracking system cannot track.

Overall, we consider the unit tracking granularity more restrictive than the variable-level and character-level counterparts when general-purpose application development is considered. The unit tracking granularity is better suited for applications that rely more on isolation and, when data of different security classes have to be combined, the process occurs via trusted code (the Chrome extension architecture [BFSB10] is such an example).

Variable-level versus character-level taint tracking. Variable-level and character-level taint tracking systems are better positioned to track data propagation in general purpose applications. Character-level systems, in addition to discovering violations of data flow policy, can transparently mitigate many types of injection attacks. It is not clear, however,

if the additional overhead that they introduce is justified.

Variable-level taint tracking systems are easy to implement and introduce low overhead if implemented as part of the language interpreter. Their typical use case is to raise errors when dangerous data flow occurs and let the developers of the application devise the best solution to tackle the underlying security issue. This is often acceptable even if it means that an error in data flow analysis may prevent an application from executing in a specific use case.

In contrast, character-level systems such as PHP Aspis attempt to protect also against injection attacks and, for this, they require more detailed taint meta-data (i.e. per character). This additional layer of security is of particular importance when administrators have to secure legacy applications, which they do not maintain themselves. Unfortunately, the overhead of updating character-level taint meta-data is significant. Resin [YWZK09], for example, introduces overheads of up to 33% for serving a common web application despite a taint tracking implementation within the interpreter. In addition, due to the existence of control and implicit flows, which can lead to false negatives, a character-level runtime taint tracking system should not be trusted as the sole mechanism to enforce data flow.

For the above reasons, we consider variable-level taint tracking systems as most suitable for generic data flow analysis and enforcement. Character-level systems are also useful but the increased performance overhead and their inability to be used as the sole protection mechanism reduces their applicability for data flow enforcement across different applications.

Data flow policy enforcement in DEFCon using labels. DEFCon tracks data flow and enforces data flow policy specified in DPL using DEFC labels. Label-based data flow policy enforcement improves performance, however, it introduces challenges that may hamper the adoption of a taint tracking system.

The first limitation of label-based data flow enforcement is the inability to enforce policy that combines data flow with functional requirements. One such example was presented in Section 4.4.1, in which only reports that identified cancer incidents had to be forwarded to the cancer registry. DEFCon cannot enforce this policy without relying on a trusted unit because identifying which pathology reports constitute cancer reports is a functional requirement that cannot be expressed with labels. Instead, systems that use policy objects as taint meta-data (§2.3.3) can enforce functional requirements if these requirements are expressed with executable code by a policy administrator.

The second limitation of label-based enforcement is the need to translate high-level data flow policy to tags, labels and privileges. DPL facilitates this process but significant effort is still required to specify data flow policy and DEFCon cannot detect many policy misconfigurations. The reason for requiring a complex translation effort is that application-specific concepts that a data flow policy may need to reference (e.g. users, method invocations, data stores) do not have straightforward equivalents in labels, tags or privileges. Expressing data flow policy with code via policy objects does not suffer from this problem.

The last obstacle for practical label-based data flow policy enforcement is the observation that external data sources and data stores rarely provide labeled information to bootstrap data flow tracking or support storing labels with data. This makes maintaining the association of labels with data challenging without tools to help with this process (e.g. a label-aware file system or a database server that preserves labels).

Character-level taint tracking via source code transformations. PHP Aspis demonstrates that character-level taint tracking via source code transformations offers a way to secure existing web applications without support from the interpreter. It does not, however, demonstrate that this is the ideal way to implement taint tracking in PHP.

The performance overhead of performing taint tracking via source code transformations is high. Most of that overhead is introduced by replacing operations on scalar values with operations on arrays. While an alternative implementation may avoid this cost, language semantics limit the available design space (§5.3.2).

A second limitation when implementing taint tracking for a dynamic language via source code transformations is the need for an additional transformation step before executing the application. This can slow application development down significantly and renders the wide acceptance of the technique as the preferable way of securing PHP applications unlikely.

An ideal implementation of taint tracking for PHP will have to operate within the interpreter. A recent ongoing project[2] aims to offer variable-level taint tracking support as a binary extension to the PHP interpreter. The extension leverages the interpreter's ability to use extension-provided handlers for executing the intermediate code being generated. This approach, despite the fact that it requires significant expertise in the internals of the PHP interpreter to be implemented, is the most likely to be adopted widely.

7.4 Future work

Runtime taint tracking will continue to be an important tool for the security research community. However, for it to be a practical option to secure applications, research has to focus on runtime overhead and on interoperability with the languages and platforms popular with developers.

