Protecting Systems from Within:
Application-Level Observation and Control Mechanisms

By

BENJAMIN G. DAVIS
B.S. (University of Nebraska, Omaha) 2008
M.S. (University of California, Davis) 2014

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____

Hao Chen, Chair

____

Matthew Bishop

____

Karl Levitt

Committee in Charge

2014

Copyright © 2014 by
Benjamin G. Davis
All rights reserved.

To my wife, Caitrin, for her unyielding patience, understanding, and support.

CONTENTS

Contents iii

List of Figures vii

List of Tables viii

Abstract ix

Acknowledgments xi

1 Introduction 1

2 Android App Bytecode Rewriting 4

2.1 Introduction 4

2.1.1 Threat Model 6

2.1.2 Contributions 6

2.2 Design 7

2.2.1 Approach: Method Call Interception 7

2.2.2 Identifying Method Invocations 8

2.2.3 Intercepting Method Invocations 9

2.2.4 Policy Specification 12

2.2.5 Code Generation and Integration 17

2.3 Challenges 18

2.3.1 Reflection 18

2.3.2 Native Code 19

2.3.3 Intercepting Unexamined Code 19

2.3.4 Integration with App UI 19

2.4 Applications of Rewriting 20

2.4.1 Fine-Grained Network Access Control 20

2.4.2 HTTPS-Everywhere 21

2.4.3 Automatic App Localization . . . 22
2.4.4 Automatically Patching Vulnerable Apps . . . 23
2.5 Evaluation . . . 24
2.5.1 Evaluation Set Selection . . . 24
2.5.2 Rewriting Real-World Apps . . . 25
2.5.3 App Performance . . . 28
2.5.4 Rewriting Performance . . . 30
2.6 Discussion . . . 30
2.6.1 Transformation Policy Development . . . 30
2.6.2 Transformation Policy Application . . . 30
2.6.3 Legal and Ethical Discussion . . . 31
2.7 Comparison to Alternative Approaches . . . 32
2.7.1 Modifying the Android Platform . . . 32
2.7.2 Single-Purpose Rewriting . . . 33
2.7.3 Other Android-Specific Approaches . . . 34
2.7.4 Java-based Approaches . . . 35

3 Cross-Application Information Flow Tracking via Databases 37
3.1 Background . . . 39
3.2 Design . . . 41
3.2.1 Taint Model . . . 42
3.2.2 Information Flow Tracking in the Database Server . . . 44
3.2.3 Information Flow Tracking in the Database Client . . . 45
3.2.4 Database Client-Server Integration . . . 45
3.3 Implementation . . . 48
3.3.1 Database . . . 48
3.3.2 Perl Implementation . . . 50
3.3.3 Java Implementation . . . 51
3.4 Evaluation . . . 52
3.4.1 Database Operations . . . 52

3.4.2 Web Application: RT . . . 54
3.4.3 Analyzing Database Taint Values . . . 55
3.4.4 Enhancing Functionality . . . 57
3.4.5 Performance . . . 57
3.4.6 Web Application: JForum . . . 58
3.5 Discussion . . . 59
3.5.1 Benefits . . . 60
3.5.2 Additional Applications of DBTaint . . . 60
3.5.3 Inadvertent Untainting . . . 62
3.6 Comparisons to Alternate Approaches . . . 63

4 Privacy Preserving Alibi Systems **65**
4.1 Introduction . . . 66
4.1.1 Contributions . . . 67
4.2 Public Corroborator Scheme . . . 68
4.2.1 Overview . . . 68
4.2.2 Design . . . 69
4.3 Threat Model . . . 72
4.3.1 Identity . . . 73
4.3.2 Context . . . 73
4.3.3 Privacy . . . 74
4.3.4 Trust . . . 74
4.4 Properties of the Public Corroborator Scheme . . . 74
4.4.1 Security Properties . . . 75
4.4.2 Other Properties . . . 76
4.5 Private Corroborator Scheme . . . 77
4.5.1 Motivation . . . 77
4.5.2 Overview . . . 78
4.5.3 Initialization . . . 79
4.5.4 Alibi Creation . . . 79

4.5.5 Alibi Corroboration 80
4.5.6 Alibi Verification 81
4.6 Properties of the Private Corroborator Scheme 82
4.6.1 Privacy 82
4.6.2 Reciprocity 82
4.7 Comparison to Physical Alibis 83
4.7.1 Common Properties 84
4.7.2 Benefits 84
4.7.3 Weaknesses 85
4.8 Performance Evaluation 85
4.8.1 Benchmarks 86
4.8.2 Storage 87
4.9 Discussion of Alternate Approaches 87

5 Conclusions **90**
5.1 Acknowledgments 91

List of Figures

2.1 RetroSkeleton system diagram 8
2.2 Internal representation of a target method and associated properties 13
2.3 Policy specification for selecting targets by method name and parent class . . 15
2.4 Examples of handler generator helper values 16
2.5 Example handler generator function 17

3.1 Serial throughput of web services running with and without DBTaint 61

4.1 The public corroborator scheme 68
4.2 The private corroborator scheme 79

List of Tables

2.1 Success rate of rewritten apps 26

2.2 Impact of app rewriting on bytecode size 28

2.3 Impact of app rewriting on overall app size 29

3.1 Overhead of DBTaint database operations 53

3.2 Overall overhead of DBTaint for web services 58

4.1 Average execution times for alibi operations on a Motorola Droid 86

ABSTRACT OF THE DISSERTATION

Protecting Systems from Within:
Application-Level Observation and Control Mechanisms

Popular software systems are used in a wide variety of settings and each user or organization that depends on a system may have requirements or needs beyond what the system is designed to provide. Many of these systems do not provide the insight into or control over the system needed to satisfy these additional requirements. In this work we show that we can address many of these challenges by augmenting existing systems with new application-level mechanisms. This approach has a number of advantages, including the flexibility to leave policy decisions up to the user, the power to enforce complex policies that depend on the data, context, and setting of the system, and the practicality to be used with existing, real-world systems. This dissertation demonstrates this philosophy applied to three different settings.

First, we empower users of Android devices to add, remove, and enforce behavior in the third-party apps they install. We present RetroSkeleton, our Android app bytecode analysis and rewriting framework, and show how it can be used to build observation and control mechanisms into Android apps. Our bytecode analysis engine enables app-agnostic policies to be applied to any app without any manual or app-specific guidance. We develop policies, including enforcement of fine-grained network access control, HTTPS-Everywhere functionality for app network activity, automatic app localization, and patching apps to protect users from vulnerabilities in the Android platform, and apply these policies to the top Android apps.

Second, we enable web service administers to track the flow of information through their web applications and back-end databases. Our system, DBTaint, propagates fine-grained information flow tracking metadata across application boundaries and through database operations in web services automatically. Our system operates transparently to web applications and requires no changes to the database engine. We apply our system to two existing web services and demonstrate it is effective, efficient, and practical for real-world settings.

Third, we develop a system that gives mobile device users the ability to establish and present “alibis” (evidence of their past locations) while retaining control over the disclosure

of their location history. We present two cryptographic schemes to facilitate the automatic, opportunistic creation of these alibis. In our designs, the identity associated with an alibi can only be revealed by the owner of that alibi, giving the user complete control over their privacy. Our schemes require no trusted third party for our privacy guarantees and we show that our schemes run efficiently on mobile devices in terms of both storage and computation requirements.

ACKNOWLEDGMENTS

I feel incredibly lucky for the opportunities I've had to work with and learn from so many thoughtful, intelligent, and talented people. I would like to thank my advisor, Professor Hao Chen, for all of his guidance and support throughout my time at the University of California, Davis. Thank you to all of the members of the Computer Science department for providing a positive environment to work, learn, and grow. I am particularly grateful to Professor Matthew Bishop and Professor Karl Levitt for serving on my committee and providing valuable feedback and guidance on my research. Thank you to the many mentors and guides I've had along the way who encouraged me to pursue exciting and interesting challenges, including Dr. Blaine Burnham and Steve Nugen at the University of Nebraska, Omaha, and Dr. Deborah Frincke and Mark Hadley at Pacific Northwest National Laboratory.

I have many fond memories of the time I spent with my fellow grad students. Thank you to the original Kemper team for helping me find my way when I started, especially Gabriel Maganis, Matthew Van Gundy, Liang Cai, Francis Hsu, and Yuan Niu. Also, thank you to the entire Watershed crew for all of the fun we had at our meetings, outings, and everything between. I've learned so much from you all.

Most of all, I'd like to thank my family, especially my parents Tom and Jo, my brother Luke, and my wonderful wife Caitrin. I cannot overstate how much your unwavering encouragement and support have meant to me, and I am truly grateful to have you all in my life.

Chapter 1

Introduction

As software continues to become an increasingly important part of our personal and professional lives, it is our responsibility to understand the capabilities of the systems on which we depend. Most users do not develop their own systems and instead rely on those created by others to meet the requirements of their environment and fulfill their organizational goals. Unfortunately, it is often difficult for a user to observe and understand exactly what the systems they use are doing, much less control the behavior of these systems, in order to ensure the system performs appropriately in their setting. This can be particularly challenging when a user of a general-purpose system designed for widespread use has specific requirements beyond the scope (or incentives) of the system's original designers. The following work is motivated by the desire to give users insight into and control over the systems on which they depend.

This work demonstrates that it is possible to develop application-level mechanisms that empower users to observe and control the systems they use in a way that is flexible, powerful, and practical. An in-application approach supports the development and deployment of complex and useful policies that enforce requirements based on the data and context of the system and the environment in which it runs. The frameworks described below enable users to enforce their policies in a wide variety of systems, including third-party systems not developed with the user's policies in mind. While the most obvious applications of these capabilities may be enforcement of security and privacy requirements, these frameworks are general enough to support other features, such as localization, advanced testing, and data provenance. Each framework is evaluated by using it to augment existing real-world systems to provide new

functionality to users.

The following chapters describe the application of this philosophy to three different settings:

Automatic Android App Bytecode Analysis and Rewriting. Android is the world's most popular mobile platform, and Android users have a tremendous number of third-party apps available to them. However, as mobile devices become increasingly integrated into our lives, users are right to be concerned about what these third-party apps may do with their personal data and the sensors on their devices. Unfortunately, Android provides only limited, coarsely-grained control over app behavior, and platform designers may not have the incentives to provide the insight and control over apps that some users desire. In Chapter 2, we present a general Android app bytecode rewriting framework that enables users to observe and control the functionality of the third-party apps they use. We empower users to enforce the policies they want in the apps they use without requiring the cooperation of the app developers, modifications to the Android platform, or rooting or other changes to the configuration of the users' devices. The Android platform is large and complex, and analysis is complicated by frequent dependence on dynamic features such as reflection, so our system provides advanced features for determining and applying the appropriate policy implementations automatically. We demonstrate several uses of this approach by developing a variety of policies and apply them to real-world apps.

Cross-Application Information Flow Tracking through Web Services via Databases. Many organizations rely on web services for communication and coordination amongst their members. Some general services, such as forums and ticket-tracking systems, are deployed in a wide variety of environments, and may not be designed to provide the insight or control necessary to enforce the specific policies each organization desires for their setting. Furthermore, there are inherent challenges in securing a web service based on receiving, processing, and displaying user content, as mistakes may allow the disclosure of private data or execution of untrusted input in the browsers of victims using the service.

Traditionally, information flow tracking systems enable system administrators to monitor the flow of data through a program, but existing implementations are unsuitable for the

common web service architectures that include web applications backed by a database. Prior approaches lack either the necessary granularity for web services, the ability to preserve tracking metadata across the application/database boundary, or the capability to propagate metadata appropriately with understanding of the semantics of database operations. In Chapter 3 we describe our framework that provides information flow tracking through entire web services, from the web application into the database, through database operations, and back. Our approach provides this functionality by augmenting existing services in a way that is transparent to the web application and requires no changes to the underlying database engine implementation. This design enables use with systems designed without information flow tracking in mind, which we demonstrate on two popular, real-world web services: RequestTracker, a ticket-tracking system written in Perl, and JForum, a forum system written in Java.

Privacy-Preserving Alibis for Mobile Device Users. In a recent high-profile criminal investigation [68], charges against a murder suspect were dropped when he provided his NYC Transit MetroCard as evidence that he was elsewhere at the time of the murder. This example demonstrates why it can occasionally be extremely important to be able to provide evidence of being at a particular location at a particular time, and as more people carry mobile devices with location sensors it seems natural to explore how this technology can be used towards this end. Unfortunately, most existing location-tracking systems simply record all past locations, providing no privacy to users and little evidence to corroborate claims, and others assume a trusted third party or rigid infrastructure requirements unavailable and impractical for use by existing mobile users.

In Chapter 4 we present two schemes that allow a user's mobile device to opportunistically and automatically establish evidence to support claims of past locations ("alibis") on the user's behalf without revealing the user's identity. These schemes bind the user's identity to an alibi when the alibi is created, but the identity is only revealed if and when the owner claims the alibi, leaving the user in complete control of the disclosure of their identity. These schemes are designed to work within the existing infrastructure available to mobile device users, and do not require additional or existing trusted third parties to fulfill the privacy guarantees of our schemes.

Chapter 2

Android App Bytecode Rewriting

Android is the world's most popular mobile OS, with over one million 3rd-party apps available. Unfortunately, the Android platform provides users with very limited insight into and control over the behavior of these apps. In this chapter, we describe our solution to this problem: RetroSkeleton, a system for automatic Android app bytecode rewriting. Our system enables users to enforce or change the behavior of existing apps without requiring source code or app-specific guidance. Following app-agnostic transformation policies, our system rewrites apps to insert, remove, or modify behavior, producing rewritten apps that can run on any unmodified Android device. Our application-level instrumentation makes it easier to develop sophisticated policies that integrate deeply into an app's behavior. We develop and apply a variety of useful policies to real-world apps, including providing flexible fine-grained network access control, building HTTPS-Everywhere functionality into apps, implementing automatic app localization, and embedding workarounds for platform bugs and deprecated APIs.

2.1 Introduction

Android is the "world's most popular mobile OS," currently running on more than a billion mobile devices [3]. A major feature of the Android platform is the tremendous number of third-party apps available for Android devices, with more than one million Android apps available on Google Play [6] alone.

Many Android apps have access to a wide variety of sensitive information, including banking and payment details, email and messaging content, photos, and browsing history. It may

be difficult for a user to determine exactly what an app may do, so the user must rely on their limited assessment of the trustworthiness of the developers of the app and of any included third-party libraries. The controls provided by the Android platform, such as the permission system, are extremely limited, coarse-grained and inflexible. For example, installing an app that requests the `INTERNET` permission grants the app permanent, unlimited access to the network. Most users with devices associated with a telecommunication provider either can not or do not root their devices, which can be a risky and unsupported operation, leaving them without access to privileged and low-level functionality on their device. As a result, users are unable to apply many practices commonplace on traditional desktop PCs, such as installing a third-party firewall to monitor incoming and outgoing connections.

Our work is based on the observation that the vast majority of Android apps (estimated over 95% [72]) are compiled entirely into Dalvik bytecode and related assets (i.e., contain no native code). Not only do most apps target the same platform, but Dalvik bytecode does not include the ambiguity that frequently makes analysis of other compiled software (e.g., x86 machine code) impractical. We take advantage of the consistency in app implementation to design and implement an Android app rewriting framework for customizing behavior of existing applications without requiring source code or app-specific guidance. We call our app rewriting system RetroSkeleton after the ability to retrofit apps with new behavior by modifying their internals.

RetroSkeleton can be used to provide users with more control over the security, usability, and functionality of apps, even compared to desktop applications. We provide a system for automatically embedding policies completely into applications via automated app rewriting to insert, remove, or modify app behavior. Our system is capable of supporting a wide range of flexible, customizable transformation policies that can be written and applied to apps automatically without any app-specific knowledge or guidance. In this work we describe several different policies we have implemented and applied to real apps, providing features such as fine-grained network access control, HTTPS-Everywhere features, or on-the-fly localization of app UI elements. Apps transformed by our rewriting system can be installed on stock, unmodified and unrooted Android devices.

Our Android app rewriting system was designed to have four important properties:

- **Complete**: intercept all target method invocations, including dynamic invocation via reflection
- **Flexible**: powerful enough to support a variety of complex transformation policies
- **App-Independent**: no manual app-specific guidance needed to create or apply transformation policies
- **Deployable**: rewritten apps work on unrooted, unmodified Android devices, no additional software required

2.1.1 Threat Model

We assume that the user of a rewritten app trusts the Android platform, the transformation policy writer, and rewriting process, but not the original app code. We do not attempt to hide the fact that the app has been modified from the rewritten app.

2.1.2 Contributions

The work described in this chapter includes the following contributions.

- Design and implementation of an Android app rewriting system capable of replacing arbitrary method calls with custom handlers
- Flexible framework supporting the application of app-agnostic transformation policies to arbitrary Android apps
- Sample rewriting policies, including the integration of fine-grained access control, improved security of network communications, automatic localization of UI content, and protecting users from platform vulnerabilities
- Application and evaluation of these polices to over one thousand top real-world apps available via Google Play

A key aspect of our system is the ability for policy-writers to specify high-level policies that can be applied to arbitrary apps automatically without any app-specific guidance. Policy writers identify the Java methods of interest and write Java source code, or higher level functions that generate Java source, for the bytecode operations they wish to embed into rewritten apps. Our system automatically generates all handler classes, methods, and supporting code to intercept all method calls of interest, even those invoked via reflection. Our system can apply a single policy to any real-world app automatically without requiring app-specific guidance.

2.2 Design

2.2.1 Approach: Method Call Interception

Most (estimated over 95% [72]) Android apps are compiled entirely into Dalvik bytecode, and method calls are the primary mechanism that Android apps use to interact with the underlying device. Rather than providing direct access to underlying system resources, the Android platform provides a set of APIs that apps use to observe and influence the state of the device (such as making network requests, accessing GPS and other sensor data, accessing the user's contacts list, and managing the UI).

To support a wide range of practical and useful policies, we've designed a flexible Android app bytecode rewriting framework that analyzes Android app bytecode to statically identify all potential invocations of methods of interest, which we refer to as *target methods* (Section 2.2.2). Our system rewrites the app bytecode to replace every invocation of a target method with an invocation of a corresponding *handler method* containing the replacement functionality. Intercepting app behavior at method invocations allows policy writers to take advantage of existing knowledge of the Android platform, rather than building policies around the implementation details of how app behavior may manifest itself in low-level system calls invoked by the Dalvik virtual machine.

We describe how our system identifies and rewrites each of the many ways a method can be invoked in Dalvik[1] bytecode below. While our in-app Dalvik method interposition capabilities form the foundation of our system, these operations can be challenging to use directly for non-

[1] While tools exist that attempt to convert Dalvik bytecode into Java bytecode for easier analysis, they are imperfect [62], so we parse, analyze, and transform Dalvik bytecode directly.

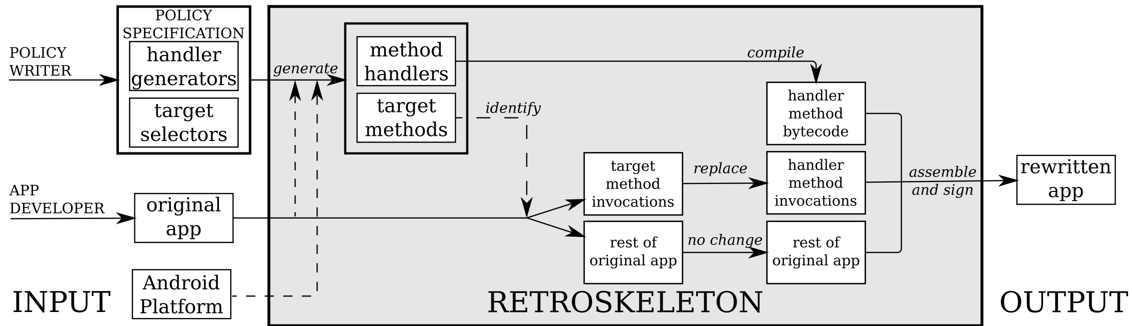

Figure 2.1. RetroSkeleton system diagram

trivial policies, as the precise methods to target and behavior to inject may vary between target apps and across different versions of the Android platform. Furthermore, because Dalvik supports indirect and dynamic invocation mechanisms (e.g., for virtual method invocation), the method specified in a bytecode instruction may differ from the method executed at runtime, requiring the transformation of additional invocation points. In Section 2.2.4 we describe how our system determines the appropriate transformations for any Android app automatically from a single, high-level, app-agnostic policy specification. See Figure 2.1 for a high level view of the components of our rewriting system design.

