Abstract Semantics for Software Security Analysis

Kevin Zhijie Chen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-210
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-210.html

November 20, 2015

Abstract Semantics for Software Security Analysis

by

Zhijie Chen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor David Wagner
Professor John Chuang

Fall 2015

Abstract Semantics for Software Security Analysis

Abstract

Abstract Semantics for Software Security Analysis

by

Zhijie Chen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Program analysis and formal methods have enabled advanced automatic software security analysis such as security policy enforcement and vulnerability discovery. However, due to the complexity of the modern software, recent applications of such techniques exhibit serious usability and scalability problems. In this thesis, we address these problems using automatically or semi-automatically constructed abstract program semantics. Specifically, we study two typical scenarios where the power of formal techniques is limited by the problems above, and develop novel techniques that address these issues. First, we propose a new algorithm to construct event-based program abstraction, and check contextual security policies under this abstraction. Our approach addresses the usability and scalability problems in the model-checking of security policies in event-driven programs. Second, we propose a synthesis-based algorithm to learn and check web server logic without having access to the server-side source code. The key insight is that the client-side behavior reflects partially the server-side logic, thus we infer server-side logic by observing the client-side's execution. We develop a declarative language to encode our domain specific modeling of common server-side operations, as well as an efficient algorithm to synthesize a server model in that language. In summary, we demonstrate that abstract semantics can bridge the gap between the human and the massive details of the program, and make formal techniques applicable in a large scale.

To my wife, Chang Gao, and my parents, Yuhua Chen and Mingyun Wang.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my advisor, professor Dawn Song, for being an excellent mentor and inspiring me with her tireless passion in exploring unknown areas and being a better ourselves. In addition, I would like to thank my committee members, professor David Wagner, professor John Chuang, and professor George Necula for their guidance, support, and encouragement.

I also want to thank my collaborators and colleagues. During my graduate career, I have collaborated with Guofei Gu, Jose Nazario, Xinhui Han, Jianwei Zhuge, Noah Johnson, Juan Caballero, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, Chia Yuan Cho, Domagoj Babic, Edward Wu, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Martin Rinard, Chao Zhang, Chengyu Song, Zhaofeng Chen, Mehrdad Niknami, Warren He, Devdatta Ackhawe, Preteek Mittal, and others. Furthermore, I would like to thank other graduate students in the security group and teachers of the courses I have taken. I was lucky to study together with such fantastic graduate students and learn from excellent teachers.

Finally, I want to thank my family. This thesis would not be possible without their love, belief and support.

Chapter 2 is based on joint work with Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard and Dawn Song. Chapter 3 is based on joint work with Warren He, Devdatta Ackhawe, Vijay D'Silva, Prateek Mittal and Dawn Song. Earlier versions of these chapters appeared in [14] and [13].

Chapter 1

Introduction

Modern software security analysis leverages program analysis and formal methods to automatically construct abstractions from programs. The invention of new security analysis technologies is in large the invention of new abstractions and algorithms that expose the program properties in concern. In computer science, abstraction is the process of using computational or statistical structures to represent the abstract semantics of the software or the hardware. By thinking abstractly, we derive general rules and concepts from the tedious details and reason about the commonalities more efficiently. Depending on the goal, a system can have several abstraction layers, exposing different aspects and amounts of details. Nowadays, mobile and web-based software are becoming increasingly popular. However, due to the increase of software complexity, recent applications of traditional program analysis and formal techniques exhibit serious usability and scalability problems. In my thesis, I will introduce two novel abstractions tailored for mobile and web-based software security analysis. In the first abstraction, interactions between the user, the mobile applications, and the operating system are revealed in terms of events and permission uses. In the second abstraction, protocols between multiple web-services are synthesized from observed execution behaviors. These abstract semantics enable the analysis of program behaviors that are beyond the reach of traditional security analysis techniques.

Contextual policy enforcement in Android applications with permission event graphs: The difference between a malicious and a benign Android application can often be characterized by context and sequence in which certain permissions and APIs are used. In the first part of my thesis, I will present a new technique for checking temporal properties of the interaction between an application and the Android event system. The system that implements this technique, called Pegasus, can automatically detect sensitive operations being performed without the user's consent, such as recording audio after the stop button is pressed, or accessing an address book in the background. The algorithms center around a new abstraction of Android applications, called a Permission Event Graph, which we construct with static analysis, and query using model checking. Pegasus has been evaluated for checking application-independent properties on 152 malicious and 117 benign applications,

and application-specific properties on 8 benign and 9 malicious applications. In both cases, Pegasus can detect, or prove the absence of malicious behavior beyond the reach of existing techniques.

Iterative security analysis of web protocols using synthesized models: How to perform a systematic security analysis of web applications is a challenging and open question. Lack of visibility into server-side code makes white-box static/symbolic analysis inapplicable, while approaches based on formal verification are impeded due to the lack of application specifications. To address this challenge, we develop an approach and a system called WebSyn, that enables analysts to find security vulnerabilities in the *implementations* of web applications. The key novelty of our approach is the use of 1). a domain specific language (DSL) to provide a set of high-level building blocks that are common in web protocol models, and 2). program synthesis techniques to automatically model and verify web applications in a highly-customized and efficient search space. WebSyn first uses program synthesis to construct models of protocols, in terms of the DSL, from executions of web applications. Next, WebSyn uses the Alloy analyzer to discover potential vulnerabilities in the synthesized model. Based on the analysis results, the analysts can refine the verification by providing more example executions, or suggesting or removing possible attacks. In 3 proof-of-concept case studies, WebSyn demonstrates the discovery of complex vulnerabilities in implementations of real world web applications. WebSyn is a step towards leveraging the benefits of program synthesis and domain specific languages for enhancing application security.

Chapter 2

Pegasus: Contextual Policy Enforcement in Android Applications with Permission Event Graphs

2.1 Introduction

Users of smartphones download and install software from application markets. According to the Google I/O keynote in 2012, by June 2012, the official market for Android applications, Google Play, hosted over 600,000 applications, which had been installed over 20 billion times. Despite recent advances in mobile security, there are examples of malware that cannot be detected by existing techniques.

A malicious application can compromise a user's security in several ways. Examples include leaking phone identifiers, exfiltrating the contents of an address book, or audio and video eavesdropping. Consult recent surveys for more examples of malicious behavior [23, 25, 30].

In this chapter, we focus on detecting malicious behavior that can be characterized by the temporal order in which an application uses APIs and permissions. Consider a malicious audio application which eavesdrops on the user by recording audio after the stop-button has been pressed, and a benign one. Both applications use the same permissions, and start and stop recording in response to button clicks. Malware detection based on syntactic or statistical patterns of control flow or permission requests cannot distinguish between these two applications [25, 24, 31]. The difference between the two applications is semantic and requires semantics-based analysis.

The intuition behind our work is that user expectations and malicious intent can be expressed by the context in which APIs and permissions are used at runtime. A user expects that clicking a start or stop button, will respectively, start or stop recording, and further, that this is only way an audio application records. This expectation can be encoded by two API usage policies. The API to start recording audio should be called if and only if the event

handler for the start button was previously called. The API to stop recording should be called if and only if the event handler for the stop button was previously called. Policies requiring that sensitive resources are not accessed by background tasks or in response to timer events can also aid in distinguishing benign from malicious behavior.

We present Pegasus, a system for specifying and automatically enforcing policies concerning API and permission use. Pegasus combines static analysis, model checking, and runtime monitoring. We anticipate several applications of such technology. One is automatic, semantics-based screening for malware. Another is as a diagnostic tool that security analysts can use to dissect potentially malicious applications. A third is to provide fine-grained information about permission use to enable users to make an informed decision about whether to install an application.

Our system can be attacked by malware writers who obfuscate their applications to avoid static detection. However, such obfuscation will trigger our runtime checks and lead to convoluted code structures. Thus an attempt to evade our system and may only result in drawing greater scrutiny to the application.

Problem and Approach

We now describe the challenges behind policy specification and checking in greater detail, and the insights behind our solution.

Problem Definition We consider three closely related problems. The first problem is to design a language for specifying the event-driven behavior of an Android application. The second problem is to construct an abstraction of the interaction between an Android application and the Android event system. The third problem is to check whether this abstraction satisfies a given policy. A solution to these problems would allow us to specify security policies and detect (or prove the absence of) certain malicious behavior.

Challenges Property specification mechanisms typically focus on an application. Executing a task in the background, or calling an API after a button is clicked, are properties of the Android event system, not the application. Specifying policies governing event-driven API use requires a language that can describe properties of an application as well as of the operating system. For example, specifying that audio should not be recorded after a stop button is clicked, requires us to describe an application artefact, such as a button, a system artefact, such as a recording API, and the interaction between the two.

Checking policies of the form above is a greater challenge. Software model checking is a powerful technique for checking temporal properties of programs. Software model checkers construct abstractions of a program and then check properties using flow-sensitive analysis. The execution of an Android application is the result of intricate interplay between the application code and the Android system orchestrated by callbacks and listeners. Constructing an abstraction of such behavior is difficult because control flow to event handlers is invisible

in the code. Moreover, static analysis of event-driven programs has received little attention, and was recently shown to be EXPSPACE-hard [45]. The first analysis challenge is to model control flow between the event system and application.

The second analysis challenge is to design and compute an abstraction that can represent event-driven behavior, but is small compared to the Android system. Most existing techniques abstract data values in a program, but our focus on the Android event system mandates a new abstraction. The challenge in computing an abstraction lies in modelling the Android event system, and dealing with complex heap manipulation in Android programs, and their use of reflection, and APIs from the Java and Android SDK.

Insights We overcome the aforementioned challenges using the insights described next. Our first insight is that though the Android system is a large, complicated object, it changes the state of the application using a fixed set of event handlers. It suffices for a policy language to express event handlers, APIs, and certain arguments to APIs to specify the context in which an application uses permissions.

Even a restricted analysis of the Android event system or event handlers defined in an application is not feasible due to the size of the code and the state-space explosion problem. Our second insight is to use a graph to make the interaction between an application and the system explicit. We introduce *Permission Event Graphs* (PEGs), a new representation that abstracts the interplay between the Android event system, and permissions and APIs in an application, but excludes low-level constructs.