Explore techniques to speed up character-level taint tracking. The overhead of updating taint meta-data at the character level is the most important limitation for character-level taint tracking systems. In addition, the size of taint meta-data becomes significant because even though large parts of the character data used by an application may not be tainted, the taint tracking system has to reserve space to mark such data as untainted.

[2]TAINT extension, `http://svn.php.net/viewvc/pecl/taint/`, last accessed: 20/2/2013

There may exist multiple different techniques to bring the overhead of keeping taint meta-data per character down to more acceptable level. PHP Aspis groups equal taint meta-data of nearby characters but this increases the cost of random access to character strings. An alternative approach would be to store taint meta-data along with the actual string characters by taking advantage of the binary representation of characters in different encoding schemes.

Integration with web frameworks. Past research [FW11] has suggested that automatic protection mechanisms provided by popular web application frameworks, in contrast to manual protection, appear to be most effective against persistent security problems in the web. Future research must therefore focus on providing an implementation of taint tracking for most popular web frameworks that, in addition, can be customised to enforce arbitrary data flow policies.

A prominent example is the Ruby-on-Rails (RoR)[3] framework for Ruby web applications. By taking advantage of the Ruby features we employed in Chapter 6, taint tracking can be added to RoR with minimal overhead when activated. Such an implementation may be complimented with pre-configured taint tracking policies to protect web applications against different types of attacks (e.g. injection attacks). A taint tracking policy that, similar to SafeWeb, acts as a shadow protection mechanism to prevent sensitive user data disclosure due to implementation errors would also be useful.

Rethinking cloud security. Whilst cloud computing gains traction, security concerns about the loss of control over data are common between adopters [SSSF12]. Cloud providers consolidate data from multiple services, and this may result in widespread data disclosure if their security is compromised. Security in the cloud is challenging to achieve because it requires that the cloud platform cannot be compromised by hosted applications and that applications belonging to different cloud tenants are isolated to prevent data leakage.

A potential approach to improve cloud security would rely on taint tracking to enable clients and cloud providers to control how sensitive data are transferred across their systems and to prevent user actions violating data flow policy. Runtime taint tracking can offer the cloud platform a "safety net", protecting it against data flow policy violations caused by implementation errors in applications or vulnerabilities in the cloud platform itself.

Support for untrusted code in the browser. Web applications typically combine source code of different origin; external mapping services, analytics frameworks and presentation libraries can all be used as part of the same web page. Users of such web applications effectively trust all third-party source code with their personal data—even if they do not realise it. A malicious analytics framework, for example, which is imported in a page with the `<script>` tag, could leak a user's cookies to a third party. Existing Javascript security

[3]Ruby on Rails, `http://rubyonrails.org/`, last accessed: 12/9/2012

mechanisms, such as the same origin policy[4], cannot control how user data flow to different servers when third-party source code is used.

The browser can employ runtime taint tracking to augment the security that it provides. A potential application is to give users the ability to mark input data as sensitive and have the browser alert users before sensitive data is sent to any other destination but the domain of the web site currently being visited.

The browser is a particularly challenging environment for runtime taint tracking as the benevolent developer assumption does not hold for untrusted Javascript. Systems described in Section 2.3.3 [VNJ+07] combined taint tracking with static analysis, yet their effectiveness is unclear if the attackers are aware of the protection mechanism employed. Another challenge is how to reduce the rate of false positives, which increases significantly due to the need to monitor control flow.

[4]Same Origin Policy, W3C, `http://www.w3.org/Security/wiki/Same_Origin_Policy`, last accessed: 24/9/2012

Bibliography

[AF09] Thomas Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Programming Languages and Analysis for Security (PLAS)*, Dublin, Ireland, 2009. ACM.

[AF10] Thomas Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Programming Languages and Analysis for Security (PLAS)*, Toronto, Canada, 2010. ACM.

[AKD+08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Software Testing and Analysis (ISSTA)*, Seattle, WA, 2008. ACM.

[ALSU07] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, techniques, and tools.* Addison-Wesley, second edition, 2007.

[And99] Andrew Myers. JFlow: Practical mostly static information flow control. In *Principles of Programming Languages (POPL)*, San Antonio, TX, 1999. ACM.

[Arm07] Joe Armstrong. *Programming Erlang.* Pragmatic Bookshelf, 2007.

[Asa09] Matt Asay. Marketcetera gives hedge funds cloud-based trading. *CNET News*, 2009.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Annual Technical Conference (ATC)*, Anaheim, CA, 2005. USENIX.