2.2.2 Identifying Method Invocations

Most Android apps are written in Java and compiled to Dalvik bytecode for Android's Dalvik virtual machine. Dalvik bytecode includes several different instructions for each type of method invocation (e.g., `static`, `direct`, `virtual`, `super`). Each invocation instruction references the method signature (method name, containing class, and parameter types) and the registers associated with the call. While the architectures differ, Dalvik bytecode is similar to Java bytecode in that instructions are clearly defined by the specification and can be interpreted without the ambiguities that complicate the analysis some other platforms (e.g., x86 machine-code).

While invocation instructions can be identified unambiguously, it may be impossible to statically determine the precise method that will be invoked when the app is run because the Dalvik VM supports dynamic dispatch and virtual method invocation. For example, a target method may be invoked at runtime by an instruction that specifies a corresponding

method in the parent class (invoked virtually), or in a child class (which has not overridden the target method). To guarantee completeness, we intercept all invocations that *may* result in execution of the target method at runtime. We describe how our system identifies all related classes, methods, and appropriate transformations automatically from an app-agnostic policy specification in Section 2.2.4.

2.2.3 Intercepting Method Invocations

After identifying all instructions related to (potential) invocations of target methods, our system replaces these with calls to the corresponding handler methods, passing all parameters and instance objects involved in the original call to the handler. This enables handlers to perform dynamic runtime introspection and invoke the original target method if desired (e.g., after notifying or asking the user), though this is not required. Methods may be invoked in many ways, so our system combines multiple strategies for bytecode rewriting based on the type of invocation, app and platform class hierarchy, and attributes of the target method.

2.2.3.1 Stub Methods

Our system replaces an invocation that directly specifies a target method with a call to a type of handler method we call a *stub* method. All stub methods are public and static, but the way our system replaces calls to target methods with calls to stub methods depends on the type of the target method.

Static Methods Static target method invocations are replaced with invocations of stub methods that take exactly the same parameters.

Instance Methods Instance target method invocations are replaced with calls to stub methods that take, as parameters, a reference to the instance object on which the target method was originally invoked and all parameters associated with the original call. This allows these handlers to inspect and invoke the original target method on the instance object if desired.

Constructors Intercepting invocations of target constructors is more difficult than static and instance methods because object construction spans multiple Dalvik instructions. First, the `new-instance` instruction creates and stores a reference to an uninitialized object in a register. Second, the `invoke-direct` instruction takes a register containing a reference to the uninitialized object, and invokes the constructor to initialize this object in place.

The `new-instance` and varieties of `invoke-direct` instructions can reference different ranges of registers, so not only is it common for these instructions to not be adjacent, but frequently multiple registers hold the same uninitialized reference when the constructor is invoked.

For some policies it is impossible to create a new constructor to replace each target constructor, such as when the target constructor's class is `final` preventing the addition of a descendant class. Instead, our system generates static "factory" methods that take the parameters passed to the original constructor, and constructs an object of the appropriate type. Unfortunately, the Android verifier rejects Dalvik bytecode that invokes non-constructor methods that operate on uninitialized references, so our factory methods cannot initialize the object in place. Each factory method returns an object of the appropriate type, and a reference to the new object is copied into the register on which the constructor was originally invoked.

Replacing calls to target constructors with factory methods only updates the register originally passed to the constructor. We must also update any other registers that contain a reference to the original uninitialized object. In our system we use static analysis to identify all registers that hold a copy of the original reference at the point of construction, so each can be updated to reference the new object. The Dalvik VM uses virtual registers local to each method, so our system analyzes each method body to match related instructions and identify the flow of the uninitialized references through multiple registers. After inserting the instruction to construct an object using the appropriate factory method, our system updates all related registers as appropriate.

2.2.3.2 Wedge Classes

While the stub-method approaches described above cover many cases, we need a separate technique to handle the use of inheritance and virtual method invocations frequently found in Android apps. We handle the remaining cases by analyzing the classes declared in each Android app during rewriting, and injecting classes providing handler methods into the class hierarchy of the original app.

Developers of Android apps often create new classes that extend existing classes. Imagine we wish to intercept the `bind` method for the `DatagramSocket` class in the `java.net`

package, and an app developer creates a `DevSocket` class that extends `DatagramSocket`. The way we handle the invocation of the `bind` method on an instance of the developer's class depends on the goals of our policy.

Full-hierarchy interception Our system supports full-hierarchy interception, meaning that given a transformation policy containing a target method m belonging to class A, our system also intercepts all invocations of m on objects of type B where B extends A. During rewriting, our system identifies the hierarchy of all classes defined and used in an app, and augments the transformation policy by adding B's method m to the targets list and generates the associated handler code.

Partial-hierarchy interception In practice, it is often more useful to intercept only calls that invoke the method in the parent class. For example, many Android platform methods provide the API to low-level interactions with the underlying device, such as sensor, network, and file system access. Many useful transformation policies control the app's ability to interact with the API in the framework, but are not concerned with intercepting developer methods that override target methods without invoking the original target method.

For example, if the developer creates a `DevSocket` class that extends the platform class `DatagramSocket` and overrides the `bind` method with an implementation that performs no network operations, a transformation policy controlling network access would not want to intercept this method. On the other hand, if the developer overrides `toString` with code that invokes `super.bind` then we want to intercept this request.

We provide this functionality by generating *wedge classes* that extend non-final classes containing target methods and include *wedge method* handlers for each target method in the parent class. In our example, our wedge class extends `DatagramSocket` and overrides `bind` with the handler method implementation. Our system identifies all classes extending the target class, modifies them to instead extend our wedge class, and replaces all internal calls (e.g., via `super`) as appropriate.

In our rewritten app, invoking the `bind` method on an instance of `DevSocket` only invokes our handler when `DevSocket` has not overridden `bind`, or if the developer's class calls `super.bind` from within the class. This allows us to properly intercept exactly those

calls to the platform target methods, without intercepting developer methods that do not invoke our target methods. This approach supports invoking the original target method from within the handlers even when the target method cannot be invoked directly from the app, such as `protected` target methods.

2.2.3.3 Use of Stub Methods and Wedge Classes

We combine the stub and wedge class approaches to intercept all target method calls however invoked in the original app. Static, direct, and virtual method invocations of target methods are rewritten to stub method calls. Virtual and super invocations on non-target methods that resolve to target methods are intercepted by wedge methods. We do not generate stub or wedge methods in cases where they are unnecessary. Specifically, we do not wedge final or abstract methods or create stub methods for target methods that cannot be called publicly. We describe how we handle reflection and other special cases in Section 2.3.

2.2.4 Policy Specification

The method-call interception strategies above require concrete identification of each individual target method, every associated method property, and implementations of handler behavior in Dalvik bytecode. Not only can it be tedious to determine, specify and maintain this data manually for complex policies, but the precise targets and properties may depend on the version of Android used and the implementation of the target app. Our system provides a high-level interface that enables policy writers to define sophisticated policies with a single app-agnostic specification without needing to understand how the policy will be applied in Dalvik bytecode.

2.2.4.1 Target Method Specification

Internally, RetroSkeleton specifies a target method in a representation that maps the unique signature of a method, which contains the method name, parameter types, and containing class, to a key/value representation of the properties of the method. See Figure 2.2 for an example representation. As RetroSkeleton runs, additional properties are added to the key/value pairs associated with each method, providing information about how invocations of the target should be rewritten, and the identity and behavior of handler methods associated with that method. This allows our single internal representation to be used throughout the system for bytecode analysis, rewriting, and code generation.

```
{
  ["Lcom/example/ParentClass;",
   "myMethod",
   ["Ljava/lang/String;", "Landroid/os/Bundle;"]]

  {:return "Ljava/lang/Object;",
   :attributes #{:final, :protected},
   :exceptions #{"java.lang.IllegalAccessException",
                 "java.lang.IllegalArgumentException"}}
  ...
}
```

Figure 2.2. Simplified internal representation of a target method and associated properties

Our system provides mechanisms to aid policy development by automatically identifying every desired target methods and associated properties from the set of all available platform and app methods, and allowing policy writers to create functions that generate handler behavior based on the properties of these automatically-selected target methods.

2.2.4.2 Target Method Selection

The precise methods that should be intercepted for a given policy may vary based on the version of the Android platform for which the app was developed. For example, imagine a policy writer wishes to intercept platform method calls that make network requests. PScout [13] identifies 223 platform method calls associated with the `INTERNET` permissions in Android 2.2.3, but 316 in Android 2.3.6, 478 in Android 3.2.2, 529 in Android 4.0.1, and 471 in Android 4.1.1. Furthermore, policy writers may wish to intercept app methods without requiring app-specific knowledge or targeting for policy development.

RetroSkeleton allows policy writers to specify *selectors* that our system uses to identify the appropriate target methods automatically. During rewriting, our system determines every class and method defined in the app by analyzing the Dalvik bytecode directly. Then, after determining the version of the Android platform targeted by the app our system uses ASM [16]

to analyze all classes and methods in the official Android SDK `jar` files for that version. Our system combines all of this class, method, and attribute data into a single relational structure representing all classes and methods potentially available to the target app. This structure is a directed, acyclic graph with `java.lang.Object` at the root. Some methods and properties of a class are not specified in the bytecode of the given class, but are inherited from a parent class, often crossing the boundary between the app and the platform. We propagate inherited properties through our unified (platform and app) class hierarchy to form a complete tree containing all classes and methods, adding annotations indicating origin (app or platform code) and implementation location (local or inherited from a parent class). Within this structure containing the representation of the class hierarchy, the methods associated with each class are represented following the form of the example shown in Figure 2.2.

Instead of specifying the complete list of target methods manually, policy writers use declarative *selectors* that our system applies to identify all necessary target methods at rewrite time. These selectors are app-agnostic declarative queries that identify target methods based on the properties of the method and containing class. While policy writers are free to query this structure however they wish, our system provides a number of convenience function for most common cases. Among these is the `target-all` function that, given a class and method name, will identify all methods in that class with that method name (regardless of parameter types) and all methods with that name in descendant classes (inherited or overridden), across all available platform and app code. The policy writer only needs to provide the method and containing class name and a function for generating the handler behavior for each target method it finds (described in Section 2.2.4.4). Our system also provides similar convenience functions for intercepting all `native` methods declared in the app (see Section 2.3.2).

Figure 2.3 shows how a policy writer specifies that all constructors in platform and app classes that descend from the `Socket` class, and all methods named `onResume` in classes descending from the `Activity` class, should be intercepted. Both sets of target methods will be intercepted by handlers that add logging to the `logcat` service before proceeding as normal, as described further in Section 2.2.4.4.

```
(target-all with-policy-logcat
  "Ljava/net/Socket;" "<init>")

(target-all with-policy-logcat
  "Landroid/app/Activity;" "onResume")
```

Figure 2.3. Policy specification for selecting targets by method name and parent class

2.2.4.3 Selective Transformation Application

Our system also supports policies that apply different transformations based on the invocation (call) site of the target method. For example, imagine a policy that intercepts methods associated with network requests invoked from classes in an advertising library, but not those called from app code. Policy writers can identify call-sites at a class-level using selectors over our unified class structure, or from the results of an external tool, making the classification of call-sites of interest orthogonal to the application of our transformations. Our system also supports a dynamic approach capable of identifying the source of invocations at run-time, using the call-stack for policy decisions (see Section 2.4.1.1).

2.2.4.4 Handler Method Behavior

Policy specifications identify the methods to intercept and the behavior of the handlers associated with each target method. These handlers completely replace the behavior of the target method when called by the rewritten app, and policy writers may insert nearly any functionality available to normal app developers. We only require that handler methods return values corresponding to that of the original method, and declare the same set of checked exceptions (in order to preserve app functionality). While we recommend that handler methods do not throw new runtime exceptions that the original app would not expect from an invocation of the target method, we do not enforce this.

Our system identifies the relevant attributes and generates the declaration of each handler method automatically, so policy writers only need to provide the method bodies. We provide an interface for specifying handler behavior by writing Java source code so policy writers can leverage knowledge and experience from normal app development and avoid specifying

Helper Name	Example Value
[cls mname params]	["Landroid/app/Activity;" "onResume" []]
passthrough	"super.onResume(); return;"

Figure 2.4. Examples of handler generator helper values computed automatically when targeting `Activity.onResume()` for wedge-class interception

Dalvik bytecode directly. Because our policy-specification interface allows target methods to be identified at rewrite-time, the policy writer may not know exactly which target methods will be selected when specifying the handler behavior. To support this variability, policy writers may also associate generator functions with each selector statement. These generator functions produce the associated handler behavior (as Java source code) for each target method given the properties of the target method at rewrite-time.

To further assist policy writers, our system provides helpers to automatically generate source code for common handler behaviors. The `passthrough` expression invokes the original target method from within the handler. The `deny` expression ends the handler method without invoking the original target or violating the expectations of the call-site. For example, the `deny` for `void` methods simply `returns`, but the expression for target methods with checked exceptions will throw an exception of the expected type, as it will be handled by the calling code. Handlers also have access to helper expressions for the containing class, method name, and parameter types (represented as `cls`, `mname`, and `params` in these examples), as well as for the method attributes, checked exceptions declared, return type, and more. See Figure 2.4 for an example values generated for some of these helper expressions.

While their use is not required, these helpers can facilitate general handler generators applicable to a wide variety of target methods. Figure 2.5 shows an example handler generator function named `policy-logcat`, which creates handlers that logs a method call to the `logcat` logging systems before invoking the original target method. The policy writer creates a single function with the implementation shown. This function uses several of the helper values available (shown in Figure 2.4) to create a general handler function that can be used with any target method. The bottom of Figure 2.5 shows the Java source code generated when RetroSkeleton invokes the handler generator for the wedge-class interception of the

Generator Name	`policy-logcat`

Implementation	`(str "android.util.Log.i(\"RSLogger\", \""`
	`     cls "->" mname`
	`     "(" (join params) ")"`
	`     "\");\n"`
	`     passthrough))`

Output (Java Source)	`android.util.Log.i("RSLogger",`
	`  "Landroid/app/Activity;->onResume()");`
	`super.onResume();`
	`return;`

Figure 2.5. General logging handler generator function and the handler body produced when targeting `Activity.onResume()`. This generator uses helpers shown in Figure 2.4.

`Activity.onResume()` target method. A policy writer can use whatever programming features they wish when implementing their generator functions, though the availability of these helper values means that many generators require little more than string concatenation, as in this example.

2.2.5 Code Generation and Integration

All Dalvik bytecode in an Android app is combined into a single `classes.dex` file. We allow policy writers to specify handler behavior in Java source code, but without the source code for the original app we cannot recompile the entire app with the new handler source. Instead, we generate the Dalvik bytecode for our handler methods in new classes separately, then merge the handler method bytecode into the rewritten application bytecode before assembling the rewritten app.

During rewriting, our system creates a new, nearly empty Android project, where it generates source code for all wedge classes, classes containing stub methods, and other support code. In each of our wedge classes, we also generate constructors corresponding to each

constructor in the parent (target) class. Constructors are not automatically inherited, so if a wedged class is expecting a particular constructor to be present in the parent class, we must provide it in our wedge class. These constructors simply invoke the associated target classes' constructor unless otherwise specified in the transformation policy. These classes are placed into a unique package that will not collide with any classes in the original app.

If the policy targets methods declared in the original app then handler methods may require app-specific classes as parameter, return, or exception types. To enable the compilation of these handlers, our system generates dummy *skeleton classes* for each of the missing types and methods. These skeleton classes are not included in the rewritten app, and are only used so the Android build tools can compile the source for the handlers and generate valid Dalvik bytecode with the appropriate type signatures.

We use the official Android development tools to compile and build an app containing Dalvik bytecode for all of our handler classes. We disassemble the bytecode of this app and extract the bytecode for all of the stub, wedge and other support classes included in the policy. We merge the bytecode for these classes into the app we are rewriting, and then assemble[2] the result into a new `classes.dex` file including all of the rewritten versions of the original app code and our handler classes.

2.3 Challenges

2.3.1 Reflection

Android apps can invoke methods specified dynamically via Java's reflection API. While the precise methods invoked via reflection cannot always be identified statically, we *can* identify the calls to the reflection API statically. For example, we statically identify and rewrite calls to Java's `Method.invoke` reflection method even though the parameters may be determined at runtime.

Our handler methods for the reflection API perform dynamic inspection of the parameters to determine at runtime the method the app is attempting to invoke via reflection. If it resolves to a target method, the handler invokes the associated target handler and returns the

[2] using the `smali` [40] Dalvik bytecode assembler/disassembler

result, otherwise our handler performs the requested invocation as expected. These reflection handlers are recursive in the sense that they can handle attempts to invoke the reflection API via reflection.

2.3.2 Native Code

While it is relatively uncommon (estimated less than 5% by Zhou et al. [72]), Android applications may also include native code. Native code is compiled into machine-code specific to a particular architecture, and the techniques for analyzing and instrumenting this kind of code is outside the scope of this work. While we make no guarantees about what native code may do if invoked, our system can detect and prevent the invocation of native code in rewritten apps by identifying and intercepting invocations of native code. App-agnostic policies can target native methods by selecting (Section 2.2.4) methods with the `native` attribute, specifying handler behavior as they would for any other method (e.g., blocking the request, or asking the user for approval). We do not attempt to rewrite Android apps written entirely in native code.

2.3.3 Intercepting Unexamined Code

Android apps have several mechanisms for dynamically generating or loading classes, bytecode, or libraries. We include signatures for the methods that provide these capabilities to prevent rewritten apps from executing unexamined code. Policy writers can specify handlers for these methods to block or allow these requests dynamically based on runtime decisions (e.g., user approval).

2.3.4 Integration with App UI

By embedding our policies directly into our rewritten apps, we have the ability to achieve deep integration with the operation of the rewritten app. It is particularly useful to be able to integrate into the UI of the rewritten app. In Android, most operations involving the UI must be made within the main (UI) thread, with a reference to the current Context (such as the foreground Activity, Android's name for a GUI frame). These features may not be easily accessible within all method handlers (e.g., invoked from a background thread).

In order to simplify UI-based operations made from within method handlers invoked from a variety of contexts, we have implemented a small support class and set of method handlers

for various `Activity` lifecycle methods (e.g., the `onCreate` and `onResume` methods) that maintain a globally accessible model of the current Activity displayed to the user. This context can be used to execute events on the UI thread, which we use to display notifications to the user via dialog boxes, popup messages, and integration with app UI elements in our example policies described below.

We build on this to provide a general mechanism that blocks handler methods invoked from a background thread and performing some task in the UI thread before returning. Normally, dialog boxes are requested asynchronously in Android, but we used this mechanism to allow the user to make a decision about how to handle a request in our fine-grained network access control policy. As a convenience, these UI support methods are available to all policy writers, who are also free to implement their own.

2.4 Applications of Rewriting

The power, flexibility, and practicality of our rewriting system empowers users to develop and enforce a wide variety of policies in Android apps. In this section, we describe several example policy specifications we developed that demonstrate some useful applications of app rewriting. These policies take advantage of the high-level specification capabilities described in Section 2.2.4, including automatic generation of support code, such as that needed to intercept methods invoked dynamically (Section 2.3.1). In Section 2.5 we evaluate these policies by applying them to real-world apps.

2.4.1 Fine-Grained Network Access Control

Our system can be used to restrict app behavior in ways that exceed the capabilities and granularity of the controls provided by the Android platform. On Android, installing an app that requests the `INTERNET` permission grants the app unlimited access to make network requests. We developed a policy that gives users fine-grained control over the network connections an app makes by intercepting, logging, and blocking unapproved network operations.

Target Methods To intercept app network requests, our policy targets the methods associated with the `INTERNET` permission as identified by the PScout [13] project.

Handler Behavior The handler methods check the destination of the network request

against the user-configurable whitelist and blacklist. If it matches either list, the request is automatically allowed or denied, respectively. When applying the policy, the user can choose the default behavior when a request doesn't match either list: allow, deny, or ask the user when applicable. The handler uses the mechanisms described in Section 2.3.4 to present the user with a dialog box for runtime decisions. If the network request is attempted in a background thread, the background thread is blocked until the user responds to the dialog window presented in the UI thread.

When denying a request (either automatically by policy or by user decision), the handler throws an appropriate exception (e.g., `java.io.IOException`), or returns `null` if the target method would not normally throw an exception. All requests are logged to Android's built-in logging system `logcat`, along with the allow/deny policy decision applied to the request, and a rewritten app notifies users of background requests via `Toast` alerts (simple, brief, noninteractive popup messages).

2.4.1.1 Interception behavior based on call-site

Our system also allows policy writers to select and apply transformations to only a subset of the app code. One use for this functionality is to apply different policies to original app code than to particular third-party library code. Policy writers can specify classes or packages directly, select them using the declarative interface described in Section 2.2.4, or as determined by an external analysis tool. To demonstrate this functionality, we include a setting in our network access control policy to (optionally) intercept only network requests made from within code in the Android Admob advertising library package.