Our third insight is that a PEG can be viewed as a predicate abstraction of an Android application and the Android system, where predicates describe which events can fire next. Standard predicate abstraction engines use theorem provers to compute how program statements transform predicates over data. We implement a new, Android specific, *event semantics engine*, which can compute how API calls transform predicates over the Android event queue.

The final challenge, once an abstraction has been constructed is to check that it satisfies a given policy. We use standard model checking algorithms for this purpose. Detecting sequences or repeating patterns in an application can be implemented using basic graph-theoretic algorithms for reachability and loop detection.

Our experience suggests that PEGs reside in a sweet-spot in the precision-efficiency spectrum. Our analysis based on PEGs is more precise than existing syntactic analyzes and is more expensive to construct. However, we gain efficiency because a single PEG can be queried to check several policies pertaining to a single application.

Content and Contributions

In this chapter, we study the problem of detecting malicious behavior that manifests via patterns of interaction between an application and the Android system. We design a new abstraction of the context in which event handlers fire, and present a system for specifying, computing and checking properties of this abstraction. We make the following contributions:

1. Permission Event Graphs: A novel abstraction of the context in which events fire, and event-driven manner in which an Android application uses permissions.
2. Encoding user and malicious intent: We encode user expectations and malicious behavior as temporal properties of PEGs.
3. PEG construction: We devise a static analysis algorithm to construct PEGs. The algorithm computes a fixed point involving transformers, generated by the program, the event mechanism, and APIs. Our event model supports 63 different event handling methods in 21 Android SDK classes.
4. PEG analysis: We implement Pegasus, an automated analysis tool that takes as input a property, and checks if the application satisfies that property.
5. Experiments: We check 6 application-independent properties of 269 applications, and check application-specific properties of 17 applications. Pegasus can automatically identify malicious behavior, which was previously discovered by manual analysis.

The chapter is organized as follows: We summarize background on Android and introduce our running example in Section 2.2. PEGs are formally defined in Section 2.3 and can be constructed using the algorithm in Section 2.4. The details of our system appear in Section 2.5, followed by our evaluation in Section 2.6.

2.2 Background and Overview

In this section, we give an overview of the Android platform as relevant for this chapter and illustrate PEGs with a running example.

Android

Android is a computing platform for mobile devices. It includes a multi-user operating system based on Linux, middleware, and a set of core applications. Users install third-party applications acquired from application markets. An *Android package* is an archive (`.apk` file) containing application code, data, and resource information.

Applications are typically written in Java but may also include native code. Applications compile into a custom *Dalvik executable format* (`.dex`), which is executed by the Dalvik virtual machine.

Permissions A *permission* allows an application to access APIs, code and data on a phone. Permissions are required to access the user's contacts, SMS messages, the SD card, camera, microphone, Bluetooth, and other parts of the phone. All permissions required by an application must be granted by a user at install time.

The Manifest Every application has a *manifest file* (`AndroidManifest.xml`) describing the application's requirements and constituents. The manifest contains component information,

the permissions required, and Android API version requirements. The component information lists the components in an application and names the classes implementing these components.

Components The building blocks of Android applications are *components*. A component is one of four types: activity, service, content provider, and broadcast receiver, each implemented as a subclass of `Activity`, `Service`, `ContentProvider`, and `BroadcastReceiver`, respectively. An *activity* is a user-oriented task (such as a user interface), a *service* runs in the background. a *content provider* encapsulates application data, and a *broadcast receiver* responds to broadcasts from the Android system. Components (consequently, applications) interact using typed messages called *intents*.

Lifecycles A lifecycle is a pre-defined pattern governing the order in which the Android system calls certain methods. An application can define callbacks and listeners that contribute to the lifecycle.

An activity is started using the `startActivity` or `startActivityForResult` API calls. During execution, an activity may be *running*, meaning it is visible and has focus, *paused*, if it is visible but not in focus, or *stopped* if it is not visible. Application execution usually begins in an activity. A service may be *started* or *bound*. A service is started if a component calls `startService`, following which the service runs indefinitely, even if the component invoking it dies. The `bindService` call allows components to bind to a service. A bound service is destroyed when all components bound to it terminate.

Events and APIs Events and APIs are the two ways an Android application interacts with the system. We define an *event* as a situation in which the Android system calls application code. Examples of events are finger taps, swipes, SMS notifications, and lifecycle events. The code that is called when an event occurs is called an *event handler*. We define an API to be a system defined function, which applications can call. In this chapter, we are concerned with event and permission APIs. An *event* API is one that changes how events are handled, such as registering a `Button.onClick` listener, or making a button invisible. A *permission* API is one requires a permission, such as the `LocationManager.requestLocationUpdates` API, which requests the location information about the phone, and requires either the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permissions, depending on the arguments. We obtain permission information from permission maps [3, 31].

Overview and Running Example

We now demonstrate the concepts in this chapter with a running example, as well as how we envision the system being used. Consider a malicious audio recording application, which eavesdrops on the user. On startup, the application displays the interface shown in Figure 2.1. This interface is implemented as a `Recorder` *activity* and contains two buttons, `REC` and `STOP`.

Figure 2.1: User interface for the running example. The application records audio in a background service after the user has clicked the stop button.

Figure 2.2: Permissions requested by the recording application during installation.

Initially, only REC is clickable. Clicking REC initiates recording, makes STOP clickable, and disables REC from being clicked. Clicking STOP terminates recording, enables REC, and disables STOP. When the application is started, it registers a service, which creates a system timer callback, which is invoked every 15 minutes. The callback function records 3 minutes of audio and stores it on the SD card. Since services run in the background, this application will eavesdrop even after the recorder application is closed.

We now consider two problems: How can we precisely define malicious behavior such as surreptitious recording? How can we automatically detect such behavior?

Defining Malicious Intent Rather than define malicious intent, we focus on defining user intent, or user expectations. In our example, the details of *how* recording happens is determined by the developer, but a user expects to be defining *when* recording happens. Moreover, the user expects that clicking REC will start recording, that clicking STOP will stop recording, and that this is the only situation in which recording occurs. This expectation contains a logical component and a temporal component, and can be formally expressed by a temporal logic formula.

$$(\neg\text{Start-Recording}\ \mathbf{U}\ \text{REC.onClick}) \wedge (\text{Stop-Recording} \iff \text{STOP.onClick})$$

This formula, in English, asserts that the *proposition* Start-Recording does not become true until the proposition REC.onClick is true, and that Stop-Recording is true if and only if STOP.onClick is true. Such a formula is interpreted over an execution trace. REC.onClick and STOP.onClick are true at the respective instants in a trace when the eponymous buttons are clicked. The propositions Start-Recording and Stop-Recording are true in the respective instants when the APIs to start and stop recording are called.

A second example of user expectation is that an SMS is not sent unless the user performs an action, such as clicking a button. A third example is that when an SMS arrives, the

user is notified. These properties can be expressed by the two formula below. The second formula expresses that a broadcast message (such as an SMS notification) is not aborted by the application.

$$\neg\text{Send-SMS } \mathbf{U} \text{ Button.onClick}$$
$$\neg\text{BroadcastAbort}$$

The three formula above fall into two different categories. The SMS and broadcast properties are application independent. They can be checked against all applications, and are part of a cookbook of generic properties we have developed. The properties about recording are application specific and have to be written by the analyst.

The set of propositions is defined by our tool, and includes permissions, API calls, certain event handlers, and constant arguments to API calls. To aid the analyst, we have implemented a tool that extracts from an application's manifest, the names and types of user interface entities such as buttons and widgets, and their relevant event handlers.

We express user intent with formula. We say that an application exhibits *potentially malicious intent* if it does not satisfy a user intent formula. Our tool Pegasus automatically checks if an application satisfies a formula. If an application violates a property, Pegasus provides diagnostic information about why the property fails. The analyst has to decide if failure to respect user intent is indeed malicious. We discuss this issue in greater detail later.

Detecting Potentially Malicious Intent How can we determine if an Android application respects a formula specifying user intent? Figure 2.2 depicts the permissions requested by the recorder during installation. Techniques that only examine permission requests [3, 31, 32] will only know that the application uses audio and SD card permissions. Since control flow between the Android system and event handlers is not represented in a call graph, structural analysis of call graphs [23, 35], will not identify the behaviors discussed above.

The challenge in checking temporal properties is to construct an abstraction satisfying two requirements: It must be small enough for model checking to be tractable. It must be large enough to avoid generating a large number of false positives. Permissions used by an application, call graphs, and control flow graphs can be viewed as abstractions that can be efficiently analyzed but do not satisfy the second requirement. We now describe an abstraction that enriches permission sets and call graphs with information about event contexts.

Permission Event Graphs We have devised a new abstraction called a Permission Event Graph (PEG). In a PEG, every vertex represents an event context and edges represent the event handlers that may fire in that context. Edges also capture the APIs and permissions that are used when an event handler fires. Since permissions such as those for accessing contact lists, are determined by APIs calls and the argument values, knowledge of APIs does not subsume permissions. Example information that a PEG can represent is that clicking a spe-

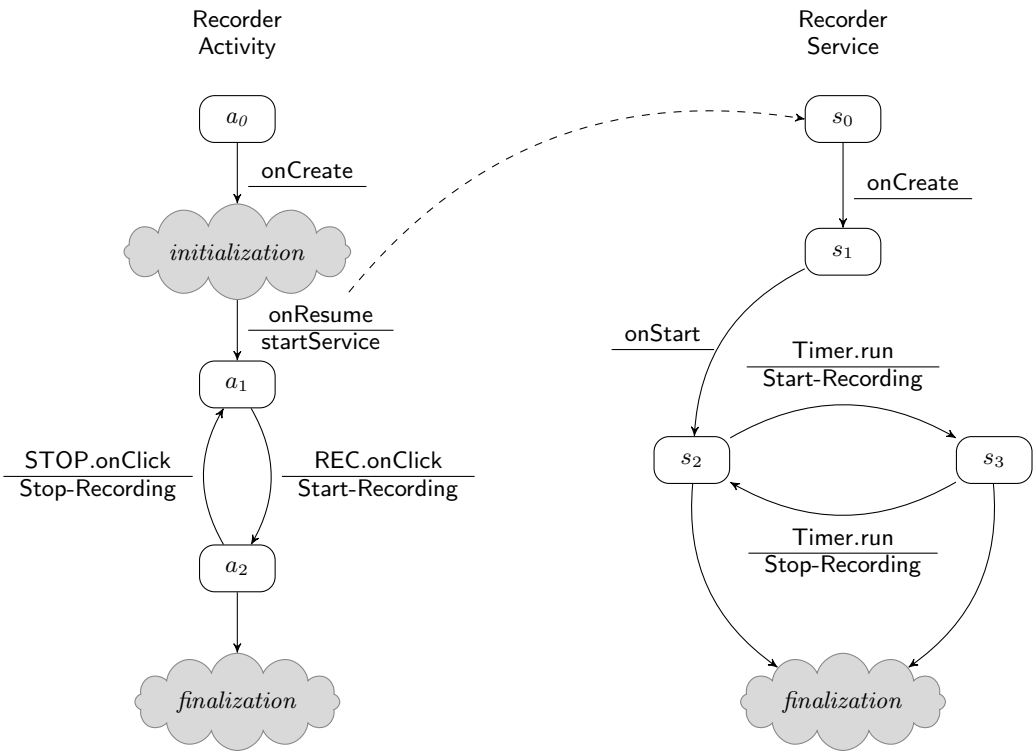

Figure 2.3: Permission Event Graph for the running example. Vertices represent event contexts and an edge label $\frac{E}{A}$ represents that when the event handler E fires, an action A is performed. Dashed edge represent asynchronous tasks.

cific button causes the `READ_CONTACTS` permission to be used, while the `ACCESS_FINE_LOCATION` permission is used in a background task.