[BEMP07] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Monitoring business processes with queries. In *Very Large Data Bases (VLDB)*, Vienna, Austria, 2007.

[BFSB10] Adam Barth, Adrianne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2010. Internet Society.

[BHL00] Godmar Back, Wilson Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2000. USENIX.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE, 1977.

[BKMW10] Sruthi Bandhakavi, Samuel King, Parthasarathy Madhusudan, and Marianne Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *Security Symposium*, Washington, DC, 2010.

[BL73] Elliott Bell and Lewis LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE, Bedford, MA, 1973.

[Blo08] Joshua Bloch. *Effective Java*. Prentice Hall, second edition, 2008.

[BMW+11] Jonathan Burket, Patrick Mutchler, Michael Weaver, Muzzammil Zaveri, and David Evans. GuardRails: A data-centric web application security framework. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[BSB11] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Recent Advances in Intrusion Detection (RAID)*, Menlo Park, CA, 2011.

[CD01] Grzegorz Czajkowski and Laurent Daynes. Multitasking without compromise: A virtual machine evolution. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2001.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2008. USENIX.

[CER00] CERT. Malicious HTML tags embedded in client web Requests (advisory CA-2000-02), 2000.

[CF10] Avik Chaudhuri and Jeffrey Foster. Symbolic security analysis of Ruby-on-Rails web applications. In *Computer and Communications Security (CCS)*, Chicago, IL, 2010. ACM.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[CMS03] Aske Christensen, Anders Moller, and Michael Schwartzback. Precise analysis of string expressions. In *Static Analysis Symposium*, San Diego, CA, 2003. ACM.

[CSL08] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Computer and Communications Security (CCS)*, Alexandria, VA, 2008. ACM.

[CSS08] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Paris, France, 2008. German Informatics Society.

[CVM07] Stephen Chong, K Vikram, and Andrew Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Security Symposium*, Boston, MA, 2007. USENIX.

[CW09] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. In *Secure Web Services (SWS)*, Chicago, IL, 2009. ACM.

[CXN+05] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks (DSN)*, Yokohama, Japan, 2005. IEEE.

[DC10] Benjamin Davis and Hao Chen. DBTaint: Cross-application information flow tracking via databases. In *Web Application Development (WebApps)*, Boston, MA, 2010. USENIX.

[Den76] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DG10] Vladan Djeric and Ashvin Goel. Securing script-based extensibility in web browsers. In *Security Symposium*, Washington, DC, 2010. USENIX.

[DJLS10] Jang Dongseok, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Computer and Communications Security (CCS)*, Chicago, IL, 2010. ACM.

[DKZ09] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Security Symposium*, Montreal, Canada, 2009. USENIX.

[Doc12] Perl Documentation. perlsec - Perl Security. `http://perldoc.perl.org`, May 2012.

[Duh09] Charles Duhigg. Stock traders find speed pays, in milliseconds. *The New York Times*, June 2009.

[EFGK03] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EFL+99] Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen. SPKI certificate theory. `http://www.ietf.org/rfc/rfc2693.txt`, 1999.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, 2010. USENIX.

[EK08] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, 2008. ACM.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005. ACM.

[Ell96] Carl Ellison. Establishing identity without certication authorities. In *Security Symposium*, San Jose, CA, 1996. USENIX.

[ESKK12] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Krugel. A survey on automated dynamic malware analysis techniques and tools. *ACM Computing Surveys*, 44(2), 2012.

[Exc09] London Stock Exchange. Hosting capacity increases fivefold. Press Release, November 2009.

[FJL+01] Francois Fabret, Hans-Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *ACM SIGMOD*, 2001.

[Fla07] Robert Flatley. Check your speed. *The Trade News*, 2007.

[Fla08] Yukihiro Flanagan, David Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[Foy07] Brian Foy. *Mastering Perl*. O'Reilly, 2007.

[FW11] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[GD07] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, Berlin, Germany, 2007.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Annual Technical Conference (ATC)*, Monteray, CA, 2002.

[GLS+12] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazi, and John Mitchell. Hails: Protecting data privacy in untrusted web applications. In *Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, 2012.

[GM82] Joseph Goguen and Jose Meseguer. Security policies and security models. In *Symposium on Security & Privacy*, Oakland, CA, 1982. IEEE.

[GMPS11] Min Gyung, Stephen McCamant, Pongsin Poosankam, and Down Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2011. Internet Society.

[Gol99] Dieter Gollmann. Public key infrastructures a tutorial. Microsoft Research, 1999.