In addition to statically identifying and rewriting regions of an app differently, policy writers can also make policy decisions (e.g., to allow a request or not) *dynamically* based on the run-time call stack. We include another option in our network access policy to deny network requests made from Android Admob advertising library code by examining the run-time call-stack data.

2.4.2 HTTPS-Everywhere

Many popular web services support both HTTP and HTTPS connections. Unfortunately, many of these services, and apps that use these services, default to the HTTP version of these sites.

This can leave users vulnerable to sniffing, session-hijacking attacks like Firesheep [18], SSL-stripping and other attacks. We have developed a policy that protects users from some forms of these attacks while preserving app functionality.

The HTTPS-Everywhere [29] project provides browser extensions that rewrite HTTP web URLs to HTTPS alternatives for many popular services. This URL rewriting often involves more than simply replacing `http` with `https` in the URL. For example, some sites serve static content from Amazon's S3 storage service. While original domain may only serve content via HTTP, the same content can be retrieved via HTTPS from the Amazon S3 servers. The HTTPS-Everywhere contributors maintain a set of manually-specified rewriting rules for many major websites that map resources offered over HTTP to known HTTPS alternatives.

HTTPS-Everywhere is implemented for browsers, where there are well-defined extension systems capable of rewriting URLs. However, this does not help Android apps that make requests directly to these web services, and to our knowledge no such feature is available for Android apps. Researchers such as Fahl et al. [34] show that many Android apps fail to use HTTPS even when it is available. We can use our rewriting system to produce apps that use the HTTPS version of URLs when available using the URL-rewriting rules from the HTTPS-Everywhere project.

Target Method Handlers We have ported popular URL-rewriting rules from the HTTPS-Everywhere project into rewriting rules for our policy. This policy intercepts many platform methods for creating network connections to a URL. The corresponding handlers replace (at runtime) the destination with the HTTPS alternative when it matches an applicable HTTPS-Everywhere rule.

2.4.3 Automatic App Localization

Our rewriting system can also be used to augment apps with new behavior not considered by the developer of the original app. We have created a policy that performs a type of localization and internationalization automatically in existing apps by translating text in app UI widgets via an online web translation service.

Target Methods This transformation policy targets the `setText` methods in all Android UI widgets based on `TextView`, such as `Button` and `CheckBox`. One benefit of our

approach, as opposed to simply translating all strings found within an Android app, is that our rewritten apps will also translate content generated dynamically, or retrieved from external sources such as the network. Our policy intercepts calls that take String values as well as string asset resource identifiers.

Method Handlers Whenever the original app would normally set the text in one of these widgets, we intercept these calls and translate the text via a web-based translation service, updating the widget with the translated text. Apps set the text in UI widgets from within the Android UI (main) thread, so rewritten apps will invoke our handlers in the UI thread. Regular `setText` calls finish quickly, but delaying the UI thread while translating each string via network request would impact the responsiveness of the app. Our system allows us to specify handlers that handle this gracefully by performing the translation in the background.

Asynchronous Design In order to maintain UI responsiveness, our handlers return immediately after creating a new background task to perform the translation. After receiving the translation from the web service, this background task updates the widget with the new text asynchronously, queuing the update task to run back in the UI thread. This means that the app will remain responsive to user and other events while the translation occurs, and the UI will be updated asynchronously as translation results are received.

Our current implementation performs the translation using a simple web service we have created to test our functionality, though the handlers could be updated to function with an existing provider such as Google or Bing. Of course, the translations will only be as good as can be provided by the translation service. We note that when applying a transformation policy that includes network requests, the rewritten app must include the Android `INTERNET` permission, so this permission must be added if it is not already included in the original manifest.

2.4.4 Automatically Patching Vulnerable Apps

Our system can rewrite apps to replace system functionality in apps, such as by replacing calls to deprecated or device-specific APIs with alternatives. For example, we created a simple policy that replaces calls to some of the wallpaper API deprecated in Android version 2.0 to the equivalent functionality provided by the new `WallpaperManager` in recent Android versions. We applied this policy to apps written for the old API and observed that the rewritten

apps used the new API.

Beyond simple API replacement, our system can provide protection against some types of bugs and vulnerabilities in the Android platform. Generally, serious issues discovered in the Android platform are addressed in a later version of Android, but many users are unwilling or unable to receive and apply platform updates to their device (for example, due to lack of carrier or manufacturer support). In a report released by Google [38], more than half of the devices running an Android device with platform version 2.2 or newer and running the Google Play app are using a version of Android older than version 4.1.x (older than the three most recent API versions). Some issues have workarounds app developers can include in their apps to protect users running outdated Android versions, but if apps do not incorporate these workarounds, users are left vulnerable.

Recently a post on the Android Developers Blog [50] disclosed that apps using "the Java Cryptography Architecture (JCA) for key generation, signing, or random number generation may not receive cryptographically strong values on Android devices due to improper initialization of the underlying PRNG. Applications that directly invoke the system-provided OpenSSL PRNG without explicit initialization on Android are also affected." At the time of discovery, this vulnerability had already been exploited to steal bitcoins from users of vulnerable apps, and Symantec reported [66] that an estimated 320,000 apps used SecureRandom in the same way that introduced vulnerabilities into the exploited bitcoin wallets.

We have developed a policy that automatically protects users of apps on vulnerable devices by inserting in-app workarounds for these PRNG vulnerabilities. Our transformation policy intercepts app launch and if necessary, performs the appropriate seeding of PRNG appropriate for the version of Android running on the device.

2.5 Evaluation

2.5.1 Evaluation Set Selection

We evaluated the functionality and impact of our system by applying our transformation policies to the top real-world apps on Google Play. We crawled the Google Play app store identified the top 60 free apps in each of the 35 categories (as of December 2013), and downloaded

each unique Android application package (APK) file available to us (not restricted by region or hardware requirements enforced by Google Play). Of these, we attempted to install and run each APK on an Android emulator, eliminating APKs that failed to install or launch, crashed with uncaught exceptions, contained add-on content rather than independent apps, required proprietary libraries not available on the Android emulator, or contained invalid bytecode. We used the 1122 apps that remained (executed in our test environment) as our evaluation set.

2.5.2 Rewriting Real-World Apps

We rewrote and tested each of the 1122 apps four different times: once for each of the four general transformation policies described in Section 2.4. In this section, we abbreviate the names of the four general transformation policies as follows:

- **Network**: fine-grained network access control
- **HTTPS**: apply HTTPS-Everywhere rules to replace HTTP requests with HTTPS equivalents when available
- **Localization**: translate UI widget content via web service
- **PRNG-Fix**: patch apps to apply workaround for PRNG platform vulnerabilities

2.5.2.1 Rewriting Evaluation

We applied each of our four general sample transformation policies to each app to produce four sets of rewritten apps. Success at this stage means that the support and handler methods were able to be compiled and integrated into the new app, producing a complete, installable Android application package file (APK). Our system successfully applied each of the four policies to every app in our evaluation set except for one. The Dalvik bytecode format has a hard-coded limit on the maximum number of methods, and this one app contained so many methods that it was very near to this limit. Our system detected that inserting all necessary handlers would exceed this Dalvik limit and aborted the rewriting of this app and provided a notification to the user. It is possible to work around this Dalvik method limit by splitting app bytecode into multiple files loaded dynamically, though we have not automated this approach in our current implementation.

Policy	Failure		Success	
	#	%	#	%
Network	2	0.2%	1120	99.8%
HTTPS	28	2.5%	1094	97.5%
Localization	19	1.7%	1103	98.3%
PRNG-Fix	17	1.5%	1105	98.5%

Table 2.1. Rewritten apps successfully run (out of 1122 total)

2.5.2.2 App Usage Evaluation

We tested rewritten apps by installing and running them in an Android emulator, using an automated testing tool that exercises the functionality of the app by randomly selecting and interacting with app UI widgets. In these tests we consider the test of a rewritten app to be successful if the rewritten app installs, launches and executes under our automated interaction without crashing or throwing uncaught exceptions. While we do not attempt to achieve full code coverage, during testing each successfully rewritten app executed at least one of our added handler methods. General techniques for automatically verifying the behavior of Android apps is an open and difficult problem [61] and may depend on the desired outcome of rewriting. For example, should blocking network access in an app that requires it be considered "breaking" the app?

All four of our transformation policies had a success rate of over 97.5%. Rewritten apps threw errors when they handled the new behavior poorly, such as checking UI widgets for fixed string patterns (failing to match when translated), or assuming all requests will be made over HTTP (and failing to follow redirects across protocol boundaries, such as when requests are replaced with HTTPS locations). See Table 2.1 for the exact results for each policy.

2.5.2.3 Transformation Policy Evaluation

In addition to evaluating the rewriting and execution of each app, we verified the invocation and functionality of the handler methods by monitoring the behavior of the Android device emulator when running each rewritten app.

Network For the network policy, we observed the interception of network requests, dialog

boxes presented to the user, and policy decisions applied to network requests.

HTTPS In our policy we include the URL-rewriting rules from the HTTPS-Everywhere project for several popular sites, including AdMob, Blogger, Google App Engine, Foursquare, Doubleclick, Reddit and related domains. After applying our policy to each app, we ran each rewritten app in an emulator and logged the network requests made by the app. We found that 284 of the 1122 apps made at least one request to a HTTP URL matching one of our URL-rewriting rules. We used the network request log to confirm that each of these requests were replaced with requests for the corresponding HTTPS URL.

Localization Our localization policy is designed to use online translation services with APIs like those provided by Google or Bing. To simplify testing and avoid being tied to any particular provider, we created a simple web service that provides translation functionality by receiving a string via `POST` message and returning the translated string. For easy verification of policy behavior, our test service returns strings enclosed in sentinel prefix and suffix values. We observed our transformed strings in the expected widgets in the rewritten apps, and confirmed the operation of our transformation policy by logging the requests made to our web service.

PRNG-Fix We verified that the system PRNG was initialized and seeded on launch of each app rewritten with this policy. We confirmed that each of these apps recorded the execution of the handler applying the PRNG workaround in the system `logcat` log.

2.5.2.4 Impact on App Code Size

We add our handler code as separate methods, rather than injecting our custom behavior inline at every point the original application invoked a target method. This means that even if handlers are complex and large, they are only included in the app once. Whenever the original app invokes a target method, we replace this single instruction with the few instructions (generally one) necessary to invoke our handler instead (depending on the invocation type), so the increase in app size due to these replacements in minimal. The primary source of increases to bytecode size is the number and complexity of handler methods in the policy.

Android apps are packaged in APK files, which contain all of the bytecode and assets for the application. Inside this compressed archive, the `classes.dex` file contains all of

Policy	Mean Increase	
Network	45.9 KiB	1.8%
HTTPS	14.0 KiB	0.56%
Localization	43.3 KiB	1.7%
PRNG-Fix	11.2 KiB	0.45%

Table 2.2. Mean impact on size of uncompressed `classes.dex` (compared to original 2,492 KiB)

the Dalvik bytecode for the application. We measure the impact of the policies described in this chapter on the size of an app by measuring the size increase of the uncompressed `classes.dex` file. We note that the `classes.dex` file is compressed in the rewritten APK, further minimizing the impact of our added code.

Table 2.2 lists the mean increase in size of the uncompressed `classes.dex` file when rewriting an app. The mean size of `classes.dex` files in our evaluation set is 2,492 KiB and we observe that applying each of our rewriting policies results in only a minor increase in total bytecode. The bytecode added scales linearly with the number of target methods in the policy and the sizes of the method handlers. This means that a policy that has little variance in the number of target methods specified or handler sizes will add approximately the same amount of bytecode to each app.

Table 2.3 lists the size analysis on the complete APKs produced by our rewriting system. These APKs are the actual files that are deployed to Android devices. We discovered that because several APKs in our evaluation set had been poorly compressed, our system actually produced some rewritten APKs that were smaller than their originals. To discount the effect of different compressions, Table 2.3 compares the sizes of APKs before and after rewriting, both of which were compressed in the same way by us.

2.5.3 App Performance

The impact of our modifications of the performance of a rewritten app depends greatly on the behavior of the handler methods. However, the only overhead required by our design is the addition of one new method call for each target method invocation in the original app. In [27]

Policy	Mean Increase	
Network	13.08 KiB	0.13%
HTTPS	5.08 KiB	0.049%
Localization	10.43 KiB	0.10%
PRNG-Fix	4.89 KiB	0.047%

Table 2.3. Overall impact on APK size (compared to mean size of recompressed original APKs: 10,317 KiB)

we describe our strategy for method interception and perform a microbenchmark test to measure the overhead of adding an additional method invocation for each target method call. We applied a very simple transformation policy that intercepts the `StringBuilder.append` method, with a handler that only invokes the original method, to an app we created that performs one million appends. We ran the original and rewritten versions of the app on a HTC Thunderbolt phone running Android 2.3.4. We measured the run time of each and in this test executing our method handler for each target method invocation adds an average of less than 0.2 microseconds per call. Due to the minor impact on resource usage, this overhead is also unlikely to make a measurable impact on power consumption on the device. Rewritten apps only pay this performance penalty when executing a target method, as the remainder of the app code is left unmodified.

While the inherent overhead of our method interception is low, the overall impact on app performance depends on the functionality of the handler methods and the target methods intercepted. The overhead added by many of our handlers is marginal relative to the original target method call (e.g., logging a network request method). When more time-consuming handlers are required, our deep integration into the app allows for sophisticated policies that minimize the impact on app performance (e.g., our localization policy translates UI text in background threads and updates the UI asynchronously). In general, well-written method handlers will not impact the app performance more than if the original app had implemented the desired behavior. Policy writers can weigh the performance impact against the value of the added functionality.

2.5.4 Rewriting Performance

Apps can be rewritten quickly on a normal desktop PC. On average, a normal desktop machine with a 2.66 GHz Intel Core 2 Quad CPU (Q9450) with 4 GB of RAM our system can rewrite an APK in approximately 5 seconds. This is the time required for the entire end-to-end rewriting process, which includes disassembling the original app, analyzing the contents of the target app, generating the Java source for all target method handlers, compiling the handler code, rewriting the app bytecode and merging in the handler code, reassembling and signing the rewritten app. The bytecode rewriting is performed in a single pass, which scales linearly with the size of the app bytecode.

2.6 Discussion

2.6.1 Transformation Policy Development

While users may develop transformation policies themselves, it seems more likely that most users will use community developed, vetted, and maintained policies, similar to the communities that have grown to produce and maintain policies for Adblock Plus [1], NoScript [8], HTTPS-Everywhere [29], and similar security-focused browser extensions. Policy writers can use static and dynamic techniques to analyze the Android platform (such as those described in by Felt et al. [35]) to ensure proper coverage of functionality across the Android platform.

Our system only requires handler methods to declare the same checked exceptions and return type as the target methods, leaving policy writers with the power to modify the state and operation of the app in dramatic ways. This allows for powerful transformations to app behavior, but policy writers must carefully consider the effects of their modifications. It is possible for transformation policies to change the internal state of an app in ways the app does not expect or handle gracefully (e.g., by throwing an unexpected runtime exception). We leave it to the policy writers and users of the rewriting system to decide what changes to app behavior are appropriate and necessary to achieve their rewriting goals.

2.6.2 Transformation Policy Application

Our system could be deployed in a number of ways. Users may download Android apps and run the rewriting tools themselves to produce a new app to install on their device. Alterna-

tively, this rewriting system may be provided as part of an online service that rewrites Android apps on the fly as requested by a user. Corporate environments may rewrite apps before allowing installation on a device (e.g., via a proxy as apps are downloaded to the device, or by providing a private "market" of rewritten apps).

Imagine an enterprise that requires all apps to be rewritten before being installed on a corporate device. This rewriting process could allow the enterprise to mitigate the risk of installing apps developed by untrusted third parties. This enterprise could provide a trusted rewriting service that applies a trusted transformation policy to any arbitrary app requested by a user. This trusted policy can be applied to many apps, so it seems more feasible for an enterprise to maintain a single trusted transformation policy than to develop all of the apps that users desire.

Android apps include a digital signature from the developer over the contents of the app. Rewriting the app changes the content, so the original signature is not valid for the rewritten version of the app. Our rewriting system signs each new rewritten app, so the recipient can verify that the rewritten app was created by a trusted rewriting service and was delivered by without tampering.

Rewriting requires just a few seconds on a normal desktop PC, which is fast enough to support on-line rewriting before installation on users devices. While our current rewriting system runs on traditional PCs, it may be possible to perform the rewriting entirely on the Android device itself.

2.6.3 Legal and Ethical Discussion

We designed RetroSkeleton to be a powerful tool to enable flexible rewriting of apps. Like many other tools, RetroSkeleton may be used for a variety of purposes. While many uses benefit app markets, developers, and users, some may not. The legality of reverse engineering, modifying, and redistributing apps depends on the jurisdiction and contract (e.g., EULA) of the involved parties. Technically savvy users could disassemble, modify, and repackage Android apps without RetroSkeleton, so all the operations performed by our system could be applied manually when RetroSkeleton is unavailable. Furthermore, many effects of RetroSkeleton can also be achieved by other means, such as blocking network requests with a

proxy instead of the Fine-Grained Network Access Control policy described in Section 2.4.1.

Currently, even markets that vet apps before admission make only a binary decision between accepting or rejecting an app. Markets could improve by using automated app rewriting to enforce policies on submitted apps, thus acquiring an additional level of control over the quality of the apps that they distribute to their users. Developers may benefit if app rewriting adds functionality that their users desire without having to implement those features themselves, or in ways that their users find more trustworthy. Users may be more likely to install an app if it has been rewritten using a policy provided by a trusted party to give them additional control or guarantees about its behavior.

On the other hand, rewritten apps may restrict behavior that developers or ad providers depend on for revenue (e.g., delivering targeted advertisements). As a result, developers may refuse to distribute their apps on markets that rewrite apps. Users of rewritten apps may not know if undesirable behavior in apps results from rewriting.

While developers may not bypass the interception mechanisms of RetroSkeleton, they are free to detect modifications and change the behavior of their apps (e.g., refusing to run) when rewritten. We make no attempt to hide our modifications from rewritten apps (Section 2.1.1). Developers can easily check if their apps have been rewritten, e.g., by comparing a checksum of the code with the expected value. Also, rewritten apps are signed by the rewriter, rather than the original developer.

2.7 Comparison to Alternative Approaches

2.7.1 Modifying the Android Platform

Android includes a permission system that limits apps to categories of functionality declared at install time. Android apps declare the coarse-grained categories of functionality they wish to use in their application manifest. If the user installs the app, the app is granted unlimited access to all data and functionality requested in the manifest. These permissions are very coarse-grained (e.g., a single `INTERNET` permission for network access), and once installed, the user cannot control or constrain what the app does with the granted permissions.

Finding the Android permission system lacking, many researchers modified the open-

source Android platform to develop custom builds of Android [14, 46, 58, 73]. Many of these, such as Apex [58] and TISSA [73], are designed to improve the security and privacy controls on Android, modeled around the permission-based methods. While the Android platform is open source, many drivers for hardware support are not, and installing custom Android builds may require rooting the device, voiding the warranty, and/or violating the carrier's terms of service. This makes it infeasible for most normal users to update or modify the firmware or platform to provide additional controls or functionality.

It is also difficult for these custom builds to keep up with new versions to the Android platform, and to support all hardware devices, as open-source versions of device drivers may not be available. In contrast, using our system rewritten apps may be deployed to any stock Android device. In addition to these deployment issues, policies integrated into the platform are far less flexible than our app-specific approach. It is relatively difficult to extend platform-based policies to add new capabilities, and policies in the platform will apply to all apps on the device. Our approach can apply different policies to different apps, and we can easily add and apply new policies at any point.

2.7.2 Single-Purpose Rewriting

Some researchers use single-purpose rewriting to apply a single type of policy to Android apps. For example, Jeon et al. [49] developed a system to provide finer-grained permissions for Android. Their work is based on a specialized replacement for some privacy-sensitive APIs and use Dalvik bytecode rewriting to modify apps to use their replacement API. Their replacement APIs fulfill the requests using inter-process communication to shuttle the request to an independent service, also installed and running on the device, which contains all permissions. Similarly, Reynaud et al. [62] discovered vulnerabilities in Google In-App Billing by rewriting applications that communicated with the Google Market app to bind to their own fake Market app instead.

As a fully generalized app rewriting system with a powerful policy specification language, our system has a number of advantages. We believe that RetroSkeleton is general and flexible enough for future researchers to use to perform investigations like these without having to develop independent, specialized, single-purpose rewriting systems. Additionally, provid-

ing a powerful policy specification interface frees policy writers from having to understand exactly how the rewriting is performed. For example, our system will generate the handlers for dynamic dispatch via reflection for any given policy without any action required by the policy writer. We feel this separation of concerns better supports policy development and maintenance.