A portion of the PEG for the running example is shown in Figure 2.3. Every vertex represents an event context. There are two types of edges. Solid edges represent synchronous behavior, and dashed edges represent asynchronous behavior. We refer to the firing of one or more event handlers as an *event* and the use of one or more APIs and permissions as an *action*. An edge label $\frac{E}{A}$ represents that when the event E occurs, the action A is performed.

Figure 2.3 shows that when the event handler REC.onClick is called, the action denoted Start-Recording occurs. This action represents calling an API to start recording. We have omitted portions of the PEG related to the activity initialisation, destruction, and the service lifecyle. Next, the event STOP.onClick is enabled, and when it occurs, causes the Stop-Recording action. The dashed edge from onResume indicates an asynchronous call to start a service.

The PEG captures semantic information about an application that is not computed by existing techniques. For example, we see that there are two distinct contexts in which the audio is recorded. We also see that recording stops if we click STOP, but this is not the only

way to stop recording.

Examining the PEG reveals that the application records audio even if REC is not clicked. Moreover, we can determine the sequence of events leading to this malicious behavior: a new service is started, a timer is then created, and timer events start recording. PEGs generated in practice are too large to examine manually. In such cases, specifications can be treated as queries about the application, and model checking can be used to answer such queries.

Security Analysis with PEGs The techniques we develop have several uses. All the uses follow the workflow of starting with a set of properties, automatically constructing a PEG for an application and model checking the PEG, manually examining the results of model checking, and repeating this process if required.

There are several kinds of properties that an analyst can check. We have developed a cookbook of application-independent properties, such as background audio or video recording. An analyst can write application-specific properties to check that an application functions as expected. For example, clicking REC should start recording, and STOP should stop recording. An analyst can also pose questions about the behavior of specific event handlers: Does clicking the STOP button stop recording? If the application is sent to the background, will recording continue or stop? If the application is killed while recording, will the data be saved to the SD card? All these questions can be encoded as temporal properties.

Our tool Pegasus can be used to automatically construct the PEG for an application and model check the PEG. If a property is satisfied, the analyst will have to check if it was too general, and try a more specific property. If a property is not satisfied, the model checker will generate a counterexample trace: a sequence of events and actions violating the property. The analyst has to examine the trace and see if it is symptomatic of malicious behavior. If the behavior is potentially malicious, the analyst will have to reproduce it at runtime. If the behavior is benign, the analyst will have to strengthen the property that is checked to narrow the search for malicious behavior.

To summarize, a vocabulary based on events and actions allows for describing a new family of benign and malicious behavior beyond the reach of existing specification mechanisms. Events are a runtime manifestation of user interaction with an application, and actions describe an application's response. Specifications involving events and actions allow us to encode user intent in mechanical terms. From a user's perspective, a PEG summarizes the dialogue between a user (via events) and an application. From an algorithmic perspective, a PEG is a data-structure encoding the interaction between the Android event system (via calls to event handlers) and application code.

2.3 An Abstraction of Android Applications

The contributions of this section are a formal definition of PEGs, and a symbolic encoding of PEGs.

Transition Systems from Android Apps

An Android application defines an infinite-state transition system, which describes the runtime behavior of an application. The transition system we define makes the control flow and event-relationships in an application explicit. It is a mathematical object that we never construct, but it informs the design of our analysis.

States A *runtime state*, or just state, is composed of an application state and a system state. The *application state* consists of the application program counter, a valuation of program variables, and the contents of the stack and the heap. The *system state* consists of the contents of the event queue, event handlers and listeners that are enabled, and other global system states. The set of states

$$\mathit{State} = \mathit{App\text{-}States} \times \mathit{Sys\text{-}States}$$

contains all combinations of application and system states. These sets include unreachable states. A state $\sigma = (p, s)$, consists of an application state p and a system state s. We denote the set of initial states of execution as Init.

Transitions An application changes its internal state by executing statements, and changes the system state by making API calls. Conversely, the system may change its own state or change application state by calling event handlers. A *transition* is the smallest change in the state of an application or system, and

$$\mathit{Trans} \subseteq \mathit{State} \times \mathit{State}$$

is the *transition relation* of an application. A transition t is caused by executing a statement, denoted $\mathit{stmt}(t)$, in the application or system code. We call t is an *application transition* if $\mathit{stmt}(t)$ is in the application code and is a *system transition* otherwise.

Transition Systems The evolution of an application and the Android system over time is mathematically described by a *transition system*

$$T = (\mathit{State}, \mathit{Trans}, \mathit{Init}, \mathit{stmt})$$

consisting of a set of states State, a transition relation Trans, a set of initial states Init, and a function stmt that labels transitions with statements.

Traces We formalize the execution of the application in the system. A *trace* of T is a sequence of states $\pi = \pi_0, \pi_1, \ldots$ in which π_0 is an initial state and every pair (π_i, π_{i+1}) is a transition. We write $\mathit{traces}(\sigma)$ for the set of traces originating from σ, and write $\mathit{traces}(T)$ for the set of traces of T. A trace contains complete information about application and system transitions.

Permission Event Graphs

The transition system defined by an application is infinite and checking its properties is undecidable in general. The standard approach to addressing such undecidability is to construct an *abstraction* of this transition system. We now introduce *Permission Event Graphs*, a variation of transition systems, which can represent finite-state abstractions of Android applications.

Example 1. Revisit the PEG for the running example in Figure 2.3. A vertex in the figure is called an abstract state. The abstract state a_1 represents all possible runtime states in which the REC.onClick event handler may be called in the recorder activity. An edge in the figure is an abstract transition. The abstract transition from a_1 to a_2 has a label representing that if the event handler REC.onClick is called, the application will disable the REC, enable the STOP, start recording, and transition to the state a_2. The dashed edge is an *asynchronous transition*, representing that the action Start-Recording launches an asynchronous task. In this case, a service is started. ◁

A formal definition of PEGs follows. We use the prefix "a" to indicate sets used as abstractions. We write $\mathcal{P}(S)$ for the set of all subsets of a set S.

We define an event to be a set of event handlers. For example, the event STOP.onClick represents a single event handler. We can also define an event onClick that corresponds to all event handlers which may be called when a button is clicked. Formally, let *Handler* be a set of event handlers and *Event* be a set of symbols, each representing one or more event handlers.

Definition 1. A *Permission Event Graph* (PEG) over a set of event symbols *Event* and APIs API is a tuple

$$PEG = (aState, aTrans, bTrans, aInit)$$

consisting of the following.

- A set of abstract states *aState*. Every abstract state represents a set of runtime states, which form the context of an event.
- A labelled transition relation $aTrans \subseteq aState \times Event \times \mathcal{P}(\mathsf{API}) \times aState$, where each transition (s_1, E, A, s_2), represents that in state s_1, the event E may fire, and causes the APIs in A to be called, leading to abstract state s_2.
- A relation $bTrans \subseteq \mathcal{P}(\mathsf{API}) \times aState$, where each tuple (A, s), represents that the action A causes an asynchronous transition to the abstract state s.
- A set *aInit* of abstract initial states.

PEGs are different from control flow graphs, call graphs, and other standard graph-based abstractions of programs. A PEG is different from a control flow graph because it does not represent the syntactic structure of source code. A PEG only contains calls to system APIs, rather than all calls, as in a call graph, but also includes the values of arguments, hence is

related, but incomparable (mathematically) to a call graph. We use the word "Permission" in the name because permissions are determined by calls to APIs and their arguments. We use the word "Event" to emphasise that state transitions represent the effect of firing event handlers.

We use graph algorithms to analyze PEGs. To derive the PEGs of an application efficiently, our abstraction engine uses the symbolic encoding introduced next.

A Symbolic Encoding of PEGs

We now devise a compact encoding of PEGs. Our encoding uses Boolean variables to represent PEG states and labels to represent actions, and can be exponentially more succinct than representing a PEG as a labelled graph.

Mode Variables and Event-Modalities We first encode PEG states using Boolean variables. Define a set *ModeVars* of Boolean-valued *mode variables*. The *Boolean encoding* of an abstract state is a function

$$s : \mathit{ModeVars} \to \{\mathsf{true}, \mathsf{false}\}$$

that assigns truth values to mode variables. The number of mode variables we need is logarithmic in the number of states of a PEG.

A Boolean formula φ over *ModeVars* represents the set of Boolean encodings that make φ true. Recall that a *literal* is a Boolean variable or its negation, and a *cube* is a conjunction of literals. Let *Cube* be the set of cubes over mode variables in a subset of *ModeVars*. We only use cubes and not arbitrary Boolean formula over *ModeVars* to represent sets of encodings because cubes can be efficiently manipulated, while arbitrary formula cannot. The same encoding choice is used in the SLAM project [7].

We encode abstract transitions using tuples called *event-modalities*. An event-modality is a tuple

$$(\mathit{Pre}, A, \mathit{Post}) \in \mathit{Cube} \times \mathcal{P}(\mathsf{API}) \times \mathit{Cube}$$

consisting of a *precondition* Pre, a set of API labels A, and a *postcondition* Post. An event-modality $(\mathit{Pre}, A, \mathit{Post})$ represents the set of abstract transitions that begin in some abstract state represented by Pre, and transition to some abstract state represented by Post, while causing the action A.