[GTM+09] Nicolas Geoffray, Gael Thomas, Gilles Muller, Pierre Parrend, Stephane Frenot, and Bertil Folliot. I-JVM: A Java virtual machine for component isolation in OSGi. In *Dependable Systems and Networks (DSN)*, Estoril, Portugal, April 2009. IEEE/IFIP.

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From languages to systems: understanding practical application development in security-typed languages. In *Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, December 2006. IEEE.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Annual Technical Conference (ATC)*, New Orleans, LA, 1998. USENIX.

[HMP+11] Petr Hosek, Matteo Migliavacca, Ioannis Papagiannis, David Eyers, David Evans, Brian Shand, Jean Bacon, and Peter Pietzuch. SafeWeb: A middleware for securing Ruby-based web applications. In *Middleware*, Lisbon, Portugal, 2011. ACM/IFIP/USENIX.

[HO05] William Halfond and Alessandro Orso. AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks. In *Automated Software Engineering (ASE)*, Long Beach, CA, 2005. ACM.

[HOM08] William Halfond, Alex Orso, and Panagiotis Manolios. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *World Wide Web (WWW)*, New York, NY, 2004. ACM.

[Iat09] Rob Iati. The real story of trading software espionage. *Advanced Trading*, 2009.

[JKK06a] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. Technical report, Vienna University of Technology, 2006.

[JKK06b] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security & Privacy*, Berkeley, CA, 2006. IEEE.

[KEF+05] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazieres, Robert Morris, Michelle Osborne, Steven VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, 2005. USENIX.

[KGJK12] Vasileios Kemerlis, Portokalidis Georgios, Kangkook Jee, and Angelos Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Virtual Excecution Environments (VEE)*, London, UK, 2012. ACM.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, Finland, 1997.

[KT09] Maxwell Krohn and Eran Tromer. Noninterference for a practical DIFC-based operating system. In *Symposium on Security & Privacy*, Oakland, CA, May 2009. IEEE.

[KWH11] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: exploring a new aproach. In *Symposium on Security and Privacy*, Berkeley, CA, 2011. IEEE.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, 2007. ACM.

[Lam71] Butler Lampson. Protection. In *Information Sciences and Systems*, Princeton, NJ, 1971.

[LL05] Benjamin Livshits and Monica Lam. Finding security vulnerabilities in Java applications with static analysis. In *Security Symposium*, Baltimore, MD, 2005. USENIX.

[Luc02] David Luckham. *The power of events: An introduction to complex event processing in distributed enterprise systems.* Addison-Wesley, 2002.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems.* Springer-Verlag, 2006.

[Mic99] Sun Microsystems. Java Remote Method Invocation (RMI) specification, 1999.

[Min05] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *World Wide Web (WWW)*, Chiba, Japan, 2005. ACM.

[MIT12] MITRE Corporation. Common Vulnerabilities and Exposures (CVE) database. `http://cve.mitre.org`, 2012.

[ML00] Andrew Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Moc87] Paul Mockapetris. RFC 1034: Domain Names - Concepts and Facilities. IETF, 1987.

[MPE+10a] Matteo Migliavacca, Ioannis Papagiannis, David Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. DEFCon: High-performance event processing with information security. In *Annual Technical Conference (ATC)*, Boston, MA, 2010. USENIX.

[MPE+10b] Matteo Migliavacca, Ioannis Papagiannis, David Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Distributed middleware enforcement of event flow security policy. In *Middleware*, Bangalore, India, 2010. ACM/IFIP/USENIX.

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium (NDSS)*, San Diego, CA, 2010. Internet Society.

[NHS10] NHS. The handbook to the NHS constitution. `http://www.nhs.uk/choiceintheNHS/Rightsandpledges/NHSConstitution/Documents/COI_NHSConstitutionWEB2010.pdf`, 2010.

[NSCT07] Srijith Nair, Patrick Simpson, Bruno Crispo, and Andrew Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 197(1), 2007.

[NSCT08] Srijith Nair, Patrick Simpson, Bruno Crispo, and Andrew Tanenbaum. Trishul: A policy enforcement architecture for Java virtual machines. Technical report, IR-CS-045, Vrije Universiteit, Amsterdam, Netherlands, 2008.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *International Information Security Conference*, Chiba, Japan, 2005. IFIP.

[OMG02] OMG. The common object request broker architecture: Core specification, revision 3.0. Technical report, 2002.

[PB05] Tadeusz Pietraszek and Chris Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, Seattle, WA, 2005.