2.7.3 Other Android-Specific Approaches

Xu et al. [69] have developed Aurasium, which provides reference monitor capabilities by repackaging Android apps to use a custom version of libc. Their system can enforce security policies by interposing on low-level system calls, and performing their security checks from within the replaced libc call. In contrast, our approach allows policy writers to specify what they want to intercept at the method call level, rather than requiring knowledge of how these operations manifest themselves as low-level libc calls. Furthermore, policy writers specify the new behavior they wish to add as Java code that operates on the Java objects passed to their target method, rather than on the low-level data available within libc functions. Aurasium is designed for reference-monitor capabilities, and we can achieve similar results by intercepting calls to sensitive methods, such as those that require the `INTERNET` permission. However, RetroSkeleton can be used for many other ways, such as app UI augmentation or behavior modification like our auto-localization functionality, which would be far more difficult to achieve from within a replaced libc.

Adblock Plus for Android [2] provides a mechanism for blocking many ads from being retrieved by Android apps. Adblock Plus for Android works by installing an Android service on the device that runs a local proxy server in the background at all times. Then the Android device is configured to use the Adblock Plus local proxy server as the proxy for network connections. This local proxy server blocks requests to known ad services. Our rewriting system may be used to similarly provide ad blocking capabilities, with a number of advantages. First, we can apply different ad-blocking policies for each app, while the Adblock Plus proxy cannot distinguish between call sites. We can rewrite and redistribute apps without Adblock Plus's requirements of a system with manual proxy-specification support (Android 3.1 or newer, custom Android build or a rooted device). Rewritten apps also do not require a proxy service

running in the background consuming resources as we can integrate the blocking decisions into the new app. Our system can also provide more advanced blocking, such as blocking network requests from ad libraries but permitting requests made from within the main app, as discussed in Sections 2.2.4.3 and 2.4.1.1.

2.7.4 Java-based Approaches

While Dalvik bytecode differs from Java bytecode, Dalvik bytecode is created from Java. As a result, tools such as ded [30] and dex2jar [5] have been created to convert Dalvik bytecode into Java bytecode. This approach allows for the use of Java-based analysis tools such as WALA [12] on the resulting Java bytecode. This conversion is nontrivial and frequently produces code that can not be assembled back into a functional Android app, as reported in [62]. While this matters less in pure analysis-based studies, our goal is to produce valid Android apps after rewriting so we operate on the Dalvik bytecode without conversion.

While the Dalvik and Java virtual machines differ in many ways (e.g., register-based instead of stack-based), some techniques used to rewrite Java bytecode apply to our approach to Dalvik rewriting. Java bytecode instrumentation by Chander et al. [19] involves invocation and class substitution to produce modified Java programs. While our rewriting system works on Dalvik rather than Java bytecode, our stub and wedge class approach for low-level method interception is similar to their Java-based system with respect to the class hierarchy of the rewritten software. RetroSkeleton uses this low-level mechanism to perform aspects of the method interception, and provides a new high-level abstraction that allows policy writers to create policies that can be applied to any app. Without a system automating this work, rewriters must identify and create new and different wedge classes for each rewritten app, depending on the classes used and defined by the original app developer. RetroSkeleton analyzes the app and target methods in the policy to automatically generate the appropriate stub and wedge classes with the necessary properties, and modifies the app class hierarchy as needed. This automated analysis and generation is critical for real-world use given the tremendous number and variety of Android apps available.

Erlingsson and Schneider used inline reference monitors (IRM) for enforcing security properties in Java applications [32]. The Java IRM design detailed in [31] highlights the

many benefits of integrating code into the target application itself, such as easier observation of the internal state of the application. Because our approach embeds all changes entirely into the rewritten apps, our system provides these same benefits to policy writers. Because RetroSkeleton can interpose on method calls of interest, it can be used to embed reference monitor functionality into Android apps, but this is only one potential application of our system. Our system allows policy writers to just as easily write policies that add functionality other than security checks, such as adding new UI behavior.

Chapter 3

Cross-Application Information Flow Tracking via Databases

An administrator of a web service is responsible for protecting the data in the systems, the users of the system, and the service itself. The ability to monitor and manage the way data moves through a system is particularly important for web services, as these services often make extensive use of user-submitted content. Improper use of untrusted input in web services can result in attacks on the service (e.g., SQL-injection) or on visitors to the site (e.g., cross-site scripting), or other violations of data confidentiality or integrity requirements. In general, it is increasingly difficult to identify and monitor the ways data may flow through a web service as these services become more popular and complex, especially as many organizations find themselves administering third-party or legacy systems not designed to provide the level of insight they require.

Information flow tracking is a powerful approach for observing and controlling the use of data in a live system, which seems like a natural fit for the problems faced by administrators of web services. The ability to mark and track data from untrusted sources as it moves through the service can be an invaluable way to monitor and protect complex services from attack. Unfortunately, while there are many information flow tracking systems, most are designed to run only a single program at a time. These single-application information flow tracking systems are unsuitable for most real-world web services, as these services are usually composed of at least one web application and a database server. All information flow tracking metadata is lost

when the data crosses the boundary between the application and the database. Many system-wide information flow tracking systems are too coarsely-grained to track individual entries in databases, while others pay significant performance costs for tracking all memory operations and still lack the support of database semantics. Most existing web services were developed without information flow tracking in mind, and solutions requiring rewriting each application or changing the database engine seem infeasible for widespread adoption in real-world systems.

To address these problems, we created DBTaint, which is a system that provides cross-application information flow tracking via databases. DBTaint allows administrators to leverage existing single-application information flow tracking systems in their multi-application services by propagating this metadata into and through their databases, with full support for database semantics. Our system is designed so it can be applied to existing, real-world web services without requiring any changes to the web application, and only SQL-level changes to the database (no database engine modifications). DBTaint augments the database interface to performs automatic rewriting of SQL statements, supplying information flow tracking metadata ("taint" values) when sending data to the database, and tainting data from the database appropriately before returning it to the application. Rather than enforcing a specific policy, our framework propagates whatever information flow metadata it is given, allowing administrators to use whatever policy is most appropriate for their setting. We have evaluated our system by applying it to real-world web services and discuss the unique insights our cross-application system design can provide. In this work we address a number of challenges and design and implement DBTaint with the following properties:

- efficient end-to-end taint tracking through applications and databases
- full support of database semantics
- deployment is transparent (requires no changes) to the web application
- requires only SQL-level changes on the database server
- support for Perl and Java applications and the PostgreSQL database

3.1 Background

A primary application of information flow tracking is to protect a program from malicious input. The system identifies and marks data from untrusted sources as "tainted," tracks this taint metadata associated with the data as it flows through the system, and detects when tainted data is used in security sensitive contexts, such as the return addresses of function calls or the parameters of risky system calls [59, 70]. Information flow tracking mechanisms can be grouped into two categories based on the scope of tracking capabilities: application-wide and system-wide. The former tracks information flow within the same application [59, 70], while the latter tracks information flow in the entire operating system [65].

The popularity of web services has made them highly attractive targets to attackers, and unchecked malicious input can lead to some of the top reported software vulnerabilities in web applications [9]. For example, Cross-Site Scripting (XSS) attacks remain one of the top categories of software vulnerabilities [10]. Information flow tracking is a logical approach for preventing these attacks by tracking malicious input [41, 56]. However, the application and effectiveness of information flow tracking is limited by the only two types of current mechanisms: single-application and system-wide tracking.

A typical web service consists of multiple applications, such as a web application, which implements business logic and generates web pages, and a database, which stores user and application data. In multi-application settings like web services, single-application information flow tracking is inadequate, as it forces web applications to decide between treating all the results of database queries as tainted or treating them as untainted. Inevitably, this results in false positives or false negatives when the database contains both tainted and untainted data.

To enable cross-application information flow tracking, one might resort to system-wide information flow tracking systems. However, there are several problems with these systems. Many either lack the granularity necessary for web services (e.g., tracking process-level taint values), while others track more information than needed for protecting web services, which comes at an unnecessary performance cost. Protecting against XSS attacks requires tracking information flow only in the web application, database, and the information flow between them, rather than in every operation in the entire system. Also, these system-wide mechanisms

generally fail to take advantage of application semantics. Without the high-level semantics, these systems cannot properly perform taint propagation throughout complex database operations.

We introduce DBTaint, a system that provides information flow tracking across databases to enable cross-application information flow tracking in web services. DBTaint extends the database to associate each piece of data with a taint tag, propagates these tags during database operations, and integrates this system with existing single-application taint tracking systems. By providing this integration via SQL rewriting at the database client-server interface, DBTaint is completely transparent to web applications. This allows developers to use DBTaint with existing, real-world legacy applications without modifying any web application code.

DBTaint can provide more accurate information flow tracking than single-application taint tracking systems. Services using DBTaint will have fewer false positives than systems that consider all values from external applications (like databases) as tainted. Similarly, services using DBTaint will have fewer false negatives than systems that consider all values from the database as untainted. Furthermore, since DBTaint tracks taint propagation inside the database, it takes advantage of database semantics to track taint propagation accurately. Besides providing a more accurate taint tracking system, DBTaint can also be used to detect potential vulnerabilities in applications. If user input is untainted during sanitization in a web application, inspecting the taint values of database columns may reveal subtle security vulnerabilities. Our insight is that if a column in the database contains both tainted and untainted data, it may signal incomplete sanitization in the database client, e.g., when user input is sanitized only on a subset of program paths. Such observation should alert the programmers to audit the sanitization functions in the program carefully.

We make the following contributions:

- We design and implement a system that tracks information flow across databases. This allows cross-application information flow tracking to protect web applications from malicious user input.

- We improve on single-application taint tracking systems by reducing the number of false positives and false negatives and improve on system-wide taint tracking systems

by tracking only the taint propagation that matters to the target application and by taking advantage of database semantics to improve tracking accuracy.

- Our system is also useful for analyzing certain behavior of database client applications, such as identifying potential incomplete sanitization in web applications.
- We design a flexible system to integrate single-application taint tracking systems with the PostgreSQL database. This system allows legacy applications to take advantage of our system transparently.
- We implemented two implementations of DBTaint that work with real-world web applications written in Perl and Java. These implementations, despite lack of optimization, have low performance overhead.

3.2 Design

DBTaint is a system that allows developers to track information flow throughout an entire web service consisting of web applications and database servers. DBTaint provides information flow tracking in databases and integration with single-application information flow mechanisms. The system is completely transparent to the web applications, which do not need to be modified in any way to take advantage of DBTaint.

We assume that the developer is benign, and has the ability to replace the database interface and database datatypes used in the web service with the modified (DBTaint) versions. We also assume that the single-application taint tracking engine(s) used for each individual web application appropriately taints input from unsafe sources (e.g., user input).

DBTaint propagates the taint information throughout the multi-application system, but does not attempt to actively prevent the program from operating unsafely. Rather, by propagating and maintaining taint values for each piece of data in the system, we provide developers with the information needed to perform the sink checking and handling appropriate for their setting. Although DBTaint was motivated by web services, our implementations provide cross-application information flow tracking to any multi-application setting where applications communicate via databases. For brevity, we will refer to these applications as web

applications onward.

3.2.1 Taint Model

3.2.1.1 Soundness

DBTaint helps improve the security of web services by tracking the trustworthiness of all the data values used by the service. DBTaint marks each value as either *untainted* or *tainted*. DBTaint marks a value as *untainted* only if it can determine that the value is trusted. Therefore, when DBTaint marks a value as *tainted*, it could be because DBTaint has determined that the value is indeed untrusted or because DBTaint cannot determine whether the value is trusted.

We say two values are in the same *context* if they belong to the same column or their respective columns are compared in a `JOIN` operation. With DBTaint, the database marks an output value as untainted only if there was an occasion when the same value in the same context was marked as untainted when it entered the database, or if the output value is derived from untainted values only.

The above property implies that:

- When a context contains two identical values but one is tainted and the other is untainted, DBTaint may return this value as either tainted or untainted. Our implementation chooses to always return this value as untainted to improve the accuracy of taint tracking in the web application.

- DBTaint will never return a value as untainted if this value has never entered the context as untainted and is not derived from only untainted values.

3.2.1.2 Scope of Taint

DBTaint works with any taint tracking mechanism inside the web application by propagating the taint information associated with each value into and through database operations and back. The taint value DBTaint associates with the result of a database operation depends on the data directly related to the result. In other words, our system performs information flow tracking of explicit flows through database operations. This explicit-flow philosophy matches those used in many other dynamic information flow tracking systems, including the Perl and Java taint-tracking systems we use in our implementation.

To better understand the motivation for this design, imagine querying the database for a piece of data associated with a specific identifier, and that this data was marked as untainted when it was inserted into the database. If the database checks a different row containing a tainted value, determines it does not match the desired identifier, and continues on until it finds the correct row containing the untainted value, we believe DBTaint should not mark the requested value as tainted when returning it. Our goal is to propagate the taint information received from the web application back to the web application. Because queries are often made over many or all elements in a database table, incorporating taint values of data related to the execution of a query but unrelated to the results (e.g., an implicit flow) could lead to massive over-tainting of results in situations like these. As mentioned in Section 3.2.1.1, DBTaint will never untaint tainted values and will only return an untainted value if there are no explicit flows from tainted values to the result.

In the simplest and most common case, the taint value of a datum is the same when it exits the database as when it entered the database. Furthermore, when a query is made for the `SUM` of a column the result is marked tainted if any of the values added together were tainted, as there is an explicit flow from each of these values through the computation to the result. In contrast, consider a query for the `MAX` of two values in the database, where one value is 3 and tainted and the other value is 5 and untainted. While a comparison is performed, there is no explicit flow from the 3 data value to the result as it ultimately did not "contribute" to the final value. DBTaint returns the value 5 to the database client (the web application) untainted, because the value 5 was untainted when it entered the database and it is the only value with an explicit flow to the result of this query. Similarly, data values in the result of a `JOIN` query carry their taint values in the database, regardless of the taint values of other data (e.g., data in the common columns during `JOIN`) that may have affected the selection of the data in the result.

3.2.1.3 Backward Compatibility

We adopt the principle of "backwards compatibility," similar to the one described by Chin and Wagner [20], and design DBTaint such that a DBTaint-unaware application should behave exactly the same regardless of whether it is retrofitted with DBTaint. Under this principle, when

DBTaint compares two data, it ignores their taint values. Besides ensuring backward compatibility, this principle also allows DBTaint to set the taint value of certain output data more accurately. For example, consider computing the `MAX` of a tainted value 2 and an untainted value 2. Either value is an acceptable result for this query, but we choose to return the untainted value. Similarly, in a `SELECT DISTINCT` query, we again prefer to return untainted versions of equal values when available.

3.2.2 Information Flow Tracking in the Database Server

Because current mainstream database systems do not natively provide a mechanism for storing taint information associated with each piece of data, DBTaint provides a mechanism for storing this information without losing precision of the original data. Furthermore, DBTaint propagates the taint information for database values during database operations.

3.2.2.1 Storing Taint Data

DBTaint provides information flow tracking capabilities in databases at the SQL level, requiring no changes to the underlying database server implementation. Capabilities added to the database at the SQL-level are likely simpler and more portable than those made by modifying the underlying implementation of a particular database server. Furthermore, by using SQL to maintain and operate on the taint information, we avoid the need to provide new mechanisms to insert, retrieve and operate on taint information in the database server.

Many databases support *composite* data types, where each data cell may store a tuple of data. We used this feature to store taint information alongside associated data values, allowing DBTaint to use the well-understood SQL API for interacting with these taint values. The additional functionality (like auxiliary functions) DBTaint needs to add to the database can be done at the SQL-level as well (e.g., via `CREATE FUNCTION`).

We chose to implement our system using composite types because we feel it is the simplest of the available approaches (e.g., compared to storing taint bits in mirrored tables or additional columns). It allows our SQL-rewriting operations to rewrite each original query into exactly one new query, avoiding the need for extra queries to maintain mirrored tables. Also, using composite types allows us to build taint propagation logic into the database type system rather than into each rewritten query.

3.2.2.2 Operating on Taint Data

In addition to creating the database composite types, DBTaint provides some database functions to make it easier to work with these new types. We provide the database functions `getval()` and `gettaint()` to extract the data and taint values from a DBTaint tuple, respectively. These functions are used in the SQL rewriting phase, described in Section 3.2.4.1. DBTaint also provides necessary database functions for these composite types (e.g., equality and comparison operators). Finally, DBTaint provides functions to propagate taint values in aggregate functions (like `MIN` and `MAX`) and arithmetic operations (when one or more operand is tainted, the result is tainted).

3.2.3 Information Flow Tracking in the Database Client

DBTaint leverages existing single-application information flow tracking systems to manage taint information in the client, and integrates the single-application taint tracking system with the new database server functionality at the interface between the two applications. DBTaint works with any mechanism for taint tracking in the database client (the web application). For instance, we have implemented a version of DBTaint for Perl that uses a modified version of Perl's taint mode. We also developed an implementation of DBTaint that uses an efficient character-level taint tracking system for Java [20]. While the single-application taint engines propagate taint throughout the single application, DBTaint handles the propagation of this taint information across application boundaries when this data is used in a SQL query. Many other single-application taint tracking systems exist, and DBTaint can be easily extended to integrate with these engines as well.

3.2.4 Database Client-Server Integration

Once we can track the information flow within a single application and within a database, DBTaint must provide a way to propagate the taint information between database client applications and the augmented database server. While we could perform this by modifying the web application directly, this approach does not scale well, as the service developer or administrator would need to modify every new web application individually. Furthermore, the amount of work required to make the changes would scale with the size and complexity of

each web application.

With these concerns in mind, we take a different approach. DBTaint integrates the information flow systems of the database client and database server at the interface between these two systems. For example, Perl programs generally use the DBI (DataBase Interface) module to access database servers, and Java applications often use JDBC (Java DataBase Connectivity) API. By adding our DBTaint functionality at these interfaces, we can integrate the taint tracking systems of multiple applications completely transparently to the web application.

DBTaint requires three changes to the database interface:

- Rewrite all queries to add additional placeholders for taint values associated with the data values, and to add appropriate taint values where appropriate.
- When the application supplies the parameter values, determine and pass the corresponding taint values.
- When retrieving the composite tuples from the database, collapse them into appropriately tainted data values then return them to the web application.

3.2.4.1 Rewriting SQL Queries

In DBTaint, the database server tables are composed of composite values that contain both the data and the taint value associated with that piece of data. However, since the web applications that use these databases are not modified in any way, their data values and corresponding SQL queries do not include the necessary information to maintain the associated taint values in the database. A key component of the DBTaint system is the way the SQL queries from the web application are dynamically rewritten to propagate taint information between the web application and the database server transparently to the database client.

DBTaint performs two main types of transformations on portions of SQL queries: *tupling* and *projection*. These operations are performed on the appropriate parts of a SQL query before passing it through to the database server.

Tupling is the process of taking a data value and converting it into a tuple that contains the original value and the associated taint value. For example, when a web application sends an `INSERT` query that includes a data value to the database interface, DBTaint rewrites that

portion of the query into a tuple containing the data value and the taint value of that data. If the web application passes a parameterized query (with `?` placeholders for data values to be supplied later), DBTaint rewrites the query to include additional placeholders for the corresponding taint values.

Assume we specify a composite type using the PostgreSQL syntax: `ROW(x,y)` where `x` is the data value, and `y` is the corresponding taint value. If the web application passes the following query to the database:

```
INSERT INTO posts (id, msg) VALUES (1, ?)
```

then DBTaint rewrites this query to include the taint value of the `1` substring (e.g., 0 if untainted), and adds a place for the taint value of the message data to be supplied later.

```
INSERT INTO posts (id, msg) VALUES (ROW(1,0), ROW(?,?))
```

Projection is the process of taking a tuple value in the database and removing the associated taint value when it is unneeded. We have designed DBTaint such that using the system does not change the behavior of the web application. So, sometimes it is necessary for DBTaint to extract only the data value for certain SQL operations in order to perform the appropriate operations. For example, if the web application wishes to select rows where a specific column is equal to a hardcoded value, then we disregard the taint value during the selection process.

For example, when the web application issues the request:

```
SELECT username FROM users WHERE user_id = 0
```

Since in this case the taint value of the `user_id` field is unimportant, DBTaint extracts only the data value and the query becomes:

```
SELECT username FROM users WHERE getval(user_id) = 0
```

3.2.4.2 Rebinding Parameterized Query Values

Applications often use parameterized queries for defense against SQL injection attacks, improved performance, and maintainability. Parameterized queries use placeholders for parameters that the web application passes later. Often DBTaint must augment these queries by adding additional placeholders for the corresponding taint values. Unfortunately, this means that the index-based bindings the web application uses may no longer be valid (e.g., binding a value to placeholder three may no longer be the third parameter in the rewritten query).