A *symbolic encoding* of a PEG is a tuple

$$\mathit{sPEG} = (\mathit{aInit}, \mathit{EventModality})$$

consisting of a cube *aInit* representing initial states and a set *EventModality* of event-modalities.

Example 2. Consider a Boolean variable *TimerEnabled* and a label Start-Recording. The timer-related behavior in Figure 2.3 can be encoded using the value true for *TimerEnabled*

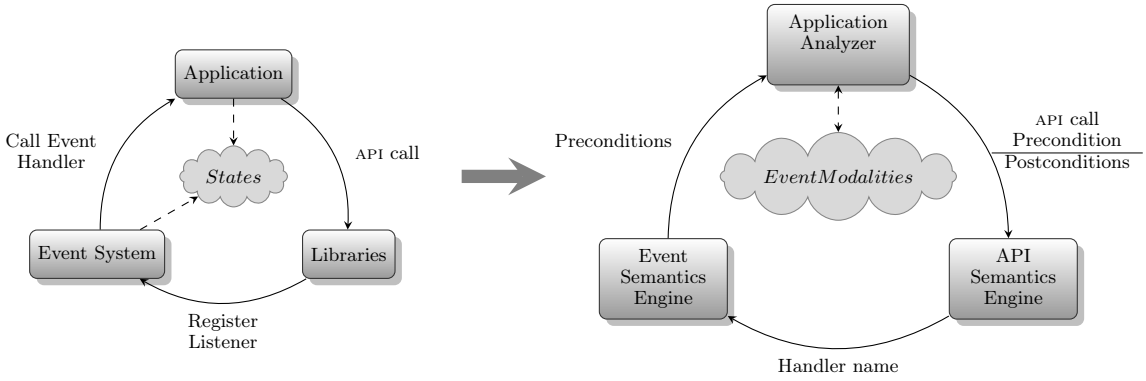

Figure 2.4: Intuition behind the abstraction engine. The system computes event modalities by combining a static analyzer, which abstracts application semantics, an API semantics engine, which abstracts API calls, and an event semantics engine, which abstracts the event system.

to represent s_2 and the value false to represent s_3. The abstract transition from s_1 to s_2 is represented by the event-modality below.

$$(\mathit{TimerEnabled}, \{\mathsf{Start\text{-}Recording}\}, \neg\mathit{TimerEnabled})$$

If the precondition *TimerEnabled* is true, the timer event TIMER.run is enabled. If the event fires, the event handler causes the application to transition to a state satisfying the postcondition $\neg\mathit{TimerEnabled}$. ◁

2.4 The Abstraction Engine

The contribution of this section is a procedure and architecture for constructing PEGs from Android applications. Our implementation of this procedure combines a model of the Android event system and APIs with fixed point iteration in a lattice to derive PEGs.

The Core Algorithm

The interaction between an application, the event mechanism and libraries in an Android application is summarized in the upper part of Figure 2.4. The dashed arrows show that the state of an execution is modified either by executing application code or when the event system fires an event handler. The solid arrows denote calls. An application may call the Android APIs, and if the call is to register a listener, the APIs in turn access the event system.

The architecture we use to compute a symbolic PEG is shown in the lower part of Figure 2.4. Each shaded box represents an engine in our implementation. The different engines interact to compute a set of event-modalities. We abstract application code with a static

analyzer, model event generation and destruction with an *event semantics engine*, and model APIs with an API *semantics engine*.

Our static analyzer determines a set of preconditions, which specify event contexts. When an API call is encountered, the precondition and API name are given to the API semantics engine. If the API modifies the application state, a postcondition is returned to the static analyzer. If the API modifies the system state, the name of the API is given to the event semantics engine. The event semantics engine computes the set of preconditions for that event handler to fire, and the static analyzer had to determine whether to analyze the event handler code. By iterating between these three engines, we derive a set of event-modalities that symbolically encode a PEG for an application.

The functions below formalize these components.

$$\begin{aligned} \mathit{entry} &: \mathit{Handler} \to \mathcal{P}(\mathsf{API}) \\ \mathit{next} &: \mathit{Handler} \times \mathsf{API} \to \mathcal{P}(\mathsf{API}) \\ \mathit{event\text{-}sem} &: \mathit{Handler} \to \mathcal{P}(\mathit{Cube}) \\ \mathit{api\text{-}sem} &: \mathsf{API} \times \mathit{Cube} \to \mathcal{P}(\mathit{Cube}) \\ \mathit{app\text{-}sem} &: \mathit{Handler} \times \mathit{Cube} \to \mathcal{P}(\mathcal{P}(\mathsf{API}) \times \mathit{Cube}) \end{aligned}$$

The function *entry* takes as input an event handler name, retrieves the code, constructs the CFG for the event handler, and retrieves the first set of API calls reachable from the entry of the CFG, without calling other APIs. The function *next* is similar to *entry*. When invoked as $\mathit{next}(h, A)$ on an event handler h with API call A, *next* will return the set C of APIs in h that may be called after calling A, such that there is no API call between A and each API in C. These functions are implemented in the static analyzer, by combining control flow reachability with pointer analysis.

The function *event-sem* takes as input an event handler and returns as output a set of cubes representing preconditions for that event handler to fire. This function is implemented by the event semantics engine.

The function *api-sem* takes as input an API call A and a precondition p and returns a set Q of postconditions. The postconditions satisfy that executing A in a state satisfying p leads to a state satisfying some cube in Q. This function is implemented by the API semantics engine.

The function *app-sem* takes as input an event handler and a precondition, and returns a set of pairs of the form (A, q). Let h be an event handler. Towards formally defining *app-sem*, we define a function

$$\mathit{reach\text{-}sem}_h : \mathcal{P}(\mathcal{P}(\mathsf{API}) \times \mathsf{API} \times \mathit{Cube}) \to \mathcal{P}(\mathcal{P}(\mathsf{API}) \times \mathsf{API} \times \mathit{Cube})$$

that maps a tuple (A, a, p) representing a set of APIs A previous executed, and the API a that will be executed with precondition p to the tuple $(A \cup \{a\}, b, q)$, where b is an API that

can be executed after a and q is the postcondition of executing a when p holds.

$$\begin{aligned} \textit{reach-sem}_h(R) = R \cup \{(A \cup \{a\}, \{b\}, q) \mid & \text{ where} \\ & (A, a, p) \text{ is in } \textit{reach-sem}_h(R), \text{ and} \\ & b \text{ is in } \textit{next}(h, a), \text{ and} \\ & q \text{ is in } \textit{api-sem}(b, p)\} \end{aligned}$$

Note that $\textit{reach-sem}_h$ occurs on both sides of the definition. The function is implemented by fixed point iteration over the CFG to compute a set of action, postcondition pairs. The function *app-sem* computes event modalities by computing $\textit{reach-sem}_h$ and projecting out the action, postcondition pairs that reach the exit point of the event handler. Formally, *app-sem* satisfies the condition below.

$$\begin{aligned} \textit{app-sem}(h, p) = \{(A, q) | & q \in \textit{Cube}, \text{ and} \\ & A = \bigcup_{(B,b,q) \in R} B \cup \{b\}, \\ & \text{where } R = \{(\emptyset, a, p) \mid a \in \textit{entry}(h)\}, \\ & \text{and } \textit{next}(h, b) = \emptyset\} \end{aligned}$$

A pair (A, q) is produced by $\textit{app-sem}(h, p)$ exactly if A is a set of APIs reachable in h, and executing h in a state satisfying p leads to the postcondition q at the exit point of h. We now describe how the precondition p is generated.

Fixed Point We combine the functions above to compute a fixed point whose result is the PEG for a given application:

$$\begin{aligned} \textit{EventModality} = \{(p, A, q) \mid & p \in \textit{event-sem}(h), \\ & \text{and } (A, q) \in \textit{app-sem}(h, p), \\ & \text{and } h \in \textit{Handler}\} \end{aligned}$$

In words, we consider each event handler h in *Handler*, use the event semantics engine to generate preconditions for h to fire, and then combine *app-sem* and *api-sem* to determine the postconditions derived by firing h. The implementation of each function above is discussed below.

Implementation of the Engines

A contribution we make, en route to computing PEGs, is to engineer a static analyzer, an event semantics engine, and an API semantics engine. We discuss implementation details below.

Static Analysis We implement a partially context-sensitive points-to analysis serving two purposes. First, the analysis overapproximates the targets of the method call. Overapproximation arises due to dynamic dispatch, where different executions of a given method call may invoke different methods. The second purpose is computing information about method arguments. For example, consider a call to `Button.setOnClickListener`. The first argument to this method is the event handler to attach as the `onClick` listener. We use the points-to analysis to disambiguate arguments and to overapproximate the set of event handlers the application will attach. Resolving arguments is necessary to derive sufficient information for verification, because an API call can map to different permissions depending on the values of its arguments. For example, a call to the `ContentResolver.query(URI)` method will access the phone's contacts if the URI points to the contacts content provider, while the same API will access the phone's SMS messages for a different URI. The two operations require different permissions.

We augment the context-insensitive analysis for event handlers by propagating the points-to information for method call parameters from the caller to the callee. This provides partial context-sensitivity. In particular, it allows the analysis of sub-functions to reason about values which are computed in parent functions. Our experience shows that this is important to handle, since many applications pass arguments for system APIs through helper functions or wrappers for those APIs. For flow-sensitivity, we use flow-insensitive analysis for class fields and flow-sensitive analysis for local variables to balance efficiency and precision. Our hybrid approach to points-to analysis is similar to the use of object representatives or instance keys [11, 33, 61].

Event Semantics Engine The event semantics engine implements the *event-sem* function. It receives a method handler name as input and returns the preconditions for the handler to execute. This engine models the semantics of events by capturing the context in which an event may fire. We implemented the engine by examining the effect of event handling mechanism as specified in the Android documentation and in the Android platform code. A list of 63 event handlers we model is given in Table 2.1.