[Pie04] Peter Pietzuch. *Hermes: A scalable event-based middleware*. PhD thesis, University of Cambridge, 2004.

[PME+10] Ioannis Papagiannis, Matteo Migliavacca, David M Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Enforcing user privacy in web applications using Erlang. In *Web 2.0 Security and Privacy (W2SP)*, Oakland, CA, USA, 2010. IEEE.

[PMP11] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[PSB06] Georgios Protokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks for advertised honeypots with automatic signature generation. In *European Conference on Computer Systems (EuroSys)*, Leuven, 2006. ACM.

[PVCD02] Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynas. Incommunicado: efficient communication for isolates. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Seattle, WA, 2002. ACM.

[RCFZ11] Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yuri Zhauniarovich. YAASE: Yet Another Android Security Extension. In *Privacy, Security, Risk and Trust (PASSAT)*, Boston, MA, 2011. IEEE.

[Rus10] Andrei Russo, Alejandro Sabelfeld. Dynamic vs static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSFS)*, Edinburgh, UK, 2010. IEEE.

[SAB10] Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Symposium on Security & Privacy*, Berkeley, CA, 2010. IEEE.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Symposium on Security & Privacy*, Berkeley, CA, 2010. IEEE.

[SB09] Asia Slowinska and Herbert Bos. Pointless tainting: evaluating the practicality of pointer tainting. In *European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, 2009.

[SBL09] Kapil Singh, Sumeer Bhola, and Wenke Lee. xBook: Redesigning privacy control in social networking platforms. In *Security Symposium*, Montreal, Canada, 2009. USENIX.

[Sec05] PHP Security Guide 1.0. Technical report, PHP Security Consortium, 2005.

[Sho09] Kristin Shoemaker. Marketcetera 1.5 Release note. *Ostatic*, 2009.

[Sim03] Vincent Simonet. FlowCaml in a nutshell. In *APPSEM-II*, Nottingham, UK, 2003.

[SLD04] Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, December 2004. ACM.

[SM03] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SR09] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Andrei Ershov International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, 2009.

[SS75] James Saltzer and Mike Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSSF12] Dawn Song, Umesh Shankar, Elaine Shi, and Ian Fischer. Cloud data protection for the masses. *IEEE Computer*, pages 39–45, January 2012.

[TGP05] Bob Thome, Dieter Gawlick, and Maria Pratt. Event processing with an Oracle database. In *SIGMOD*, Baltimore, MD, 2005. ACM.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Waisman. TAJ: Effective taint analysis of web applications. In *Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, 2009. ACM.

[US 83] US Department of Defense. Trusted computer system evaluation criteria (orange book), 1983.

[VBM08] Alex Villazón, Walter Binder, and Philippe Moret. Aspect weaving in standard Java class libraries. In *Principles and Practice of Programming In Java (PPPJ)*, Modena, Italy, 2008. ACM.

[VEK+07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos Operating System. *ACM Transactions on Computer Systems*, 25(4), December 2007.

[Ven06] Wietse Venema. Runtime taint support proposal. In *PHP Internals Mailing List*, 2006.

[Vid04] Ganapathy Vidyamurthy. *Pairs Trading: Quantitative methods and analysis.* Wiley Finance, 2004.

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2007. Internet Society.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Theory and Practice of Software Development (TAPSOFT)*, Lille, France, 1997. Springer.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007. ACM.

[WSA+11] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. An empirical analysis of XSS sanitization in web application frameworks. Technical report, Berkeley, CA, 2011.

[WYC+08] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Iinternational Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, 2008. ACM.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Security Symposium*, Vancouver, Canada, 2006. USENIX.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Security Symposium*, Vancouver, Canada, 2006. USENIX.

[YJ10] Chunyang Ye and Hans-Arno Jacobsen. Event exposure for web services: a grey-box approach to compose and evolve web services. In *The Smart Internet*, volume 6400, pages 197–215. Springer, 2010.

[YNKM09] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, 2009. ACM.

[YWZK09] Alexander Yip, Xi Wang, Nickolai Zeldovich, and Frans Kaashoek. Improving application security with data flow assertions. In *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, 2009. ACM.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. Making information flow explicit in HiStar. In *Operating Systems Design and Implementation (OSDI)*, Seattle, WA, 2006. USENIX.

[ZBWM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, 2008. USENIX.

[Zen09] Alexandra Zendrian. Don't be afraid of the dark pools. *Forbes*, 2009.

[ZJS+11] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM Operating Systems Review*, 45(1):142–154, 2011.