Furthermore, because the web application does not know about these new taint parameters, DBTaint must provide them to the database.

When a web application attempts to bind a parameter to a particular position in a SQL query, DBTaint intercepts this request and computes the new, proper index for that data value. Then, DBTaint not only binds that data value, but the corresponding taint value, if appropriate. This allows DBTaint to support web applications that use parameterized queries with requiring the web application to have any knowledge of DBTaint's underlying implementation, including these composite types.

3.2.4.3 Retrieving Database Values

The results of database queries are tuples of data and taint values. DBTaint extracts the data values from these tuples, then taints them as appropriate in the single-application taint tracking engine used by the web application. This completes the propagation of taint values back into the web application.

3.3 Implementation

We have implemented two different versions of our DBTaint system, one for Perl and one for Java, to demonstrate the effectiveness of our approach. Both implementations are fully capable of working with real-world web services that use the PostgreSQL database engine, as we demonstrate in Section 3.4.

3.3.1 Database

Both DBTaint implementations assume the use of the PostgreSQL database server. PostgreSQL is a popular, full-featured, enterprise-class object-relational database system. Users can create composite types from base types, add custom functions, and overload operators. We leverage these features to manage the taint information stored in our modified database tables.

3.3.1.1 Composite Types

DBTaint uses *composite types* to store data and taint information in PostgreSQL database tables. A composite type is a type with the structure of a user-defined record, and can be used in place of simple types in the database. Each composite type we create has two elements: the

data value, and the associated taint value. We can maintain taint values at whatever granularity we like (e.g., per character) but to simplify our examples here we use a single taint bit for each scalar value stored in the database. PostgreSQL uses the `ROW()` syntax to specify composite type values, so we express a tainted `INT4` as the `INT4t` composite value `ROW(37, 1)`.

3.3.1.2 Auxiliary Functions

At the time of deployment, DBTaint determines all the native database types used by the web application by inspecting the original database tables' schemas. DBTaint uses the `CREATE TYPE` command to create a new PostgreSQL composite type for each of these native types. Before these composite types can be used to create new composite versions of the original database tables, DBTaint creates a number of auxiliary functions to support these new types. These auxiliary functions are used to preserve the behavior expected by the database clients, and to simplify the SQL query processing DBTaint performs at the boundary of the database and other applications.

DBTaint generates the standard comparison operators to allow the database to sort and compare composite type values, and uses PostgreSQL's operator overloading capabilities to add taint-aware versions of the the common operators (e.g., `>`, `<=`, `+`, `-`). DBTaint uses `CREATE AGGREGATE` in the PostgreSQL database to create taint-aware versions of common aggregate functions, like `MIN` and `MAX`. Additionally, DBTaint generates `length()` functions for composite types with string values, and arithmetic operators for numeric composite types. DBTaint overloads arithmetic operators to provide interoperability with base types, while propagating the taint information to the resulting composite values. For example, the operation "adding the integer 2 to the tuple `ROW(5, 1)::INT4t`" returns the tuple `ROW(7, 1)::INT4t`, which retains the taint bit from the original tuple. DBTaint also creates `getval()` and `gettaint()` functions for the composite types, which extracts just the value or taint bit for a particular piece of data. DBTaint sets the values and taint bits using normal SQL statements, and therefore requires no additional PostgreSQL functions to manipulate this data on the server.

3.3.1.3 Table Creation

After creating the necessary composite types and auxiliary functions in the database, DBTaint automatically replaces all of the simple types in the database tables with their associated composite type versions. Note that unless a web application creates new tables during operation, this table creation phase only occurs during the initial installation and configuration stage. DBTaint adapts default values, column constraints, and other table properties as needed to match the new composite types. Default values are considered "untainted" in this process. For example, DBTaint converts a column with type `INT4` and default value of `0` into a composite type column of type `INT4t` with default value `ROW(0,0)`.

3.3.2 Perl Implementation

We developed an implementation of DBTaint for web applications written in Perl that use the popular DBI module for accessing PostgreSQL databases. We use a modified version of Perl's "Taint Mode" to perform information flow tracking within the web application.

3.3.2.1 Perl Taint Tracking

Our Perl implementation of DBTaint leverages the Perl taint mode to track information flow through the web application. Perl's taint mode is an active mechanism that prevents some Perl operations from using untrusted user input unsafely. Perl taints user input, and halts when tainted values are used in certain unsafe situations (like as a parameter to `system`). We only require a passive taint tracking engine for DBTaint, so we provide a modified Perl engine that does not halt in these situations, allowing us to use DBTaint with applications not normally compatible with Perl's taint mode.

3.3.2.2 Perl DBI

In our Perl implementation we add our DBTaint database interface functionality to the DBI (DataBase Interface) module. The Perl taint mode engine we use in our implementation has a limitation: it only tracks the taint bit of the entire variable as a whole. This means that, for example, a string is either completely tainted, or completely untainted. If a web application assembles a query string by concatenating tainted and untainted data, by the time this string reaches the database interface it is impossible to determine what parts of the original query was

tainted, and what was untainted. Note that this is not a problem if we use a more sophisticated taint tracking engine, such as the one used in our Java implementation below.

However, the Perl taint mode engine is still completely sufficient for DBTaint if the application uses prepared statements for its database queries. Prepared statements are SQL statements with placeholders for parameters to be supplied later. Prepared statements are used for performance reasons, to separate SQL logic from the data supplied, and to help prevent of SQL injection, and are quite common in modern web applications. When these parameters are supplied later, the DBTaint system can inspect the taintedness of these data values at the database interface. In this way, we can consistently propagate the taint information across the boundary of the web application and the database application.

3.3.2.3 Other Modifications

The web application we chose to use with DBTaint used prepared statements for all of its SQL queries, which made the DBI rewriting relatively simple. We slightly modified the Apache-Session Perl module to use prepared statements in a way that matched the rest of the Perl application to simplify our DBI rewriting logic. We also needed to modify the Encode Perl module to avoid user values being inadvertently untainted during conversion from UTF-8 encoding.

3.3.3 Java Implementation

We developed an implementation of DBTaint for web applications written in Java that use the popular JDBC API to access PostgreSQL databases. We use a character-level taint tracking system for Java, which allows us to properly rewrite both prepared and non-prepared statements without losing any taint information from the web application.

3.3.3.1 Java Taint Tracking

We use a character-level taint tracking engine for Java. [20] This taint engine marks all elements of incoming HTTP requests (e.g., form parameters and cookies) as tainted, and propagates the taint bit throughout the web application. When these values are passed to the database interface, DBTaint rewrites the queries appropriately to propagate the taint bits between the applications. We were able to use this taint tracking engine without any special configuration or modifications.

3.3.3.2 JDBC

In our Java implementation we add our DBTaint database functionality to the JDBC (Java DataBase Connectivity) classes. The Java information flow tracking engine we use tracks taint bits on each character of each string object. With this more precise information, we no longer depend on the parameters being separate from the query to determine if they are tainted or not, allowing precise taint data propagation even when prepared statements are not used. When the database interface receives a query with literals embedded in the query string, DBTaint inspects the taint values for the characters of that literal, and then adds the appropriate taint information when tupling the value.

For example, if the DBTaint system receives the following query in the JDBC interface:

```
INSERT INTO messages (msg) VALUES ('first post')
```

DBTaint will inspect the taint values of the substring consisting of `first post`. DBTaint will then rewrite the query with the appropriate taint values based on the taintedness of the substrings. For example, if the `first post` value was tainted, the query would be rewritten to:

```
INSERT INTO messages (msg) VALUES (ROW('first post', 1))
```

We use Zql [11], a Java SQL parser, to parse the queries so they can be rewritten in DBTaint. Rewriting parameterized queries is performed using the same approach described above in the Perl implementation.

3.4 Evaluation

To demonstrate the effectiveness of DBTaint in real-world systems, we evaluate the performance of our taint-aware database operations and run two popular web services with DBTaint. We executed all benchmarks on a virtual machine running Cent OS 5 on a 2.6 Ghz Intel Core 2 Quad host with 4 GB of RAM. Our DBTaint implementations are based on PostgreSQL version 8.3.7, Perl version 5.10.0, and Java version 6 (1.6).

3.4.1 Database Operations

We first evaluate the overhead of the changes we make on the database server by comparing the performance of common database operations with and without DBTaint. In our system we

Operation	native	DBTaint	overhead
SELECT ALL	23ms	26ms	13%
SELECT WHERE	23ms	26ms	13%
INSERT row	0.5ms	0.6ms	20%
LESS THAN operator	0.2ms	2.3ms	1,050%
ADDITION operator	0.2ms	2.4ms	1,100%
EQUALS operator	0.2ms	5ms	2,400%

Table 3.1. Overhead of database operations (high overhead later shown to not dramatically impact overall performance)

implement the composite types and supporting database functions entirely in SQL. This design decision results in an implementation that is portable across all database servers with equivalent SQL-level semantics (e.g., as opposed to system-specific functionality implemented in C), though there is a performance cost for this portability. We quantify the performance cost paid by our portable implementation below.

Table 3.1 lists the average run time of 1,000 executions of each of the specified database operations and percent difference between the native and DBTaint version. The database was restarted and caches were cleared between each run in order to minimize the impact of caching on our results. We note that the composite versions of many of these operations are much slower than their native counterparts, particularly when performing operations requiring introspection and comparison of data in multiple composite values.

While DBTaint versions of these database operations do operate on slightly more data, much of the overhead we observe is due to the overhead of the SQL-level implementation. In Sections 3.4.5 and 3.4.6 we find that the impact of this overhead on a web service as a whole is relatively minor and may be acceptable in many environments. However, in environments where the performance of these operations are critical, it is possible to implement these new types and corresponding database functions in system-specific C code for improved performance.

We created and compiled an implementation of a DBTaint type and corresponding sup-

port functions written in C to confirm our hypothesis that systems-specific implementations have improved performance. The equality operator had the highest performance overhead in our evaluation of our portable implementation with an overhead of 2,400%. By comparison, when performing the same test we measured our compiled C-based implementation to have an overhead of only 7.8%. This confirms the high overhead is a result of the SQL-level implementation and while we did not complete an entire system-specific implementation of DBTaint's database operations in C, we expect such a version would have similar reductions in overhead for the remaining operations.

In the rest of our evaluation, we use only the portable SQL-level implementation. We show that even with the performance costs shown in Table 3.1, the web service as a whole incurs a relatively low overall performance penalty at a level which may be acceptable in many environments even without additional optimizations.

3.4.2 Web Application: RT

We selected Request Tracker (RT) [4], the popular enterprise-grade ticket tracking web service, to evaluate the effectiveness of DBTaint in a realistic environment. RT is not designed to be used with Perl taint mode, and was not created with DBTaint or any other information flow tracking system in mind. RT has over 60,000 lines of code, and uses 21 different database tables to store information about tickets entered into the system, users of the system, transaction history of system modifications, access control, and more. Other than installing our composite datatypes and removing the inadvertent untainting in a Unicode conversion function, we ran RT with DBTaint without making any further changes to the web application. We successfully tracked the flow of user input throughout the entire web service: from the web application, into the database, and back.

RT [4] is commonly used for bug tracking, help desk ticketing and customer service. A user creates a ticket describing the issue they wish to track, which may include a subject, text description, and attachments. In addition to this information, each ticket has a number of other properties associated with it, such as a status (e.g., "new") and a destination queue. All of this data is spread across a number of tables when stored in the database. For example, each entry in the `tickets` table includes a reference into the `users` table to identify the ticket owner

and a reference into the `queues` table for the queue for the ticket. RT draws from these many tables to provide multiple views of the tickets and associated metadata. This enables users to manage and sort the tickets in the system in many ways depending on their interest and responsibilities, such as sorting by queue or priority. A user can add and modify additional information about each ticket, such as adjusting a priority value and assigning another user to the ticket, to keep a record of the issue tracked by the ticket. Tickets may be marked as closed, which prevents RT from displaying them in certain views, but internally RT maintains the history of the changes to a ticket over time in the database.

To demonstrate that DBTaint does not alter the behavior of the web service, we recorded a series of interactions with the web service installed in an unmodified environment. We saved the database contents resulting from using the application in the unmodified environment for later reference. Then, after deploying the web application and running it in the DBTaint system, we replayed these recorded web actions in this environment. When we compared the values of the database tables in the DBTaint system with the values from the unmodified run, the only differences we observed were expected variances in values like timestamps. We observed that DBTaint allows the web application to behave exactly as it would in a normal environment, transparently providing the information flow tracking capabilities to the entire web service.

The RT application was not designed to function in taint mode, and halts immediately if taint mode is enabled in a normal Perl environment. We modified the Perl engine (see Section 3.3.2.1) to allow RT to function in the taint mode. While we did not use Perl taint mode to prevent active attacks, we analyzed the taint information in the database to learn about information flow through the web application. Note that if the web application were designed to function in taint mode, it would not need our modified Perl engine to work with DBTaint.

3.4.3 Analyzing Database Taint Values

After running RT in DBTaint, we can infer properties of the application by simply inspecting the taint values of the data in the database. By glancing through the taint values of the database records, we see that nearly all of the user input stored in the database is marked tainted. This

implies that the web application performs little input filtering, and relies on output filtering to escape characters to prevent XSS attacks. We confirmed this by inspecting the application code to find that the application does store user input directly into the database, escaping and replacing dangerous characters after retrieving these values for display in web pages.

Columns with only untainted data. Many of the database columns contained only untainted values. We observed that the values in these untainted columns were either provided or generated by the web application, rather than originating directly from user input. For example, we observed that the values in the `type` field of the `tickets` table were untainted as these values originated from hard-coded values in the RT application. A taint value depends on the source of the data itself, so in this case, while user input may be used to make a selection, the data values came from hard-coded values in the application so are considered untainted. Similarly, another column contains timestamps created during internal logging of actions in the service. Because these timestamps were generated by the web application and not specified by user input, they also appeared untainted in the DBTaint database tables.

Columns with only tainted data. There were also columns composed entirely of tainted elements, such as the `subject` column of the `tickets` table. Columns with this property corresponded to mandatory form fields that the user completes while using the application. Because this web application uses output filtering rather than input filtering, it passed user data directly from the web application to the database without sanitization. Each element in the column contains untrusted data from user input, and we can immediately tell that the application is not performing input filtering on these values before storing them.

Columns with mixed tainted and untainted data. While most table columns contained uniformly tainted or untainted data, there were several columns containing both tainted and untainted data. Upon further investigation, we observe that most of these are the columns for optional form fields. The web application provides a default (untainted) value, but if the user provides a value of their own, it will show up as tainted in the database. For example, the `finalpriority` column of the `tickets` table has a default value of 0, which is untainted in the database if the user does not specify any value. However, if the user does provide a value it will show up as tainted in the database.

We manually investigated whether the application might not have sanitized any of these user-supplied values. We discovered that the application always sends data from these columns to the web framework, which sanitizes the data when it is used to generate output. Even though we did not find any sanitization bugs in RT, DBTaint helped us gain confidence in the completeness of sanitization in RT.

3.4.4 Enhancing Functionality

While DBTaint can be used to gain insight into the way that data flows throughout the web application, it can also be used to enhance the functionality of the application without incurring additional security risk. RT escapes angle braces and other potentially dangerous characters from database values before using them to create a HTML page. While this can certainly help prevent cross-site scripting attacks, it also prevents the application from using these dangerous characters in its default values. For example, when a database column contains mixed tainted (user input) and untainted (application default) data, without DBTaint the application must sanitize all of them, unnecessarily restricting default values even though they are safe.

With the cross-application information flow provided by DBTaint, we were able to expand the functionality of the web application without losing security. Since we can reliably track the flow of tainted data through the web application and the database, we can avoid concerns of false positives and false negatives that come with single-application taint tracking schemes. Instead of escaping all data values before returning them to a web visitor, we modified the application to only escape tainted values. Since user data remains tainted through the entire web service, dangerous characters will be escaped in malicious input. On the other hand, trusted values (such as application defaults) will be untainted and can be safely included in HTML pages without undergoing this same escaping procedure.

3.4.5 Performance

We tested the impact of DBTaint on the performance of the RT web application by timing the round trip time of making a request to the web application, processing the request, and receiving the response. We performed 10 sets of 1,500 requests for the original (unmodified) version of RT, and of RT running with our DBTaint implementation. To simulate and envi-

Web Application	Overhead
Request Tracker (RT)	12.77%
JForum	8.49%

Table 3.2. Overall overhead for web services

ronment of a web application under load rather than just starting up, we recorded the time for the last 1,250 requests of each set. Recall that we have made no special effort to optimize our Perl implementation, so each SQL query is reprocessed every time the web application makes a database request. As Table 3.2 shows, we note that even with no attempt at optimization, we achieve less than 13% overhead in our prototype implementation, which we believe provides a high upper bound of the performance impact of our approach.

This test was performed in an isolated environment where the only network traffic to the host was the traffic associated with the performance test. During the test, the service only handled requests made as part of the performance test. These tests were performed by using scripts to automate the requests, and the same test scripts were used for testing the original service and the service running under DBTaint.

3.4.6 Web Application: JForum

To evaluate the effectiveness of our Java implementation of DBTaint, we selected JForum, a "powerful and robust discussion board system" [7], version 2.1.8. JForum includes more than 30,000 lines of code in 350 Java classes, and uses 59 database tables to maintain subforums, posts, messages, access control, and more. We deployed JForum to a Tomcat server with the character-based taint tracking engine described in Section 3.3.3.1.

JForum provides functionality similar to that found in most forum software. Visitors to a JForum site see lists of discussions categorized by subforum. Clicking one loads a page for the selected discussion, which consists of one or more posts by users of the forum. To create a new discussion, JForum provides a page where the user enters their desired subject and body of the first post. Users see a similar page for composing their posts when creating a reply to an existing discussion. Users can submit content in the form of BBCode, a lightweight

markup language, by specifying it manually or using the provided WYSIWYG interface in their browser. This content is stored in the database as text, but JForum translates the BBCode markup to HTML to render the contents to visitors of the site when a page is requested. This allows users to create posts with a variety of formatting options, hyperlinks, and embedded media in a markup language simpler than HTML. Rendering a single discussion page requires content stored across many database tables, such as the `users` table for each poster's information, the `posts_text` table including post subject and content in BBCode, and the `posts` table that associates all supporting metadata with the post by unique identifier.

We evaluated our Java implementation of DBTaint in a similar way to our Perl evaluation in Section 3.4.2. We recorded a series of web events including logging in, posting to the forum, and viewing existing posts. We determined the performance overhead of our Java implementation on JForum to be less than 9% (Table 3.2).

Because our Java implementation uses a character-based taint tracking engine, the query rewriting phase is more sophisticated and complex than our Perl implementation. This is because it handles both parameterized and non-parameterized queries, and checks the taint values of each character in data strings. With this approach, we originally observed an overhead of close to 30%. However, in our Java implementation, we added very simple memoization to the parsing and rewriting of parameterized queries, which dropped the performance overhead to less than that of the Perl implementation, despite the increased complexity. In situations where web services serve far more requests than there are distinct parameterized queries (which we believe is the common case), caching the results from the query rewriting phase is a simple way to improve performance in implementations with sophisticated character-level query rewriting analysis.

3.5 Discussion

We have shown that DBTaint is an effective system for providing cross-application information flow tracking through databases. In this section we outline some of the benefits of DBTaint over other systems, reflections on our prototype implementations, and applications of DBTaint to interesting security problems.

3.5.1 Benefits

DBTaint has the following major benefits:

- End-to-end taint tracking throughout all applications in a web service.
- Full support of the semantics of database operations. DBTaint tracks taint flow at the high database operational semantics level, rather than at the low instruction level.
- Efficiency. DBTaint only tracks the information flow within the database and between the database and its client applications, avoiding the overhead of the extra tracking that system-wide solutions perform. Our unoptimized implementations add only a minor performance penalty to web services.
- Only SQL-level changes to the database server, and no changes to the web application. Our major implementation work is in modifying the database interface. We don't need to make any changes to the database client because DBTaint intercepts and automatically rewrites all SQL queries from the client as needed for our information flow tracking.

3.5.2 Additional Applications of DBTaint

Persistence of taint information. DBTaint allows the taint information stored in the database to remain persistent through multiple runs of the applications. This allows an application that uses the database to run many times without losing the taint information from the previous executions.

Comparison of different versions of a service. DBTaint can be used to compare two different versions of a service that use a database for storage. After refactoring some user input sanitization code, for example, programmers can run the old and new versions of a web application under DBTaint and compare the resulting database tables. Variations in the taint patterns of the database columns may indicate a change in input sanitization policies.