API semantics engine The *api-sem* receives as input a method call and a precondition, and generates as output the event-modalities generated by executing the method when that precondition is satisfied, and the postcondition in which the method terminates. Though these event-modalities are determined by the implementation of Android APIs, we *do not* analyze the Android API source code. Instead, we model every API call we have found necessary to support during analysis. Figure 2.5 summarizes the API coverage of the API semantics engine. We support 1200 API calls, which covers over 90% of the call-sites we found on a data set of over $95,000$ applications. The entire list of methods we support is too long to recall here.

Class name	Methods
A.app.Activity	onCreateOptionsMenu, onKeyDown, onOptionsItemSelected, onPrepareOptionsMenu, <init>, onActivityResult, onConfigurationChanged, onCreate, onCreateContextMenu, onDestroy, onPause, onRestart, onResume,
	onSaveInstanceState, onStart, onStop, onWindowFocusChanged
A.app.Dialog	<init>, onCreate
A.app.ListActivity	<init>, onCreate
A.app.Service	onBind, <init>, onCreate, onDestroy, onLowMemory, onStart, onStartCommand
A.content.BroadcastReceiver	<init>, onReceive
A.content.ContentProvider	query, insert, onCreate, delete, update, getType, <init>
A.content.ServiceConnection	onServiceConnected, onServiceDisconnected
A.os.AsyncTask	doInBackground, onPostExecute, onPreExecute
A.os.Handler	handleMessage
A.preference.PreferenceActivity	onPreferenceTreeClick, <init>, onCreate, onDestroy, onStop
A.preference.Preference.OnPreferenceChangeListener	onPreferenceChange
A.preference.Preference.OnPreferenceClickListener	onPreferenceClick
A.telephony.PhoneStateListener	onCallStateChanged
A.view.View.OnClickListener	onClick
A.view.View.OnTouchListener	onTouch
A.webkit.WebChromeClient	onProgressChanged
A.webkit.WebViewClient	shouldOverrideUrlLoading, onPageFinished, onPageStarted, onReceivedError
A.widget.AdapterView.OnItemClickListener	onItemClick
A.widget.AdapterView.OnItemLongClickListener	onItemLongClick
java.lang.Runnable	run, run
java.lang.Thread	run

Table 2.1: Event handling APIs supported by the event semantics engine. A = *android*.

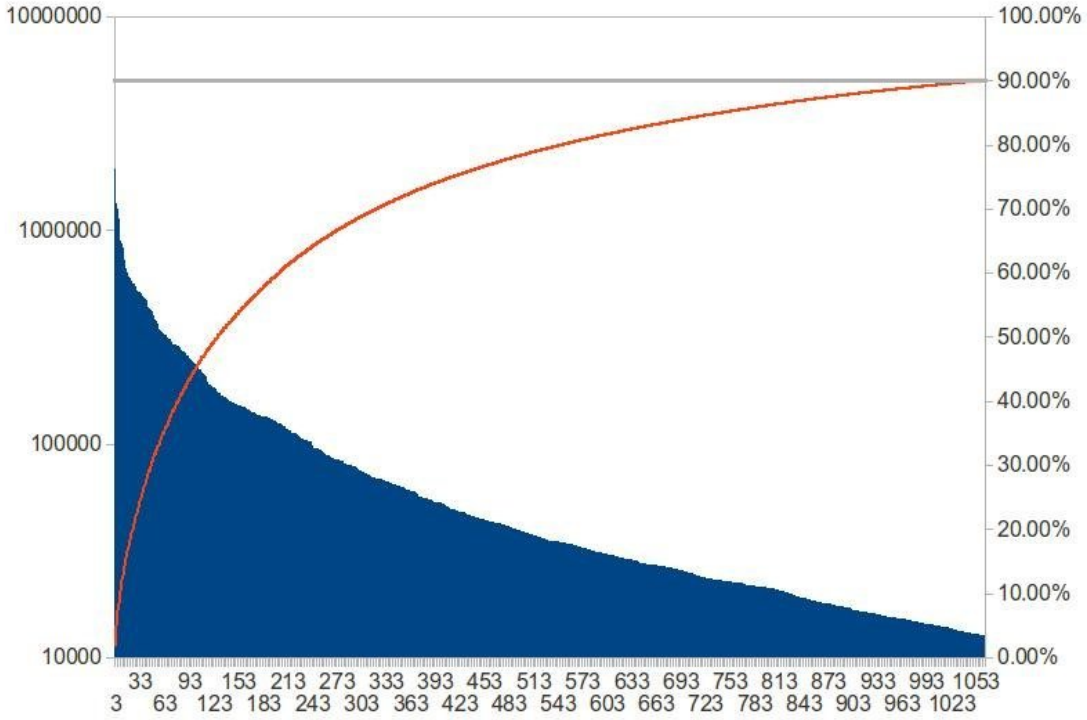

Figure 2.5: Coverage of API calls in the API semantics engine. API calls are represented by numbers on the x-axis. A vertical blue line represents the number of applications in which an API call occurs. The red line represents the cumulative distribution of API calls across call-sites. 1062 APIs make up for 90% of the calls in 95910 applications, and are part of those supported by our API semantics engine.

2.5 Pegasus

We design and implement Pegasus, an analysis system that combines the abstraction procedure in Section 2.4 with analysis of PEGs and rewriting of Android applications.

System Overview Figure 2.6 presents an overview of the Pegasus architecture. Pegasus takes as input an Android application and a specification expressed as safety property over events and actions. It uses a *translation tool* to convert Dalvik bytecode to Java bytecode. Using Java bytecode allows us to use off-the-shelf analysis frameworks.

The *abstraction engine* takes as input Java bytecode and generates an PEG as output. The PEG is fed to the *verification tool*, along with a specification to check for conformance. If certain application behavior cannot be analyzed (for instance, due to unresolved reflection), the *rewriting tool* generates a new application that contains dynamic checks when reflective calls are made.

If the PEG satisfies the specification, and the implementation of the API and event se-

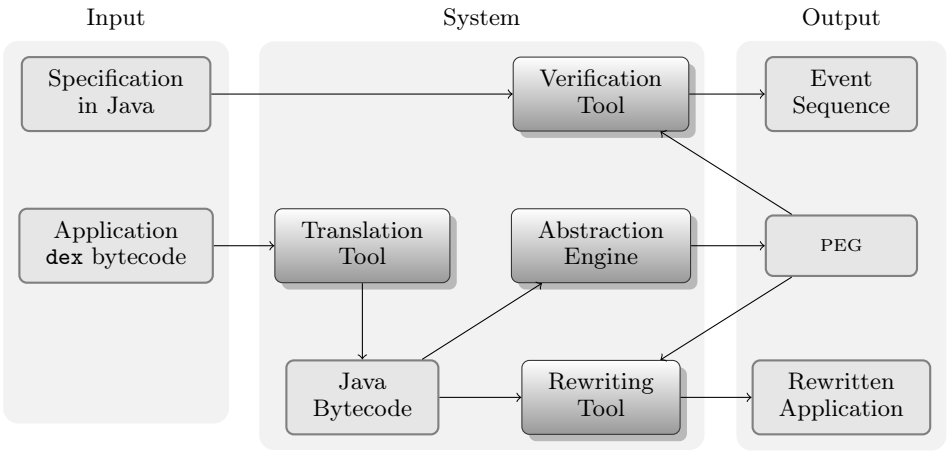

Figure 2.6: Pegasus architecture. The system consists of a translation and a verification tool, and an abstraction engine.

mantics engines, and the verification procedure is sound, the application is guaranteed to satisfy the specification as well. If the PEG does not satisfy the specification, it may be because the application violates the specification, or because the overapproximation creates false positives. For each violation, Pegasus produces a counterexample trace which can be used to determine if the violation corresponds to a feasible execution.

Specifications Recall that *safety properties* assert that certain undesirable behaviors never occur. Researchers have developed a numerous languages for safety properties. The SLAM project used a C-like specification language called SLIC [5], because it was convenient to use specification language with similar syntax to the analyzed programs. Similarly, we write specification monitors in Java.

A specification for the running example is shown in Figure 2.7. The callback function `checkEvent` is used to determine if the current event modality corresponds to the button click handler for REC. If the current event modality corresponds to the REC button click event, the specification checker sets class field `recButtonClicked` to `true`. It then scans all the behaviors associated with the current event modality. If it finds the `RecordStart` behavior and the record button has not been previously clicked, it signals a violation by returning `true`.

A user of our system implements specifications using the `SpecificationChecker` interface, which defines the callback function `checkEvent`. The verification algorithm calls this function for each event modality reached during exploration. Depending on the specification, the `checkEvent` function inspects the event type, the actions associated with the event, or both. The `checkEvent` returns `true` if a violation has occurred based on the current event modality, and `false` otherwise. The specification checker can maintain a specification state in its class fields. The specification state is stored and restored by the verification tool using Java

```
1  public class RunningExample implements SpecificationChecker {
2    // is the application currently recording?
3    private boolean isRecording = false;
4    // is the application allowed to record? (i.e., did the user
5    // press the Record button and not yet press the Stop button?)
6    private boolean recordingAllowed = false;
7
8    public boolean checkEvent(EventModality event) {
9      if (event.getEventHandler() ==
10         getClickHandlerForButtonByLabel("REC"))
11       recordingAllowed = true;
12     else if (event.getEventHandler() ==
13         getClickHandlerForButtonByLabel("STOP"))
14       recordingAllowed = false;
15
16     for (Action action : event.getActions()) {
17       if (action == RECORD_START)
18         isRecording = true;
19       else if (action == RECORD_STOP)
20         isRecording = false;
21     }
22
23     boolean violation = (isRecording && !recordingAllowed);
24     return violation;
25   }
26 }
```

Figure 2.7: Specification for the running example.

serialization.

To ease the task of writing specifications, we also implement a mapping from low-level API calls to high-level actions, such as maps from API calls to permissions [3, 31], and other security relevant actions, such as the start and the stop of recording. Pegasus enumerates the application's sequences of actions. It uses a Java interface to pass these sequences to the Java specification, which updates the state of the specification until a violation state is reached. If no sequence in a PEG leads to a violation, the application satisfies the specification.

Verification Pegasus includes a verification algorithm that uses a bounded, breadth-first graph search with pruning, to check security properties written as Java checkers. Once the PEG has been generated, specifications can also be checked using other model checkers.

Rewriting Our analysis is designed to successfully analyze many common-case uses of potentially problematic Java constructs such as reflection and dynamic invoke dispatching. The semantics of these constructs depends on information that is available only when the program executes, so static analyzes may be unable to precisely analyze programs that use

them. We rewrite applications to include runtime checks to account for cases where static analysis does not succeed.

When the abstraction engine fails to analyze part of an application, three strategies can be applied. The first one is to introduce a havoc statement and assume anything can happen. This strategy usually leads to a high false positive rate, and consequently, a tool that is not usable in practice.

The second strategy is to add runtime checks to only allow executions that respect the conditions computed by static analysis. For example, we can add runtime checks to only allow a reflective call if the target call was already derived by static analysis. Static analysis may also fail to determine all the sensitive resources an application accesses. We can similarly add runtime checks to only permit accesses to URIs that were either statically determined, or are not considered sensitive, such as contact lists or SMSes.

We use the second strategy. In specific cases, where we have manually scrutinised an application, we permit executions even if they have not been analyzed statically. This occurs when we believe the behavior that was not statically analyzed is benign. Such manually aided rewriting allows us to reduce the overhead of runtime checks.

In Section 2.6 we evaluate the number of unresolved transitions in the applications we analyzed. The number is generally small, a fact we attribute to the simple coding patterns used by most applications (e.g., using a constant string as the argument to a reflection call), as well as our per-event context-sensitive analysis which allows us to propagate information through method calls within each event handler.

We do not support unknown native code. Known native code is modelled by the API semantics engine. Known dynamic class loading is supported by analysing the class and treating its loading point as a reflective call.

Implementation Pegasus is implemented in 11,626 lines of Java code, including the code for the abstraction, model generation, and verification phases. The API semantics engine models 1218 APIs and the event semantics engine supports 62 different types of events. We developed a translation framework to translate Dalvik bytecode to Java bytecode; the dataflow analysis and rewriting are implemented in Soot [69], a compiler and static analysis framework for Java bytecode.

2.6 Evaluation

This section describes our experiments using Pegasus to demonstrate that PEGs can be used to automatically check and enforce policies in Android applications. All experiments are performed on an Intel Core i7 CPU machine with 4GB physical memory.

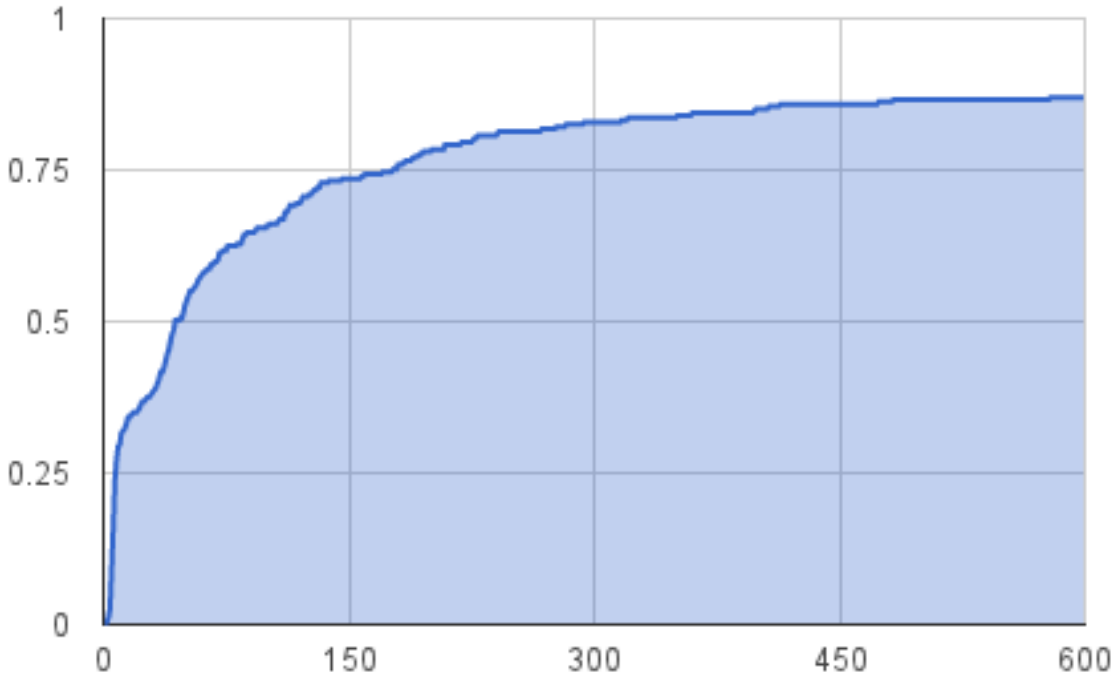

Figure 2.8: CDF of abstraction time (in seconds).

Generic Specification Checking

We run Pegasus on 152 malicious and 117 benign Android applications , and measure the execution and the size of the PEG. The malicious applications are from the Android Malware Genome Project[74], and the benign applications are randomly selected from the Google Play Store. Figure 2.8 presents the Cumulative Distribution Function (CDF) of the time spent on PEG generation. On over 80% of the applications, the abstraction phase terminates within 600 seconds. The abstraction phase also always terminated within 2 hours. Figure 2.9 presents the CDF of PEG verification time, on a logarithmic scale. The verification phase terminates within 1000 seconds, for over 80% of the inputs, and the verification phase always terminates within 3.6 hours. To boost efficiency, we heuristically bounded verification to terminate after 50000 states were explored. The justification behind this heuristic is shown in Figure 2.10, where most of the applications have at most 10000 unique states, so the probability of an unsound result is low.

In the verification phase, we check 6 application-independent properties to determine if sensitive operations are guarded by user interaction. The three sensitive operations we consider are reading the GPS location, accessing the SD card, and sending SMSes. The properties we check are that the three behaviors above are always bracketed by user interaction, such as a button click. The result of verification is shown in Table 2.2. We see that malicious applications performing sensitive operations without user consent more frequently than benign

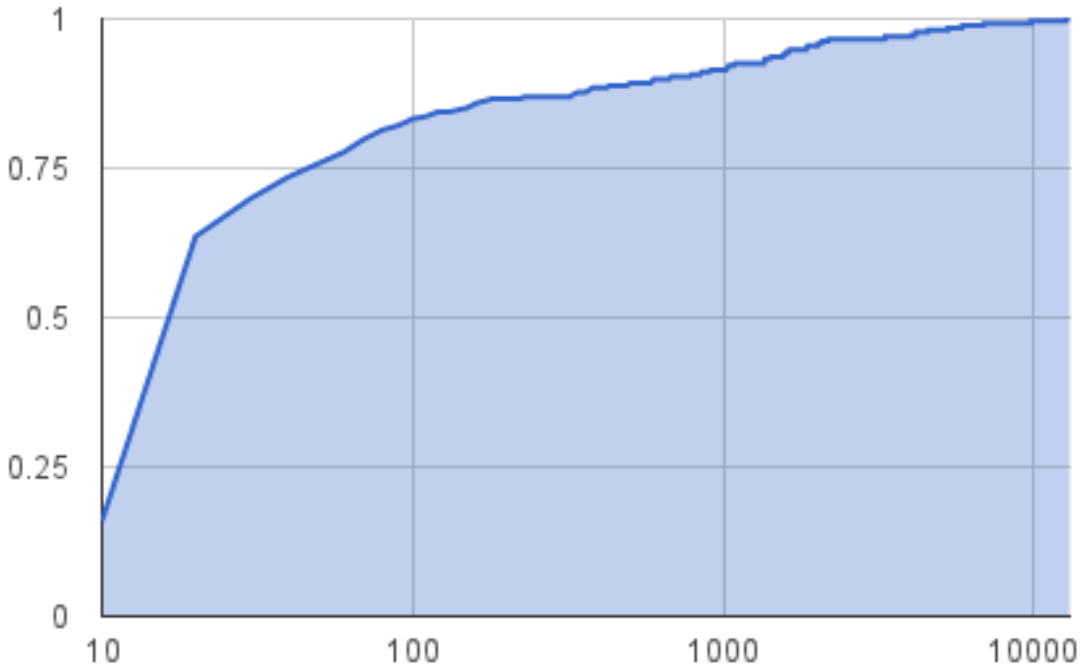

Figure 2.9: CDF of the verification time (in seconds).

Sensitive Operation	Malicious NoUI	Malicious Total	Benign NoUI	Benign Total
GPS	15	15	18	30
SD card	25	26	25	32
SMS	10	11	0	1

Table 2.2: Results of checking application-independent specifications. The first column is the name of the sensitive resource accessed. The "NoUI" columns list the number of applications accessing these resources without user consent.

applications.

Application-Specific Properties

Sample Applications In this section, we check application-specific properties on 17 sample applications, including 8 benign applications and 9 applications with known malicious behaviors, to demonstrate Pegasus used as a diagnostic tool. Table A.1 in the appendix lists these sample applications and presents a short description for each application. The first 8 applications are benign samples selected from the official Android Market and third-party application stores. We selected these applications to represent a broad variety of different

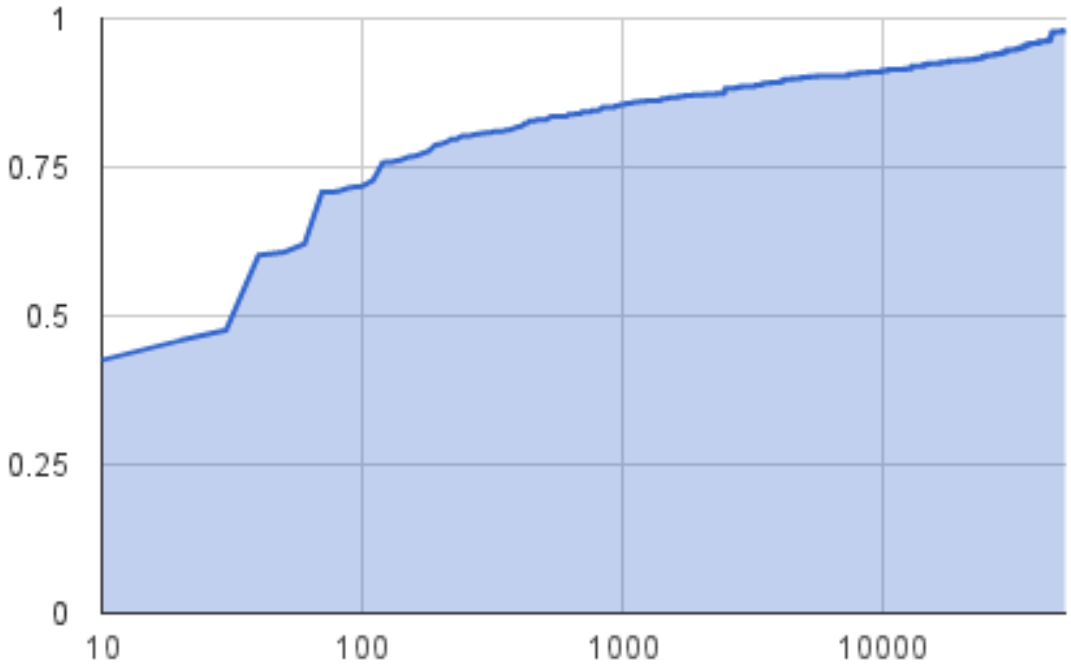

Figure 2.10: CDF of PEG size measured by the number of states.

application classes that together exercise most of the core functionality supported in the Android system. These applications implement a variety of behaviors such as recording audio, accessing the phone's contacts, sending SMS messages, and accessing the device GPS location. The remaining 9 samples are malware which exhibit a variety of malicious behaviors.