Identification of incomplete input sanitization. Incomplete input sanitization contribute to many security vulnerabilities. Common solutions for detecting incomplete input sanitization are static analysis and runtime testing. Static analysis techniques are often expensive and are

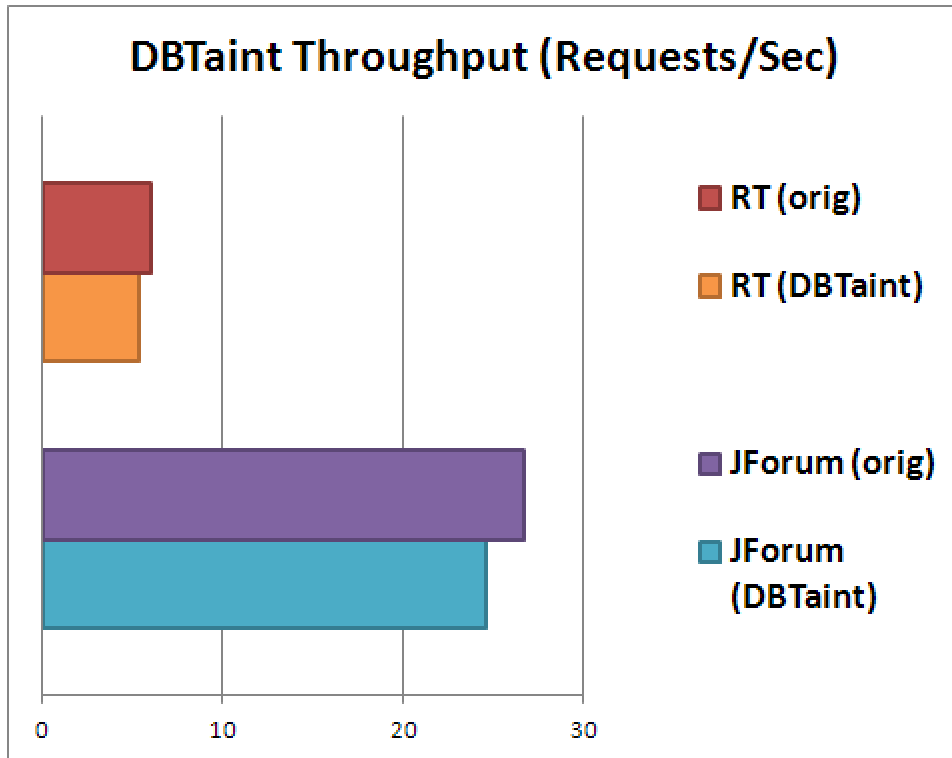

Figure 3.1. Serial throughput of web services running with and without DBTaint

prone to false positives and negatives. For runtime testing, the testers must understand the sanitization functions of the program to design malicious input to test the completeness of the sanitization functions. DBTaint provides an alternative mechanism to detect incomplete input sanitization at runtime and requires no understanding of the sanitization functions in the program.

Web applications typically sanitize untrusted input at two moments: (1) input filtering, which sanitizes an input as it enters the program; and (2) output filtering, which sanitizes untrusted data just before the program embeds the data into a generated web page.

DBTaint has the ability to provide immediate insight into the taint properties of the data in an application without requiring the user to understand the application code. For web services that perform no input filtering,

- Columns that contain only tainted data are likely for storing mandatory user data.

- Columns that contain only untainted data are likely for storing data hardcoded from or

generated by the applications themselves, rather than user input.

- Columns that contain mixed taint and untainted values are used for multiple purposes (e.g., data coming from different applications or code paths), or for optional data fields whose default value (set by the program) is untainted but user value is tainted.

In services that perform input filtering, this column analysis can be even more useful.

- Columns where data are completely untainted indicate that all of the values are either sanitized user input or values produced by the application.
- Columns where data are completely tainted suggest user data that has not been properly sanitized, which may indicate a security vulnerability.
- Columns containing both tainted and untainted data may indicate that the input sanitization is incomplete, i.e., the program sanitizes input data on some paths but not on the other paths.

In the last two cases, DBTaint helps the auditor to reduce the search space for potential sanitization bugs.

3.5.3 Inadvertent Untainting

When using a taint engine, one must be careful to never inadvertently untaint tainted data. DBTaint does not untaint any tainted values (manually or automatically), but unfortunately we found that both single-application taint tracking engines we used for our Perl and Java implementations did perform some inadvertent untainting. This is not the fault of DBTaint, but the problems in these other engines did make our evaluation more difficult.

Perl's taint mode is designed to automatically remove the taint bit on data when it is matched against a regular expression. Perl assumes that a programmer using a regular expression on a variable is validating the contents of the variable, so Perl automatically untaints the value. However, some Perl application and library code uses regular expression for simple string processing, rather than validation or sanitization, leading to inappropriate untainting. We discovered that an encoding/decoding UTF-8 conversion function was untainting all user

input in the RT application before the data reached the DBTaint database interface. We addressed this problem by manually retainting the results of the function when the original string was tainted before the decoding.

We encountered similar difficulties using our character-based taint tracking engine for Java. The Java engine we used provides efficient taint tracking by extending `String` and other `String`-like classes to maintain taint data. However, because the primitive data types are not similarly extended, the engine cannot track taint bits for a `String` converted to a character array and back, for example. All taint bits are lost, inadvertently and incorrectly untainting the resulting String. Due to this limitation, some of the tainted values from the JForum web application received via `POST` submissions became untainted before they reached the DBTaint database interface. As some user input values were inadvertently untainted, we were unable to perform a meaningful analysis of the taint values of each database column.

3.6 Comparisons to Alternate Approaches

The ability to access a web service from anywhere means that it must be able to handle input from any source. Unchecked malicious input can lead to some of the top reported software vulnerabilities in web applications [9]. The information flow tracking provided by DBTaint is like a coarse-grained version of where-provenance [17], allowing developers to identify unchecked user input though multiple applications in the web service without requiring a whole-system solution.

Application-wide information flow tracking. Splint [33] supports source code annotations that help a programmer identify the flow of tainted information within a program. TaintCheck [59] identifies vulnerabilities automatically by performing this analysis on a binary running within its own emulation environment. Xu et al. [70] leverage the source code of a program produce a version of that program that can efficiently track information flow and identify attacks. WebSSARI [47] targets web applications written in PHP specifically with static analysis to identify the information flow of unvalidated input and adds runtime sanitization routines to potentially vulnerable code using that input. Lam et al. [51] also targets web vulnerabilities with the automatic generation of a static and dynamic analysis of a program from a description of an

information flow pattern. Because most modern web services include multiple applications, single-application information flow tracking systems result in false positives and/or negatives because they must assume database values are either tainted or untainted without complete runtime information. DBTaint avoids any need for manual annotation and automatically provides information flow propagation across web service applications.

System-wide information flow tracking. With architectural support, information flow tracking systems can trace untrusted I/O throughout the system [22, 65] at a fine memory address level granularity. These systems however, require substantial changes to the underlying hardware or must emulate the entire system with a performance penalty. Ho et al. [45], provide the same system-wide tracking with the Xen virtual machine monitor and switch to a hardware emulator only when needed to mitigate the performance penalty. However, these approaches pay an unnecessary performance cost by tracking much more than necessary for most web services. Other system-wide information flow tracking systems like HiStar [71] and Asbestos [67] are too coarsely grained to track taint values of individual values throughout the web application and the database. These system-wide approaches also fail to take advantage of the semantics of information flow during database operations. WASC [57] targets web applications and provides a dynamic checking compiler to identify flows and automatically instrument programs with checks. They add support for inter-process flow tracking through databases by maintaining external logs of all SQL transactions, operands and associated tags. This approach lacks DBTaint's ability to use taint values during internal database operations (e.g., preferring untainted values in equality operations for `SELECT DISTINCT` queries).

Chapter 4

Privacy Preserving Alibi Systems

The popularity of modern mobile devices means that more users have access to location-based computing capabilities than ever before. An obvious application of this widespread technology is to track the location of a user automatically. Most existing systems are effectively a repository of the user's claim of their location, as uploaded to a third-party service. In this work, we propose a user-centric system that enables stronger evidence of a user's past location, while leaving the user in control of their privacy. Our system empowers users to observe and control the disclosure of their identity without requiring any new or existing trusted third parties to fulfill our privacy guarantees.

Inspired by the concept of an alibi in the legal setting, we generalize this notation and use "alibi" to refer to evidence of a person's past location at a point in time. It can occasionally be extremely important to have such an alibi, and we have a designed a system to enable users to opportunistically create alibis without giving up their privacy. An alibi must be bound to a person's identity to prevent from being transferred to another person; however, requiring a person to reveal her identity during alibi creation would compromise the person's privacy. We propose a privacy-preserving alibi system, where a user conceals her identity during alibi creation. The user's identity is revealed only when she chooses to present her alibi to a judge.

We design two privacy-preserving alibi schemes. In the first scheme, the alibi corroborator is a public entity and therefore needs no privacy protection. Our second scheme protects the privacy of the corroborator as well, where the identity of the corroborator is revealed only when he chooses to help the alibi owner to present her alibi to the judge. We discuss the

properties of our schemes and demonstrate their advantages over current alibis. As ubiquitous mobile computing presents an attractive platform for deploying our schemes, we have implemented our schemes on an Android device and shown its satisfactory performance.

4.1 Introduction

Black's Law Dictionary defines an alibi as *a defense based on the physical impossibility of a defendant's guilt by placing the defendant in a location other than the scene of the crime at the relevant time.* [36] The ability to provide evidence of one's past locations can be extremely important. For example, in 2008 murder charges were dropped against a Bronx man and his brother after they used their New York City Transit MetroCard to support their claim that they were miles from the scene of the crime at the time it occurred. [68]

Some alibis are based on witness testimony. Such an alibi relies on the memory of the corroborator; however, the corroborator may forget about the encounter or may misremember key details, such as the identity of the other party, or the date and location of the encounter.

Other alibis are based on physical evidence. As mobile devices become ubiquitous and accompany us on our daily activities, they have the ability to determine where we are and what we are doing. Location-based services like Google Latitude can track our every move, so they could provide physical evidence as our alibis.

If a physical evidence is not bound to a person, it can be used by other people to claim their fake alibi. On the other hand, if a person has to reveal her identity when creating a physical evidence, then her privacy is at risk. Privacy advocates are becoming increasingly concerned [15] that third-party services have so much access to information about our lives. These services generally require the user to decide whether they want to be tracked at the time the tracking occurs. Imagine a user who temporarily disables their location-tracking service to prevent their employer from learning of a long lunch break. While this may seem a reasonable decision at the time, the user has no way to "go back" and show their location later if they need to prove their innocence when they are incorrectly accused of a crime.

When a person claims an alibi, she must reveal her identity because the judge must verify that the identity in the alibi matches the identity of that person. However, our key insight is

that we could design an alibi system where a person does not reveal her identity when creating her alibi. In this system, a person remains anonymous until she chooses to use her alibi in front of a judge. This system allows a user to create alibis whenever she can without compromising her privacy.

An alibi involves two parties: the *owner*, who benefits from the alibi, and the *corroborator*, who testifies for the owner. Our goal is to allow an owner to create alibis with corroborators without revealing her identity to the corroborators. To prevent the transfer of an alibi from one owner to another, the alibi must be bound to the owner's identity, although this binding is hidden at alibi creation time. To prevent the owner from lying about the context, such as time and location, the alibi must also include the context certified by the corroborator.

The advent of ubiquitous mobile computing provides an attractive platform for implementing this privacy-preserving alibi scheme. The user's mobile device can act as the user's delegate in alibi creation. The corroborator can be a public entity, such as a subway station, or a private entity, such as another mobile device. When public entities corroborate alibis they need not protect their privacy, but private entities may wish to protect their own privacy. Therefore, we have designed two privacy-preserving alibi schemes, one with public corroborators (Section 4.2) and one with private corroborators (Section 4.5).

4.1.1 Contributions

- We propose a privacy-preserving alibi system where the identity of the alibi owner is concealed at the time of alibi creation. The owner reveals her identity only when she chooses to present her alibi to a judge.

- We design two privacy-preserving alibi schemes. The public corroborator scheme (Section 4.2) always reveals the identity of the corroborator. By contrast, the private corroborator scheme (Section 4.5) conceals the identity of the corroborator, and the corroborator reveals his identity only when he agrees to help the alibi owner to present her alibi to the judge.

- We discuss the properties of our schemes and demonstrate their advantages over current alibis. (Section 4.7)

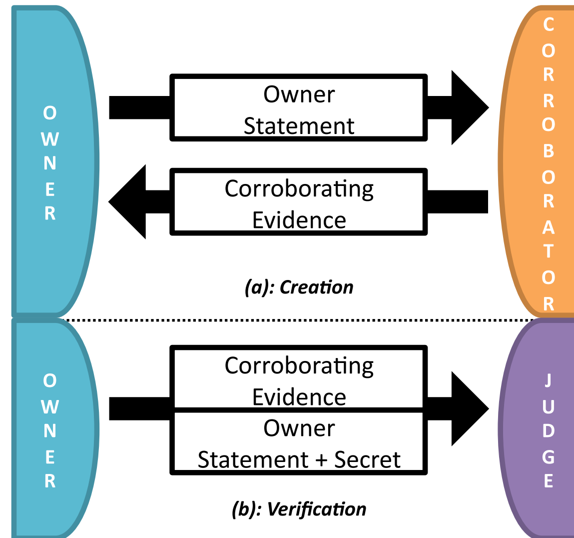

Figure 4.1. The public corroborator scheme

- We have implemented the scheme on a mobile device and evaluated its performance. (Section 4.8)

4.2 Public Corroborator Scheme

In our "public corroborator" scheme we assume that all corroborators are publicly known, and the corroborators' identities and locations are not considered private. These schemes are appropriate for settings where corroborators are public entities, such as subway stations or other infrastructure without privacy concerns. We discuss an alternative scheme in Section 4.5 that allows the corroborator to control the disclosure of his identity, which is more appropriate for corroborators who also have privacy concerns, such as other mobile device users.

In contrast to alternative schemes (such as VeriPlace [52]), we do not require that corroborators have fixed locations, nor do we require a central database mapping all corroborators to their locations.

4.2.1 Overview

Figure 4.1 illustrates the two phases in our public corroborator alibi scheme.

4.2.1.1 Alibi Creation

The owner can opportunistically participate in "alibi creation" whenever corroborators are available. The owner creates an OwnerStatement for each alibi she creates. The content of

the $\mathrm{OwnerStatement}$ is tied to the Owner who created it, but by itself can't reveal the identity of the Owner unless the Owner reveals that link.

The corroborator sends back $\mathrm{CorroboratingEvidence}$, in which the corroborator conveys, "I have received the $\mathrm{OwnerStatement}$ a in the current context c."

Participating in the creation phase does not reveal the identity of the owner, but does give the owner the opportunity to claim an alibi for their current context at a later time.

4.2.1.2 Alibi Verification

When the owner wishes to claim an alibi they have created, they participate in the "alibi verification" stage with a judge. In this stage, the owner reveals her identity associated with only the $\mathrm{OwnerStatement}$ used to create the single alibi she is claiming. This doesn't reveal her identity in any other unclaimed alibis, or allow anyone else to create more alibis on her behalf.

The owner must demonstrate two things to the judge. First, the owner must show that the corroborator certifies that they received a specific $\mathrm{OwnerStatement}$ in the context where the owner claims to have an alibi. Second, the owner must demonstrate the link between the $\mathrm{OwnerStatement}$ and the Owner. The judge checks to make sure that the $\mathrm{OwnerStatement}$ corresponds to the Owner (that is, her identity), and that it is the same $\mathrm{OwnerStatement}$ that the corroborator claims to have received at the context in question.

We note that just as in the "traditional" alibi setting, the "strength" of an alibi (e.g., as considered by a jury) depends heavily on the perceived trustworthiness of the corroborator. We compare our alibis to traditional alibis in Section 4.7.

4.2.2 Design

4.2.2.1 Initialization

Each alibi owner and alibi corroborator has their own public/private key pair (pk_o, sk_o) and (pk_c, sk_c), respectively. Both parties must have the necessary sensors to determine the current context (location, date, time), and represent this information in the same way. Also, we assume that the alibi owner has access to a collision-free hash function $\mathrm{MD}(\cdot)$ (for Message Digest).

The judge has access to both the alibi owner's and alibi corroborator's public keys (with the ability to verify signatures made by these parties). The judge can also compute the message

digest function $\mathrm{MD}(\cdot)$ used by the alibi owner.

4.2.2.2 Alibi Creation

We assume that the owner has a fixed, unique identity called $\mathrm{OwnerID}$, and agrees with the corroborator on a value describing the current context.

The alibi owner creates the tuple i which includes the owner's identity and the current context as determined by the owner.

$$i = (\mathrm{OwnerID}, \mathrm{Context}_o)$$

The alibi owner signs i, and a tuple containing i and the alibi owner's signature over i becomes what we call the $\mathrm{OwnerFeatures}$.

$$\mathrm{OwnerFeatures} = (i, \mathrm{Sign}_{sk_o}(i))$$

Sending the $\mathrm{OwnerFeatures}$ directly to the corroborator would reveal the owner's identity. We prevent this by using highly efficient cryptographic primitives to create a representation of the $\mathrm{OwnerFeatures}$ that doesn't reveal the $\mathrm{OwnerFeatures}$ values without extra information from the Owner.

We use a cryptographic commitment scheme to create the $\mathrm{OwnerStatement}$ from the $\mathrm{OwnerFeatures}$. Specifically, we use the scheme presented by Halevi and Micali [42], which is a non-interactive string commitment scheme based on collision-free hashing. We note that we are not tied to this particular scheme. Other string commitment schemes (such as [23]) could be used in place without changing the security our of scheme (except for differences in the hardness assumptions underlying the commitment schemes).

To form a commitment to the $\mathrm{OwnerFeatures}$, the alibi owner uses the message digest function $\mathrm{MD}(\cdot)$ to compute s_o where

$$s_o = \mathrm{MD}(\mathrm{OwnerFeatures})$$

The alibi owner chooses a random value x_o which we call the *verification secret*, which she keeps secret until she wishes to claim her alibi. Next, the alibi owner randomly selects a universal hash function $h_o(\cdot)$ where

$$h_o(x_o) = s_o$$

The alibi owner may do this by choosing a random x_o and selecting $h()$ of the form $h(r) = Ar + b$ by choosing A randomly and computing $b = s_o - A(x_o)$, where $h()$ is in linear space over $\mathrm{GF}(2)$.

Finally, the alibi owner computes

$$y_o = \mathrm{MD}(x_o)$$

At this point the owner has constructed the $\mathrm{OwnerStatement}$, which is the commitment to the $\mathrm{OwnerFeatures}$:

$$\mathrm{OwnerStatement} = (h_o, y_o)$$

The owner sends the completed $\mathrm{OwnerStatement}$ to the alibi corroborator. The corroborator must certify that the $\mathrm{OwnerStatement}$ was received in the current context. The alibi corroborator combines the $\mathrm{OwnerStatement}$ tuple with the current context (as determined by the corroborator) to create the tuple

$$a = (h_o, y_o, \mathrm{Context}_c)$$

and creates the $\mathrm{CorroboratingEvidence}$

$$\mathrm{CorroboratingEvidence} = (a, \mathrm{Sign}_{sk_c}(a))$$

The alibi corroborator sends the $\mathrm{CorroboratingEvidence}$ back to the alibi owner. The alibi owner confirms that the values in a are correct (including the context provided by the corroborator), and that the signature from the alibi corroborator is valid. At this point, the alibi owner has the $\mathrm{CorroboratingEvidence}$ needed to claim their alibi later.

We note that the alibi owner must store the context, h_o, x_o, and the corroborator's signature. This information allows the owner to recompute the rest of the values needed to claim the alibi. The owner can claim the alibi they just created without requiring the corroborator to maintain any information about this exchange.

4.2.2.3 Alibi Verification

An alibi owner claims her alibi by presenting her evidence to the judge, demonstrating two things. First, the owner must present the $\mathrm{CorroboratingEvidence}$, which shows the corroborator's certification of having been presented with the given $\mathrm{OwnerStatement}$ in a specific

context. Second, the owner must demonstrate that the $\mathrm{OwnerStatement}$ the corroborator received is valid and linked to the owner's identity.

The owner sends h_o, y_o, the context value provided by the corroborator and the corroborator's signature over these values, as well as the owner's verification secret x_o. Note that the if the owner did not store y_o then she may recompute it with $y_o = \mathrm{MD}(x_o)$. The judge confirms the validity of the signature.