Specifications For these applications, we constructed application-specific properties after installing each application, reading its documentation to understand its intended functionality, looking at the list of events and GUI widgets used by the application, then determining a security policy which an analyst might reasonably wish to impose on the application. We wrote 23 application-specific properties.

PEG Generation Table 2.3 summarizes the results of PEG generation. The last two columns of Table 2.3, show how often the analysis can resolve the targets of intent calls and reflective calls. We also manually inspected the decompiled source code to resolve values that were not automatically determined, then used the rewriting tool to enforce those values at runtime.

Verification We used Pegasus to check the generated PEGs for conformance to their properties. When Pegasus found a violation, we manually executed the applications and used

Name	Size	EM	Intents		Reflection	
	KB	#	UR	T	UR	T
Who's Calling?	148	83	0	2	0	0
Share Contacts	135	359	0	9	1	2
Geotag	117	226	0	19	1	2
Find My Phone	285	29	0	2	6	11
Simple Recorder	20	16	0	0	0	0
Diet SD Card	304	155	0	14	2	5
SMS Cleaner Free	159	175	0	14	0	3
SyncMyPix	425	300	4	12	1	3
SMS Replicator	63	18	0	0	0	0
ADSms	41	38	0	0	0	0
ZitMo	20	19	0	0	0	0
HippoSMS	404	434	0	38	0	0
DroidDream	204	89	12	15	0	0
Zsone	241	30	0	0	1	3
Geinimi	558	51	4	4	4	6
Spitmo	20	6	0	0	0	0
Malicious Recorder	20	25	0	0	0	0

Table 2.3: Summary of the evaluation results. The columns are: (1) name of application, (2) size of the .apk file, (3) number of event-modalities, (4) number of (unresolved) intents, (5) number of (unresolved) reflection calls.

the counterexample trace to determine if the violation represented a feasible behavior of the application. If source code was available, we also inspected the code.

Our results show that Pegasus completes verification of most applications in less than a second, with a maximum verification time of 10 seconds. The length of the counterexample traces for violations ranges from 4 to 10 events.

The properties we checked are discussed in detail in Section 2.6. We checked 8 benign applications, against a total of 16 properties. For 11 of these properties, Pegasus determined that the application satisfied the property. For 3 of the remaining properties, we determined that Pegasus found a property violation due to legitimate but unexpected application behavior. We believe that analysts would find such information valuable.

For 1 of the remaining 2 properties, Pegasus determined that an infeasible path involving dead code violated the property. For the last property, imprecision in the analysis caused Pegasus to determine that the application violated the property. Consequently, we conclude that Pegasus has false positives for 2 of the 16 properties.

For all 9 malicious applications, Pegasus correctly reported violations of at least one of

the 7 properties. In other words, there were no false negatives for these properties.

Case Studies

We now present case studies illustrating properties one can check with Pegasus. Table A.2 in Appendix A provides detailed information about the security actions and events used in the specifications in this section.

Specification Format For brevity, we present specifications as LTL formula. Note that Pegasus does not actually take LTL formula as input but requires specifications to be encoded as a Java checker. A specification for the Find My Phone application is shown below.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Receive-SMS}$$

The symbol **U** is the *until operator*, while Send-SMS is an *action* and Receive-SMS is an *event*. The specification is read in English as asserting that

> The application does not send an SMS until it receives an SMS.

A important technical clarification is in order: the temporal logics supported by standard model checkers are usually *state-based*, meaning the propositions occurring in formula describe properties of states. In our specification, mode-variables describe states and action labels describe properties of transitions. In technical temporal logic parlance, our specifications are both *state and event-based*. Consult [20] for an in-depth discussion of such issues.

Simple Recorder The simple recorder contains two buttons to start and stop recording, and behaves as expected. We check that the application only records audio when the REC is clicked, and stops when STOP is clicked. This property is expressed in LTL as

$$(\neg\text{Start-Recording}\ \mathbf{U}\ \text{REC.onClick}) \wedge (\text{Stop-Recording} \iff \text{STOP.onClick})$$

The application satisfies this property.

Diet SD Card We check that the application accesses the SD card only after a button labelled Clean is clicked.

$$\neg\text{Access-SD}\ \mathbf{U}\ \text{Clean.onClick}$$

The application satisfies this property, showing that the SD card is only accessed after Clean is clicked.

Geotag We check that the application accesses geolocation information only after the user clicks the Locate button.

$$\neg\text{Access-GPS}\ \mathbf{U}\ \text{Locate.onClick}$$

Pegasus discovers a property violation. On examining the counterexample, we determined that the main activity's `onCreate` event handler initialises the Google Ads library, which spawns a new background task and accesses the geolocation information. If we refine our property to

$$\neg\text{Access-GPS}\ \mathbf{U}\ \begin{pmatrix} \text{Locate.onClick} \\ \vee \quad \text{AdTask.onPreExecute} \end{pmatrix}$$

Pegasus no longer reports a violation. We have also learnt that the only way the geolocation can be accessed without the user's consent is via the Google Ads library.

Who's Calling? Similar to the Geotag application, we check the that contacts information is only accessed after a phone call is received.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{PhoneCall.onReceive}$$

Pegasus returns a counterexample which shows that the application retrieves and caches the contact list when it starts up. We verified this behavior manually by running the application in an emulator and observing its API calls.

Share Contacts We check whether the application accesses contacts if an SMS is not sent.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Access-Contacts}$$

This property holds. We then checked that SMSes are sent in response to user input.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Send.onClick}$$

This property also holds. Finally, we check if adding a new contact requires user consent.

$$\neg\text{Insert-Contacts}\ \mathbf{U}\ \text{Insert.onClick}$$

This property too is satisfied.

SMS Cleaner Free This application allows users to delete SMS messages that match a user-provided contact name. We check if accessing contacts is user-driven.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{Select-Contact.onClick}$$

Pegasus reports a property violation. We manually investigated the counterexample generated and concluded that the counterexample was not feasible. The reported violation is a false positive. This application uses a switch statement to register the same event handler for different button clicks. Our abstraction engine does not consider branch conditions, so in the generated PEG every button click can trigger every event handler.

Find My Phone This application responds to the receipt of an SMS containing a specific keyword by sending the phone's GPS location. We check if an SMS can be sent without any being received.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Receive-SMS}$$

Pegasus reports a violation. We manually verified that the violation was a false positive.

SyncMyPix The SyncMyPix application specifies the target of an intent by looking at the configuration file and using a default value if the user does not specify the target. The set of possible runtime targets is not arbitrary, because they must be components in the application, but static analysis does not have this information and is inconclusive. We rewrite the application to force the target of the intent to be the default one. The rewriting takes 5 seconds and the rewritten application works correctly. We check if the rewritten application can access contacts without the Sync being clicked.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{Sync.onClick}$$

Pegasus reports a property violation. The counterexample revealed that the application may access the contacts if the user clicks the Result button. We verified manually that clicking Result displays the results of synchronising pictures. This behavior is innocuous, so we refined the property as below.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \begin{pmatrix} & \text{Result.onClick} \\ \vee & \text{Sync.onClick} \end{pmatrix}$$

The application satisfies the refined property.

Malicious Recorder This application contains the same functionality as the benign recording application. However, it also records audio for 15 seconds whenever a new SMS is received. We check the same specifications as the Simple Recorder application and verification fails. The counterexample shows that that a timer can trigger recording.

ZitMo (Malware) In most benign applications, the SMS messages received should either be passed on to the next Broadcast Receiver or displayed to the user. Thus we check

$$\neg\text{BroadcastAbort}$$

to see whether the ZitMo application discards SMSes without notifying the user. Pegasus reports a violation, which we confirmed manually.

SMS Replicator Secret (Malware) We checked two properties of this application.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Button.onClick}$$

$$\neg\text{BroadcastAbort}$$

Both properties are violated, because the application malicious sample sends SMS messages without user interaction, and deletes certain incoming SMS messages.

ADSms We use Pegasus to study this application and understand how it uses permissions. We check three properties.

$$\neg\text{Kill-Background-Processes}$$
$$\neg\text{Read-IMEI}$$
$$\neg\text{Send-SMS}$$

The counterexamples show that this application registers a broadcast receiver to kill anti-virus processes. We also discovered that the broadcast receiver starts a new process, which reads IMEI information and sends SMS messages to premium-rate numbers.

2.7 Related Work and Discussion

We build upon work at the intersection of specification languages, program analysis, model checking, and Android security. These areas are all mature and a comprehensive survey is beyond the scope of this chapter. We only attempt to place our work in the context of either seminal or very recent chapters in each area.

Specification Languages A specification language may be external to a program, as with a temporal logic or internal to a program as in design-by-contract mechanisms. See [70] for a hardware-oriented survey of industrial formats for temporal logics, and [12] for an overview of JML and ESC/Java2, which is well known, but only one of many specification mechanisms for Java.

Our use of rewriting achieves a form of in-line monitoring, an idea articulated in [28]. Monitoring for security policies has been implemented in EFSA [59] and PSLang [27], and with a focus on mobile security in the S3MS project [21], The Apex [53] system uses Android permissions to guide runtime monitoring, while our monitoring policies are defined by custom security properties.