Then, the owner decommits their commitment in the $\mathrm{OwnerStatement}$ by providing x_o and the $\mathrm{OwnerFeatures}$ values. The judge inspects the contents of $\mathrm{OwnerFeatures}$ to make sure the $\mathrm{OwnerID}$ belongs to the owner, that the context in the $\mathrm{OwnerFeatures}$ matches the context signed by the corroborator, and that the owner's signature is valid. The judge then computes $s_o = \mathrm{MD}(\mathrm{OwnerFeatures})$, and checks that $h_o(x_o) = s_o$ and $\mathrm{MD}(x_o) = y_o$.

Sybil-alibi attacks. Normally alibis work only in the favor of the owner, because the owner chooses to verify an alibi only when she would benefit. However, in certain circumstances, the owner might be coerced to verify an alibi in her disfavor. In this case, we need to prevent the Sybil-alibi attack, where malicious corroborators create new alibis for the owner based on her verified alibi. In this attack, a malicious corroborator takes the $\mathrm{OwnerStatement}$ from the verified $\mathrm{CorroboratingEvidence}$ to create a new $\mathrm{CorroboratingEvidence}$. The same x_o that the owner used to verify her original $\mathrm{CorroboratingEvidence}$ could be used to verify this forged $\mathrm{CorroboratingEvidence}$.

Our scheme prevents this Sybil-alibi attack by verifying that the $\mathrm{OwnerStatement}$ in each $\mathrm{CorroboratingEvidence}$ is unique when multiple $\mathrm{CorroboratingEvidence}$ values are provided for the same owner.

4.3 Threat Model

To distinguish them from the alibis in our systems, we call traditional alibis (as thought of in the current legal system) "physical alibis." A physical alibi has three components: the identity of the owner, the identity of the corroborator, and the context (date, time, and location information).

4.3.1 Identity

We require a public key infrastructure that binds keys to legal identities. This allows us to represent witness statements in the physical world as signed messages in our scheme. Since a private key represents a legal identity, we assume that no one except the owner has her private key. Determining whether the identity associated with a private key matches the identity of the human using the key is outside of the scope of this work.

4.3.2 Context

In a physical alibi, the corroborator believes that both he and the alibi owner were in the same context based on certain facts, which determines the reliability of the alibi. For example, if the corroborator saw the alibi owner, the alibi is highly reliable; however, if the corroborator overheard the alibi owner's voice in another room but never saw her, the alibi is less reliable, because the corroborator could have heard a recording of her voice. We distinguish between the reliability of the alibi evidence and the trustworthiness of the corroborator, and we will discuss the latter next.

In our scheme, we require that the corroborator can correctly measure his context, which should also includes the means by which he interacted with the alibi owner. The judge takes the means of the interaction into consideration when determining the reliability of the alibi evidence. For example, an alibi created through an interaction via near field communication (NFC) might be considered stronger than one created over WiFi, as the corroborator is likely to be more certain of his proximity to the alibi owner. We can improve the reliability of alibis evidence by secure location verification techniques [64], which are orthogonal to this work.

Our scheme also requires the owner to include his view of the context in the alibi (in $\mathrm{OwnerFeatures}$). The purpose is to prevent the attack where a malicious corroborator creates a new $\mathrm{CorroboratingEvidence}$ from an old $\mathrm{OwnerStatement}$ without the alibi owner's participation. If the corroborator uses a different context in the $\mathrm{CorroboratingEvidence}$ than the one in $\mathrm{OwnerFeatures}$, the judge will detect the discrepancy when verifying the $\mathrm{CorroboratingEvidence}$. On the other hand, if the attacker must use the same context value, then the attacker can only create additional $\mathrm{CorroboratingEvidence}$ for the original alibi. During the verification stage a judge can detect when two $\mathrm{CorroboratingEvidence}$ values

correspond to a single OwnerStatement, revealing this misbehavior.

4.3.3 Privacy

We assume that an attacker may try to learn the identity of any party in our system only via messages in our protocols. Therefore, we do not consider privacy attacks using out of band channels. For example, the cellular network provider of the alibi owner may learn her identity; the corroborator may use recording devices, such as cameras, to determine the identity of the alibi owner. These are out of the scope of this work.

4.3.4 Trust

We require no trusted third party. Moreover, we require no trust between alibi owners and corroborators. A malicious corroborator can refuse to provide the valid CorroboratingEvidence the owner needs for the alibi. This is a form of denial of service attack. We do not attempt to prevent this attack, because a solution would force witnesses to provide valid alibis, which we do not believe to be desirable. On the other hand, our scheme prevents a malicious corroborator from discovering the identity of the alibi owner (Section 4.4.1.3).

If the corroborator collaborates with the alibi owner, they can create false but valid alibis. This, under the right circumstances, may be perjury. Just as we cannot prevent the creation of perjury (even though we may expose it by other means) in real life, our scheme does not try to prevent perjury.

If the corroborator unilaterally intends to create false alibi to benefit the owner without the collaboration from the owner, this is another form of perjury. In the physical world, we cannot prevent the creation of such perjury (even though we may expose them through other means). By contrast, our scheme can detect such attacks (Section 4.4.1.1).

As with physical alibis, the value of alibis produced by our scheme depends on the trustworthiness of the corroborators. We leave the problem of determining the trustworthiness of the corroborator to the judge.

4.4 Properties of the Public Corroborator Scheme

Now that we've defined an alibi scheme, we describe all of the security properties we desire in our setting and how our implementation satisfies them.

4.4.1 Security Properties

4.4.1.1 Non-forgeability

A CorroboratingEvidence binds the owner's identity, the corroborator's identity, and the context. An alibi is valid if it can be successfully verified (Section 4.2.2.3). We claim that no valid alibi can be created without the collaboration of both the owner and the corroborator.

First, we consider how someone, including the corroborator, could forge an alibi without the owner's cooperation.

- The forger could try to forge a fresh alibi for an alibi owner, but this would fail because he does not have the owner's private key needed to create a valid OwnerStatement.
- The forger could attempt to create another CorroboratingEvidence for an existing unclaimed alibi. In this case, no one can verify the new CorroboratingEvidence without the verification secret chosen by the alibi owner during the creation of the original OwnerStatement.
- The forger could attempt to generate a fake CorroboratingEvidence for an existing OwnerStatement with a different context from the one in which the owner created the OwnerStatement. However, this forgery would be detected in the verification stage because the OwnerFeatures linked to the OwnerStatement includes the owner's context value, which will not match the context in the forged CorroboratingEvidence.
- The forger could make a fake CorroboratingEvidence for an existing OwnerStatement using the same context in which the owner created the OwnerStatement. However, at worst this attack can only result in adding a false corroboration to an existing, valid alibi. As the owner already has a valid alibi this forgery can only give the owner an additional (malicious) corroborator of the existing alibi, and cannot result in an alibi that places the owner in a different context. If the judge inspects both alibis then this misbehavior is easily detected, as both CorroboratingEvidence values will correspond to the same OwnerStatement, which would not occur under normal circumstances.

Next, we consider how someone, including the alibi owner, could forge an alibi without the corroborator's collaboration. This is infeasible because the forger doesn't have the private

key of the intended corroborator so therefore cannot create the signature in the corroborating evidence.

4.4.1.2 Non-transferability

A corroborating evidence is non-transferable because it has the signature of both the owner and the corroborator.

4.4.1.3 Privacy

Our scheme preserves the privacy of the owner in the following properties:

- No one, including a malicious corroborator, can uncover the identity of the alibi owner at any stage before the owner verifies her $\mathrm{CorroboratingEvidence}$ in the protocol.
- No one, including any number of collaborating malicious corroborators, can link multiple unclaimed alibis created by the same alibi owner (i.e., unclaimed $\mathrm{OwnerStatement}$ and corresponding $\mathrm{CorroboratingEvidence}$ values).
- When an owner claims her alibi by entering into the verification stage, she reveals her identity. However, no one, including any number of collaborating malicious corroborators, can link her to any of her unclaimed alibis.

The above properties are guaranteed by the string commitment scheme in our protocol.

4.4.2 Other Properties

4.4.2.1 Storage

Our scheme requires the owner to store all the data necessary for verifying an alibi. In contrast, no corroborator needs to store any data about the alibis that he has helped create (except their private keys, which our threat model assumes). The advantage of this design is that it aligns with the incentive of the owner to protect and preserve her alibis.

4.4.2.2 Efficiency

Our scheme is efficient both in time and space. In Section 4.8 we evaluate the performance of our scheme on an Android device.

4.5 Private Corroborator Scheme

4.5.1 Motivation

In our public corroborator scheme, the corroborator's identity is always revealed during the creation phase, but the owner's identity isn't revealed until the owner wishes to claim that alibi. In settings where corroborators are public entities (e.g., subway stations), it is acceptable for someone to learn the identities of every corroborator with whom she creates an alibi. However, in other settings a corroborator may not want to reveal his identity every time an alibi owner wants to create an alibi with him. Particularly, if the corroborator allows his mobile device to create abilis for anyone within proximity, the previous scheme would allow an attacker to identity and track the corroborator. We wish to design a private corroborator scheme where the corroborator's identity is not revealed at alibi creation, analogous to the property that the owner's identity is not revealed at alibi creation in the public corroborator scheme.

4.5.1.1 Rejected Designs

One might simply apply our public corroborator scheme but allow the corroborator to decide to whether to create $\mathrm{CorroboratingEvidence}$ for the alibi owner during each encounter. However, this would require the corroborator to decide at alibi creation time whether he wants to reveal his identity, while the owner can wait until she verifies her alibi to reveal her identity. This deficiency would create a privacy and usability headache for the corroborator: for each alibi creation request, the corroborator would have to decide whether to help create the alibi either manually or using some policies, which could be complex and error prone.

One might require the corroborator to store each $\mathrm{OwnerStatement}$ (sent by the alibi owner) along with the associated context, and only return the $\mathrm{CorroboratingEvidence}$ when the owner requests to verify the alibi rather than during alibi creation. However, this would require the corroborator to bear the burden of storing the alibi, when the owner has much higher incentive to store her alibis safely. In this setting, an honest and willing corroborator may be unable to corroborate an important alibi because he deleted the alibi when he ran out of disk space. We would like a scheme that is completely stateless for the corroborator. We want to allow the owner to retain all of the information necessary for her and the corroborator to corroborate her alibi.

One might imagine a scheme where the corroborator sends his created alibi to a trusted third party instead of the alibi owner. However, this violates the requirement for no trusted third party in our threat model.

One might suggest that we use zero knowledge schemes to allow the provider to prove that there exists some corroborator without revealing the identity of the corroborator. However, we believe that alibis are of little value if the corroborator's identity is not revealed, because the value depends, in part, on the trustworthiness of the corroborator.

4.5.2 Overview

Under the above considerations, we have designed a private corroborator alibi scheme where the alibi owner, when wishing to verify her alibi, simply contacts the alibi corroborator to "ask" if he is willing to corroborate her alibi. The owner can use an anonymous messaging system such as SMILE [53] to contact the corroborator. Our scheme gives the corroborator as much control over his privacy as the owner over hers. Just as the alibi owner can freely ask corroborators to create alibis without revealing her identity, the corroborator can freely help owners to create their alibis without revealing his identity. The alibi owner reveals her identity only when she wishes to verify her alibi, and the corroborator reveals his identity only when he helps the owner to verify her alibi.

We show the three phases of our private corroborator scheme in Figure 4.2. Note that we have added a "corroboration" phase, in which the owner asks the corroborator to identify himself and corroborate the alibi. This means that the owner must be able to contact the corroborator without knowing his identity. In our private corroborator scheme we assume that the participants also have access to some anonymous messaging system that provides this functionality, such as SMILE [53]. SMILE requires a trusted third party, but only for message delivery. While misbehavior of a third party in SMILE might prevent an alibi owner from communicating with a corroborator (denial-of-service), this third party can not compromise the privacy of the alibis in our scheme.

If the corroborator agrees to corroborate the alibi, he returns the $\mathrm{CorroboratingEvidence}$ the owner needs to claim her alibi. To claim her alibi, the owner presents her $\mathrm{OwnerStatement}$ to the judge, along with the link to her $\mathrm{OwnerFeatures}$ and the $\mathrm{CorroboratingEvidence}$.

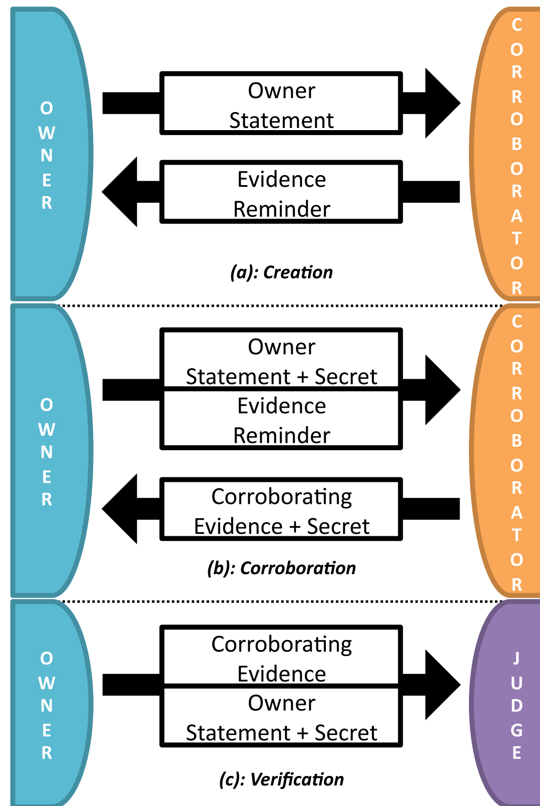

Figure 4.2. The private corroborator scheme

4.5.3 Initialization

Just as in our public corroborator scheme, each alibi owner and alibi corroborator has their own public/private key pair (pk_o, sk_o) and (pk_c, sk_c), respectively. In our private corroborator scheme, each corroborator also has her own pseudorandom function $\mathrm{prf}_c(\cdot)$ with secret key. Both parties have the rest of the capabilities as in the public corroborator scheme.

4.5.4 Alibi Creation

In this phase, the owner begins the exchange just as in the public corroborator scheme. The owner creates her $\mathrm{OwnerStatement}$ in the manner described in Section 4.2, then sends the $\mathrm{OwnerStatement}$ to the corroborator.

Upon receiving the $\mathrm{OwnerStatement}$, the corroborator computes his signature over the $\mathrm{OwnerStatement}$ and the context in which it was received by the corroborator. However, instead of sending this signature back to the alibi owner, the corroborator commits to this data using the cryptographic string commitment scheme.

To commit to the information needed to corroborate the owner's identity the corroborator selects a random value r_c and computes

$$\begin{aligned} j &= (\text{OwnerStatement}, \text{Context}_c) \\ s_c &= \text{MD}(j, \text{Sign}_{sk_c}(j)) \end{aligned}$$

The corroborator commits to s_c by computing

$$\begin{aligned} x_c &= \text{prf}_c(r_c) \\ y_c &= \text{MD}(x_c) \end{aligned}$$

and choosing a universal hash function $h_c(\cdot)$ where

$$h_c(x_c) = s_c$$

Combining these values gives the corroborator the EvidenceReminder, which the corroborator sends to the alibi owner.

$$\text{EvidenceReminder} = (h_c, y_c, r_c, \text{Context}_c)$$

In order to be able to claim their alibi later, the alibi owner only needs to store the context used in the OwnerFeatures, h_o, x_o, and the EvidenceReminder $(h_c, y_c, r_c, \text{Context}_c)$. The alibi corroborator does not need to store any values (except for their own private keys).

Note that the corroborator does not send his verification secret x_c to the owner at this time. This value does not need to be stored because the corroborator can use r_c and her pseudorandom function (keyed with her secret key) to recompute x_c in the corroboration phase.

Also in this phase the provider and corroborator must exchange whatever information necessary to allow the provider to send a message to the corroborator anonymously if the provider wishes to claim their alibi.

4.5.5 Alibi Corroboration

To claim their alibi, the owner contacts the corroborator (via anonymous messaging system) to ask if they are willing to reveal their identity in association with a specified context. If so, then the owner first reveals his identity to the corroborator by sending her verification secret x_o and

$\mathrm{OwnerFeatures}$ to the corroborator. The corroborator checks the owner's decommitment, and proceeds if the decommitment is valid.

The owner sends the $\mathrm{EvidenceReminder}$ the corroborator created in the creation phase back to the corroborator. Because the $\mathrm{EvidenceReminder}$ contains r_c, the corroborator can recompute x_c needed to decommit h_c and y_c. The corroborator takes the $\mathrm{OwnerStatement}$ and context provided by the owner and recomputes

$$
\begin{aligned}
j &= (\mathrm{OwnerStatement}, \mathrm{Context}_c) \\
s_c &= \mathrm{MD}(j, \mathrm{Sign}_{sk_c}(j))
\end{aligned}
$$

The corroborator checks to see if this s_c value is the value they committed to in the creation phase. That is, the corroborator checks to see that the $h_c(x_c) = s_c$ and $\mathrm{MD}(x_c) = y_c$.

If so, the corroborator knows that they must have computed and sent a signature over the $\mathrm{OwnerStatement}$ in $\mathrm{Context}_c$, which they would have only done if they received that $\mathrm{OwnerStatement}$ in an alibi creation exchange in that given context. So, because the corroborator has decided to support the owner's alibi claim for this context, the corroborator returns the $\mathrm{CorroboratingEvidence}$ including the signature in the same format as in the public corroborator scheme.

$$
\begin{aligned}
a &= (h_o, y_o, \mathrm{Context}_c) \\
\mathrm{CorroboratingEvidence} &= (a, \mathrm{Sign}_{sk_c}(a))
\end{aligned}
$$

The corroborator sends the $\mathrm{CorroboratingEvidence}$ to the owner. In our private corroborator scheme, the corroborator also sends the x_c value to the alibi owner along with the $\mathrm{CorroboratingEvidence}$. The alibi owner uses this to verify the corroborator's decommitment to the $\mathrm{EvidenceReminder}$. This allows the corroborator to demonstrate the link between the signature in the $\mathrm{CorroboratingEvidence}$ and the $\mathrm{EvidenceReminder}$ the corroborator gave to the owner in the creation phase. The owner can make sure that the signature received in the corroboration phase is the same as corroborator's signature made during the creation phase.

4.5.6 Alibi Verification

Alibi verification in the private corroborator scheme is done in exactly the same way as in the public corroborator scheme. Just as before, the owner sends h_o, y_o, and r_o to the

judge, along with the context as specified by the corroborator, and the corroborator's signature over these values. The owner also decommits their $\mathrm{OwnerStatement}$, and reveals the $\mathrm{OwnerFeatures}$. The judge checks the corroborator's signature, and the owners decommitment and $\mathrm{OwnerFeatures}$.

4.6 Properties of the Private Corroborator Scheme

The private corroborator scheme shares all the properties of the public corroborator scheme described in Section 4.4 (except that the corroborator learns the identity of the alibi owner at the corroboration stage in the private corroborator scheme, instead of at the verification stage in the public corroborator scheme). In this section we discuss additional properties of the private corroborator scheme.

4.6.1 Privacy

Our scheme preserves the privacy of the alibi corroborator in the following properties:

- No one, including a malicious alibi owner, can uncover the identity of the alibi corroborator before the corroborator creates the $\mathrm{CorroboratingEvidence}$ in the protocol.
- No one, including any number of collaborating malicious alibi owners, can link multiple $\mathrm{EvidenceReminder}$ values created by the same corroborator.
- When a corroborator sends a $\mathrm{CorroboratingEvidence}$ in reply to an $\mathrm{EvidenceReminder}$ sent by an alibi owner, the corroborator reveals his identity. However, no one, including any number of collaborating malicious alibi owners, can link him to any of the evidence reminders that he has created (in the creation stage) but not yet used (in the corroboration stage).

These properties are guaranteed by the string commitment scheme in our protocol.

4.6.2 Reciprocity

The private corroborator scheme raises the question of reciprocity of privacy: is it possible for one party to learn the other party's identity without revealing his own? A fair exchange

scheme (such as described by Micali [55]) might allow us to achieve privacy reciprocity but it requires a trusted third party, which our threat model precludes.

We believe that privacy reciprocity is unnecessary for our scheme. First, before the parties enter the corroboration stage, neither party's identity is revealed. Second, after the parties enter corroboration, the owner reveals her identity before the corroborator does. Therefore, it is possible that the corroborator learns the owner's identity without revealing his identity to the owner, but only when the owner chooses to reveal her identity to get a corroborated alibi from the corroborator. Just as in the physical world, a defendant cannot remain anonymous while asking a witness to testify for her, and has to bear the risk that the witness may decline to come forward after she reveals her identity.

Note that a malicious corroborator cannot force an alibi owner to reveal herself, as the owner must initiate the corroboration stage.

4.7 Comparison to Physical Alibis

We call alibis used in current legal systems *physical alibis*. A physical alibi includes the corroborator, the owner (the beneficiary), and the context, which includes the means by which the corroborator identifies the owner. For example, in the case a personal witness, the corroborator is a person and the means is via physical senses such as vision; in the case of a physical evidence, the corroborator is the entity that issues the physical evidence (e.g., a subway station), and the means is the physical evidence (e.g., a subway ticket).

We discuss some comparisons between our alibis and physical alibis. The unique properties of our alibis give participants several advantages over physical alibis, including:

- They better protect the privacy of the alibi owner (and corroborator in our private-corroborator scheme)
- They have non-forgeability properties beyond that of many physical alibis
- They embed the identities of the participants directly and unambiguously into the alibis
- They help prevent alibis from being forgotten or faded over time

4.7.1 Common Properties

Trustworthiness of Corroborator. The strength of a physical alibi depends on the reliability of the evidence and the trustworthiness of the corroborator. The same applies to our alibis. For example, our scheme cannot prevent a collaborating owner and corroborator from creating a fake but valid alibi (perjury). Just like physical alibis, our alibis leave the determination of the trustworthiness of the alibis to the judges.

Privacy of Corroborator. The corroborator of a physical alibi may wish to protect his privacy by remaining anonymous. In this case, the alibi becomes useless because no one can judge the trustworthiness of the corroborator.

In our public scheme (Section 4.2), the identity of the corroborator is public. However, in our private scheme (Section 4.5), the corroborator may remain anonymous by refusing to corroborate the $\mathrm{CorroboratingEvidence}$ that he created earlier.

4.7.2 Benefits

4.7.2.1 Privacy

Consent to Alibi Creation. In physical alibi settings, alibis may be created for a person without her consent. For example, without a person's content, she may be remembered by a doorman, or her photos may be taken by a camera. By contrast, our scheme requires the owner to initiate alibi creation.

Consent to Alibi Verification. Although alibis often benefit the owners, they may harm the owners as well, such as when they are used as evidence by the prosecutors. Therefore, the owner has to decide in advance whether she wants her physical alibi to be created (if she does not, then she may disguise herself or avoid contact with the corroborator). Since she may not know in advance whether her alibi may be beneficial or harmful, she faces a dilemma: if she chooses to have her alibi created, it may harm her in the future; however, if she chooses not to have her alibi created, she may lose important alibis that could prove her innocent in the future. The cause of her dilemma is that other people can verify her physical alibis without her consent.

By contrast, an alibi in our scheme is unverifiable unless its owner consents (by providing

the verification secret). This removes the dilemma that the owners face when creating physical alibis. In our scheme, the owners can freely create alibis. Later, she can decide to verify only the alibis that are beneficial to her.

4.7.2.2 Reliability

Accuracy. There are a number of problems with physical alibis where one person is the corroborator for another. The corroborator may misremember the identity of the alibi owner (e.g., Charlie thinks that he saw Bob when he actually saw Alice), the context (e.g., Charlie thinks that he saw Alice on Monday when he actually saw Alice on Tuesday), or the link between the alibi owner and context (e.g., Charlie thinks that he saw Alice on Monday and Bob on Tuesday when he actually saw Alice on Tuesday and Bob on Monday). An alibi in our scheme binds the identities of the alibi owner and the corroborator to the context, so it avoids all the above human inaccuracies.

Availability. Physical alibis rely on the memory of the corroborators to recall the encounters. However, the corroborator may forget the encounter partially or completely. By contrast, in our scheme the alibi owner stores all the data necessary to corroborate and verify the alibi.[1] Since the alibi owner benefits from the alibi, she naturally has the incentive to store her alibis safely.

4.7.3 Weaknesses

Our scheme requires a trustworthy public key infrastructure where each private key represents a person. In our threat model, we assume that each private key can only be accessed by the owner of that private key. We note that if Mallory gains access to Alice's private key, then she can create alibis on Alice's behalf. Our scheme is not intended to determine whether the user of a private key is actually the owner of the private key.

4.8 Performance Evaluation

To evaluate the real-world feasibility of our scheme, we have implemented all of the computational steps required to create, corroborate, claim and verify alibis in our public corroborator

[1] In the private scheme (Section 4.5), the corroborator needs to remember his private key in the corroboration stage, but this is guaranteed by the assumption of a public key infrastructure in our threat model (Section 4.3).

Operation	Time (sec)
Owner Statement Creation	0.279
Corroborator Creation (public scheme)	0.070
Corroborator Creation (private scheme)	0.279
Corroboration (private scheme)	0.277
Corroboration Verification (private scheme)	0.205
Alibi Verification	0.216

Table 4.1. Average execution times for alibi operations on a Motorola Droid

and private corroborator schemes. Our implementation runs on the Android mobile platform, and we measured the performance of all of these operations on a Motorola Droid phone.

4.8.1 Benchmarks

In order for a participant to create or corroborate alibis in our systems, they must first create their public/private key pairs and perform other initialization operations. Secure key pair creation is relatively slow on mobile devices (averaging 6.91 seconds). However, once the participants complete this one-time initialization, they can create and corroborate as many alibis as they like. After creating 1,000 alibis we computed the average time required for the Motorola Droid to complete the major operations in our scheme. The results of our benchmarks are shown in Table 4.1.

Owner Statement Creation the time required for the provider to create their $\mathrm{OwnerFeatures}$ and commitment to this value, creating the $\mathrm{OwnerStatement}$

Corroborator Creation (public scheme) the time required for the corroborator to create the corroborating evidence in the creation phase of the public corroborator scheme

Corroborator Creation (private scheme) the time required for the corroborator to create the $\mathrm{EvidenceReminder}$ in the creation phase of the private corroborator scheme

Corroboration (private scheme) the time required for the corroborator to first verify the $\mathrm{EvidenceReminder}$, then use it to create $\mathrm{CorroboratingEvidence}$ in the private cor-

roborator scheme

Corroboration Verification (private scheme) the time required for the owner to verify that the $\mathrm{CorroboratingEvidence}$ received from the corroborator matches the evidence created in the creation phase

Alibi Verification the time required for the judge to verify an alibi claim by examining the $\mathrm{CorroboratingEvidence}$, $\mathrm{OwnerStatement}$ and associated verification secret

4.8.2 Storage

In our public corroborator scheme, the alibi owner must retain h_o (160 bytes), x_o (120 bytes), the corroborator's signature (256 bytes) and the owner's context value (variable size) to claim an alibi at a later time.

In our private scheme, the owner must retain h_o (160 bytes), x_o (120 bytes), and the owner's context value (variable size). In order to claim the alibi later, the owner must also store h_c (160 bytes), y_c (20 bytes), and r_c (120 bytes), as well as the corroborator's context (variable size).

When the owner receives $\mathrm{CorroboratingEvidence}$ from the corroborator, they must store the corroborator's signature (256 bytes) to claim the alibi.

4.9 Discussion of Alternate Approaches

As mobile devices are becoming more popular, researchers are becoming increasingly interested in "location-based services" [39, 44]. Existing work such as [63] describe how to create location proofs to show that you were in a particular place, but these systems lack user control over their privacy.

Some researchers [15] are concerned about the security and privacy implications of such systems. Studies show that users are sometimes hesitant to share data about their current location with others [48], which motivates further research in privacy-based approaches.

While there are many general frameworks for privacy in location services [28,37,54], most of the approaches only seek to prevent disclosure of user identities entirely rather than leaving the user in control of this information. Systems like Nymbler [43] allow pseudonyms to be

correlated after a certain point in time, but does not provide the facilities to allow users to identify themselves in only specific exchanges as required by our alibi system. The SMILE system [53] provides a "missed encounters" service in a system where mobile devices perform passive key exchanges opportunistically. In SMILE, the results of these exchanges later requires both parties to participate when entities are claiming to have participated in the exchange. Parties connected in the SMILE scheme only means that they may have shared an encounter (exchanged an ephemeral key), but this is not bound to specific identities or locations. The results of these exchanges can be transferred to other users or be claimed to have taken place in a different context and so are unsuitable for alibis. SmokeScreen [21] is a system that allows users with existing relationships in the same area to share presence information, but requires a central, trusted broker server to reveal identities of the participants.

vPriv [60] is system for location-based vehicular services that protect driver privacy. In vPriv the emphasis is on allowing a server to perform functions on the path of a car (e.g., time/location pairs) without learning the identity of the driver for all time/location pairs. Due to the nature of these vehicular services, vPriv is only concerned with preventing widespread spoofing/cheating. vPriv detects cheating by performing random "spot checks," which reveal the identity of the user (without their consent or participation). In our scheme, there is no such trusted party that can reveal the identity of a user associated with an alibi without the owner's consent. vPriv's random spot checks are likely to catch users who cheat frequently, but it is very likely that a single faked record will go undetected. While this is acceptable in their setting, a single forged alibi may have tremendous consequences. Our schemes prevent users from forging even a single alibi successfully.

VeriPlace [52] is a different location proof architecture that has similar goals for protecting user privacy. They use wireless access points as the "corroborators" in their location proofs. The biggest difference between VeriPlace and our design is that VeriPlace requires each corroborator to have a permanently fixed, publicly-known location. Our scheme is much more flexible, as corroborators may move about and establish alibis whenever they encounter other users, allowing a much wider range of devices to be used as corroborators.

In APPLAUS [74], pseudoynms are used in location proofs to protect the privacy of the

user. This scheme requires a trusted, third-party certificate authority to maintain a mapping of pseudonyms to real identities. To verify a location proof for a given user, a verifier asks the certificate authority to look up the pseudonyms associated with a user's identity. In contrast, our scheme does not require users to trust a third party to responsibly maintain the mapping between their real identity and the source identifier in our location proof (alibi). In our schemes, the identity associate with an alibi can *only* be revealed by the owner of the alibi.

Our scheme requires a string commitment scheme. While there are many different string commitment schemes, we feel that schemes based on hashing (such as those presented by Damgård et al. [23]) are both simple to implement and efficient in our setting. We used a string commitment scheme introduced by Halevi and Micali [42] as it is practical and provably secure in the unbounded receiver model, though our system could be tweaked to use other schemes.

Chapter 5

Conclusions

It is often difficult for users of software systems to determine whether the systems they use meet the requirements for their environment, and even more challenging to enforce requirements beyond the original design of the system. We show that existing systems can be augmented with new application-level mechanisms to provide the insight into and control over these systems needed to meet these goals. By applying this philosophy to three different types of systems, we show that augmenting software systems with application-level mechanisms can provide the flexibility and power required for useful policies and the practicality to deploy in real-world settings.

We created RetroSkeleton to enable Android device users to observe and control the behavior of third-party apps on their mobile devices without requiring app source code or platform modifications. We developed a variety of useful policies and apply them to the most popular apps on Google Play. We designed our system to be a flexible, general app analysis and rewriting system in the hopes that it will be useful to users, organizations, and researchers alike.

We built DBTaint to allow web service administrators to leverage information flow tracking systems in a way not previously possible in multi-application web service architectures. Administrators of these services can use this to protect their users and infrastructure from attack and better understand the relationship between users and their data in the system. Our system can be deployed to web services without requiring any changes to the web application or to the database engine. We hope the demonstration of the effectiveness of our system on

two real-world web services will encourage others to benefit from information flow tracking in their own services.

Our two privacy-preserving alibi schemes enable new ways for users to benefit from existing mobile devices and location services. By decoupling the process of establishing evidence of a past location from revealing the identity of the participants, we empower users to participate in opportunistic alibi creation while remaining in control of their privacy. Our schemes are designed to work within the existing infrastructure, without requiring a trusted-third party to fulfill the security and privacy guarantees, and are efficient enough in computation and space for use on mobile devices.

5.1 Acknowledgments

While I was the primary researcher on each of the projects described here, much of this work was conducted in collaboration with my wonderful colleagues. My thanks to all who worked alongside me as well as those who provided valuable feedback and encouragement along the way, especially those mentioned below.

Chapter 2 contains research [25, 27] done in collaboration with Hao Chen, Ben Sanders, and Armen Khodaverdian. Portions of this chapter may be found in the following publications:

RetroSkeleton: Retrofitting Android Apps. Benjamin Davis and Hao Chen. Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys), 2013.

I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. Proceedings of IEEE Mobile Security Technologies (MoST), 2012.

Chapter 3 contains research [24] done in collaboration with Hao Chen. Portions of this chapter may be found in:

DBTaint: Cross-Application Information Flow Tracking via Databases. Benjamin Davis and Hao Chen. Proceedings of the USENIX Conference on Web Applications (WebApps), 2010.

Chapter 4 contains research [26] done in collaboration with Hao Chen and Matthew K.

Franklin. Portions of this chapter may be found in:

Privacy-Preserving Alibi Systems. Benjamin Davis, Hao Chen, and Matthew Franklin. Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2012.

REFERENCES

[1] Adblock Plus. http://adblockplus.org. Accessed: 2012/12/10.

[2] Adblock Plus for Android. http://adblockplus.org/en/android-about. Accessed: 2012/12/10.

[3] Android. http://www.android.com/. Accessed: 2013/12/20.

[4] Best Practical: Request Tracker. http://bestpractical.com/rt/.

[5] dex2jar: Tools to work with Android .dex and Java .class files. http://code.google.com/p/dex2jar/. Accessed: 2012/12/10.

[6] Google Play. https://play.google.com/. Accessed: 2013/12/20.

[7] JForum. http://jforum.net/.

[8] NoScript Firefox Extension. http://noscript.net. Accessed: 2012/12/10.

[9] OWASP top 10 2007. http://www.owasp.org/index.php/Top_10_2007.

[10] SANS: Top 20 internet security problems, threats and risks. http://www.sans.org/top20/.

[11] Zql: Java SQL Parser. http://www.gibello.com/code/zql/.

[12] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net, 2012. Accessed: 2012/12/10.

[13] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[15] A. Blumberg and P. Eckersley. On Locational Privacy, and How to Avoid Losing it Forever. *Electronic Frontier Foundation*, 2009. Technical Report.

[16] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. *Adaptable and Extensible Component Systems*, 2002.

[17] P. Buneman, S. Khanna, and W. C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 316–330, London, UK, 2001. Springer-Verlag.

[18] E. Butler. Firesheep. http://codebutler.com/firesheep/. Accessed: 2012/12/10.

[19] A. Chander, J. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *DARPA Information Survivability Conference & Exposition II (DISCEX)*, volume 2, pages 27–40. IEEE, 2001.

[20] E. Chin and D. Wagner. Efficient Character-Level Taint Tracking for Java. In *SWS '09: Proceedings of the 2009 ACM Workshop on Secure Web Services*, pages 3–12, New York, NY, USA, 2009. ACM.

[21] L. P. Cox, A. Dalton, and V. Marupadi. SmokeScreen: Flexible Privacy Controls for Presence-Sharing. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 233–245, New York, NY, USA, 2007. ACM.

[22] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural Support for Protecting Control Data. *Transactions on Architecture and Code Optimization*, 3:359–389, 2006.

[23] I. Damgård, T. P. Pedersen, and B. Pfitzmann. On the Existence of Statistically Hiding Bit Commitment Schemes and Fail-Stop Signatures. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 250–265, London, UK, 1994. Springer-Verlag.

[24] B. Davis and H. Chen. DBTaint: Cross-Application Information Flow Tracking via Databases. In *USENIX Conference on Web Applications*, Boston, MA, 2010.

[25] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, Taipei, Taiwan, 2013. ACM.

[26] B. Davis, H. Chen, and M. Franklin. Privacy-Preserving Alibi Systems. In *7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Seoul, South Korea, 2012.

[27] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *IEEE Mobile Security Technologies (MoST)*, San Francisco, CA, USA, 2012.

[28] M. Duckham and L. Kulik. A Formal Model of Obfuscation and Negotiation for Location Privacy. *Pervasive Computing*, pages 152–170, 2005.

[29] EFF. HTTPS-Everywhere. `https://www.eff.org/https-everywhere/`. Accessed: 2012/12/10.

[30] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[31] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[32] U. Erlingsson and F. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246 –255, 2000.

[33] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 2002.

[34] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 50–61. ACM, 2012.

[35] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[36] B. Garner. *Black's Law Dictionary*. Thomson/West, Belmont, 2004.

[37] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K. Tan. Private Queries in Location Based Services: Anonymizers are not Necessary. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2008.

[38] Google. Dashboards – Android Developers. `https://developer.android.com/about/dashboards/index.html`. Accessed: 2013/10/17.

[39] K. Gratsias, E. Frentzos, V. Delis, and Y. Theodoridis. Towards a Taxonomy of Location Based Services. *Web and Wireless Geographical Information Systems*, pages 19–30, 2005.

[40] B. Gruver. smali: An Assembler/Disassembler for Android's dex Format. `https://code.google.com/p/smali/`. Accessed: 2012/12/10.

[41] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–18, 2009.

[42] S. Halevi and S. Micali. Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 201–215, London, UK, 1996. Springer-Verlag.

[43] R. Henry, K. Henry, and I. Goldberg. Making a Nymbler Nymble using VERBS. Technical report, CACR 2010-05, Centre for Applied Cryptographic Research, Waterloo, ON, Canada, 2010.

[44] J. Hightower and G. Borriello. Location Systems for Ubiquitous Computing. *Computer*, 34:57–66, August 2001.

[45] A. Ho, M. Fetterman, C. Clark, A. War, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, 2006.

[46] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652. ACM, 2011.

[47] Y.-w. Huang, F. Yu, C. Hang, C.-h. Tsai, D. T. Lee, and S.-y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–51, 2004.

[48] L. Jedrzejczyk, B. Price, A. Bandara, and B. Nuseibeh. On the Impact of Real-Time Feedback on Users' Behaviour in Mobile Location-Sharing Applications. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, pages 1–12. ACM, 2010.

[49] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14. ACM, 2012.

[50] A. Klyubin. Some SecureRandom Thoughts – Android Developers Blog. `http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html`. Accessed: 2013/8/20.

[51] M. S. Lam, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2008.

[52] W. Luo and U. Hengartner. VeriPlace: A Privacy-Aware Location Proof Architecture. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 23–32, New York, NY, USA, 2010. ACM.

[53] J. Manweiler, R. Scudellari, and L. Cox. SMILE: Encounter-Based Trust for Mobile Social Services. *CCS*, 2009.

[54] C. Mascetti, X. Wang, and S. Jajodia. Anonymity in Location-based Services: Towards a General Framework. In *MDM '07: Proceedings of the 2007 International Conference on Mobile Data Management*, pages 69–76, Washington, DC, USA, 2007. IEEE Computer Society.

[55] S. Micali. Simple and Fast Optimistic Protocols for Fair Electronic Exchange. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 12–19, New York, NY, USA, 2003. ACM.

[56] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[57] S. Nanda, L.-c. Lam, and T.-c. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. *ACM/IFIP/USENIX 8th International Middleware Conference (Middleware'07)*, pages 1–20, 2007.

[58] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[59] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[60] R. A. Popa, H. Balakrishnan, and A. J. Blumberg. VPriv: Protecting Privacy in Location-Based Vehicular Services. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 335–350, Berkeley, CA, USA, 2009. USENIX Association.

[61] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[62] D. Reynaud, D. Song, T. Magrino, and R. S. Edward Wu. FreeMarket: Shopping for Free in Android Applications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[63] S. Saroiu and A. Wolman. Enabling New Mobile Applications with Location Proofs. In *HotMobile '09: Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, pages 1–6, New York, NY, USA, 2009. ACM.

[64] N. Sastry, U. Shankar, and D. Wagner. Secure Verification of Location Claims. In *Proceedings of the 2nd ACM Workshop on Wireless Security*, WiSe '03, pages 1–10, New York, NY, USA, 2003. ACM.

[65] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[66] Symantec Security Response. Android Cryptographic Issue May Affect Hundreds of Thousands of Apps. `http://www.symantec.com/connect/blogs/android-cryptographic-issue-may-affect-hundreds-thousands-apps`. Accessed: 2013/8/20.

[67] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems*, 25(4):11, 2007.

[68] B. Weiser. After MetroCard Alibi, Murder Charges Are Dropped. `http://cityroom.blogs.nytimes.com/2008/12/31/after-metrocard-alibi-murder-charges-are-dropped/`. Accessed: 2011/05/04.

[69] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 27–27. USENIX Association, 2012.

[70] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Conference on Security Symposium*. USENIX Association, 2006.

[71] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[72] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[73] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-Stealing Smartphone Applications (on Android). *Trust and Trustworthy Computing*, pages 93–107, 2011.

[74] Z. Zhu and G. Cao. APPLAUS: A Privacy-Preserving Location Proof Updating System for Location-based Services. In *INFOCOM, 2011 Proceedings IEEE*, pages 1889–1897, April 2011.