We use runtime monitoring to deal with cases in which static analysis is ineffective, such as in the presence of dynamic class loading or running native code. This combination can also be viewed as an optimisation that reduces the overhead of runtime checks, and leverages the strengths of both. JAM, developed concurrent to our work, combines model checking with abstraction-refinement to alleviate monitoring overheads [34]. Techniques from JAM can be used to improve our rewriting tool.

Static Analysis The design of our abstraction engine combines ideas from abstract interpretation [17] and software model checking. The closest related work is the SLAM toolkit [7], for checking properties of device drivers. Similar to SLAM, we check policies about the interaction of an application and the operating system, and use cubes over Boolean variables for symbolically encoding. Unlike SLAM, we abstract the operating system context of an event.

Moreover, instead of a theorem prover, we use a domain specific event- and API-semantics engines to determine how mode variables are transformed.

The ideas in SLAM has been extended to *lazy abstraction* [41], which interleaves the abstraction and checking process, and to YOGI [10], which combines testing and theorem proving to construct abstractions. Most developments that follow SLAM, such as those surveyed in [46] focus on improving either model checking or abstraction. Our work is orthogonal because we compute a different type of abstraction. Much work successive to SLAM, can be lifted to Android and used to improve the construction or verification of PEGs.

Though program analysis is a mature field, event driven programs have only recently received attention. Interprocedural data-flow analysis with a finite-height, powerset lattice is EXPSPACE-hard [45]. In a language like Java, all analysis depends on the quality of points-to analysis. Points-to-analysis in the presence of an event-queue faces similar complications as with function pointers [22]. These are obstacles that will have to overcome if we wish to improve our analysis.

Android Security TaintDroid [26] supports dynamic taint-tracking for Android applications. It explores one execution at a time, while our system checks all the possible behaviors of an application. Security analysis based on control flow patterns and simple static analysis has been used in [25] to detect a range of malicious behavior. A semantically richer static analysis has been applied to Android in [55], but the focus is on common bugs rather than security properties.

The Stowaway system [31] combines static analysis with a permission map to identify applications requesting more permissions than they use. Permission-based approaches [3, 24, 53] use a map from API calls to Android permissions. Pegasus uses use a map from API calls and arguments to a custom-defined set of actions. This set of actions extends the abstraction of APIs provided by permissions.

Access control gadgets[58] is a secure UI element to capture the user's permission-granting intent at runtime. It tries to solve the problem of enabling cooperative application developers to write applications that can be easily analyzed, while we try to solve the problem of detecting a new category of malicious behavior from legacy and potentially non-cooperative applications.

Discussion Analysis of PEGs provides richer semantic information than is available in standard program representations. PEGs abstract the operating system context in which event handlers execute, and model checking of PEGs provides more information than pattern matching on syntactic program artefacts. We can detect permissions used in background tasks, or in event handlers triggered by invisible buttons. We are not aware of other techniques that can detect such behavior.

Analysis of PEGs is not a panacea for malware detection. However, our analysis gives security analysts an advantage in their arms race against new malware, by aiding in identifying a new class of malicious behavior. While attackers can work to evade our analysis

mechanisms, e.g., using native code or dynamic class loading, such evasion requires code to adopt a more convoluted structure or exhibit more circuitous behavior, compared to benign applications — thereby making the code more conspicuous. Thus, much as the ZOZZLE defense against heap spraying attacks in Javascript [19], our analysis can support and facilitate the identification of malware. Even simple pattern matching of syntactic structure, or triggered runtime checks, may be sufficient to reveal anomalies that indicate malicious intent.

Another type of attacks to evade our detection system is to leverage the lack of details or information flow due to our approximation. For example, to evade our general policy enforcement in SMS Replicator Secret, the malware can wait util user clicks on anything and then send SMS, or lure the user into clicking on any button, or mount clickjacking attacks[42].

Chapter 3

WebSyn: Iterative Security Analysis of Web Protocols using Synthesized Models

3.1 Introduction

Web applications are a critical component of the Internet ecosystem, and provide diverse functionality such as social networking, online shopping, and storage and retrieval of data in the cloud. A unique characteristic of such applications is integration with third-party services to implement critical functions. Such *mashups* typically access diverse third party services over web (or HTTP) protocols. For example, a modern application may rely on services such as Facebook Connect to authenticate users, Disqus for comments, Dropbox for storing user data, and may rely on Paypal for monetary transactions. The web application integrates each mechanism via protocol defined on top of HTTP . Due to this integration, a vulnerability in even one of these mechanisms can have critical consequences for the security and privacy of users, other web applications, and online services. Unfortunately, implementing such mashups and the underlying protocols is challenging: subtleties of the web's security model often result in inadvertent vulnerabilities. For example, researchers identified vulnerabilities in web-based SSO protocols in 2010 [48, 2], 2011 [71], 2012 [66, 67], 2013 [72] and 2014 [75].

Existing tools approach the vulnerability discovery problem from two major perspectives. One perspective is that of a top-down approach based on applying model checking and proof systems [2, 72, 8, 9] on *manually* constructed models (specifications) of web applications. However, web applications often lack formal specifications. Manually writing specifications in a formal language requires significant effort, and remains quite challenging and error prone. Developers or security analysts wishing to write the specification or the model would need to understand the intricacies of the formal specification language as well as translate complex modern applications into the given formal language. These challenges could lead to the developed model itself being erroneous and inconsistent with the implementation. Even worse, if the user wishes to analyze another application, the whole manual process has to be repeated.

The second perspective is that of bottom-up approaches that use static analysis or symbolic execution techniques to automatically build system models [1, 6, 15] from full system implementations. However, web applications operate in a distributed setting, and the developer or the security analyst may only have *partial visibility* into the full protocol. The implementation of a web protocol is typically only partially visible because the code (or binaries) for some parties in the protocol will not be available. For example, a typical mashup developer lacks access to Facebook or Paypal code. In addition, such approaches are unable to efficiently recognize high-level protocol related semantics in the implementation.

Our Approach In this chapter, we propose a new approach for finding security vulnerabilities that is applicable under the unique constraints posed by web applications. Our approach provides a middle ground between the two perspectives of manually specifying models and fully-automated model inference. Instead of manually writing models for each application, we provide a set of basic building blocks (in the form of a *domain specific language*) that are common in web protocol models and can be assembled to form different web protocol logics; and instead of deriving the protocol models from the implementation, we inductively *synthesize* the models from examples of system execution *traces*.

Providing examples of system execution traces presents a natural interface for authoring application specifications and analyzing security; humans already generalize from examples. We leverage recent advances in the field of inductive program synthesis [37] to translate user provided examples into candidate models. The synthesis techniques make use of the domain specific language (DSL) discussed above to infer a candidate model corresponding to the execution trace. Finally, we couple the inferred model with state-of-art formal verification tools (namely ALLOY [43]) to find security vulnerabilities. Our approach is amenable to interactive analysis; developers or security analysts can refine the analysis by providing additional example traces or guiding the formal verification tool.

Our approach shares similar motivation with previous work on inductive specification generation for vulnerability discovery [4, 56, 29], and provides a general framework thanks to synthesis techniques (See Section 3.10). We note that while synthesis from examples has been explored in some other domains [37], to the best of our knowledge, we are the first to explore the benefits of program synthesis and domain languages for enhancing application security.

WEBSYN To demonstrate the concrete benefits of our approach, we present WEBSYN, a system that aims to find vulnerabilities in web applications. Web developers or security analysts can provide execution traces of web applications as an input to WEBSYN. This satisfies the unique constraints of web applications; execution traces are accessible to such users even under the constraints of partial code visibility and lack of application specification.

Our main insight is that we can use program synthesis techniques, which have recently been successful in a number of domains [63, 62, 36, 68], to guide the inference of web application models from example execution traces.

The design of WebSyn consists of three main components. First, we define a domain specific language (DSL) for web protocols, in order to control and customize the search space in vulnerability discovery. Second, we introduce a new algorithm for synthesizing code describing a protocol from execution traces and analyst feedback.WebSyn expresses the protocol as a program in a declarative DSL, which the analyst can examine and provide feedback to the tool. Third, we integrate formal verification tools such as Alloy with the inferred model to construct an end-to-end system for vulnerability detection using execution traces.

WebSyn accommodates interactive system analysis; the analyst can interact with the system via additional example execution traces, and provide feedback on the vulnerabilities and the issues WebSyn identifies. We note that our end-to-end system provides new opportunities for interaction with formal verification tools: first we are able to enable model refinement via additional execution traces, and second, we are able to translate the output of the formal verification tool in terms of execution traces using the domain specific language.

Experimental Results As a proof of concept, we implemented our WebSyn system, and evaluated it with four real world web applications. In three of them, WebSyn was able to find various CSRF vulnerabilities, demonstrating the utility of our approach. WebSyn was able to discover both previously-known vulnerabilities (previously found using manual analysis), as well as new vulnerabilities. These results demonstrate that by defining a single DSL for web protocols, and by performing synthesis using the DSL and execution traces, we can make the process of vulnerability discovery easier to implement across a number of web applications.

Contribution Our work makes the following contributions.

1. A general approach for finding security vulnerabilities using system execution traces, domain specific languages, and synthesis techniques.
2. A DSL to represent the set of typical semantics of web applications, and a new algorithm for the synthesis of web protocol models (in terms of the DSL) from system executions and analyst feedback. Our algorithm enables modeling systems in which code may only be partially available.
3. An application of our system to synthesize models of web protocols that involve HTTP requests and responses among the client and multiple websites. Our proof-of-concept case studies demonstrate how our system can efficiently detect real-world, CSRF vulnerabilities, including previously unknown ones.

We review the relevant background in Section 3.2. Section 3.3 presents an overview of our approach. In Section 3.4, we present the domain specific language we use to represent the synthesized protocol. In Section 3.5, we present the algorithm of the protocol synthesis. Section 3.6 introduces the verification process and the user feedback. We describe our implementation in Section 3.7 and evaluate it in Section 3.8. Section 3.9 presents a discussion of limitations and future directions.

```
GET /login HTTP/1.1
Host: bodgeitstore.com

HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=7ffa4512

<form method="post" action="/login">
<input type="hidden" name="csrftoken" value="3eff8527">
<input type="text" name="username">
<input type="password" name="password">
<input type="submit" name="submit" value="login">
</form>

POST /login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: session=7ffa4512
Host: bodgeitstore.com

csrftoken=3eff8527&username=user1&password=secretpwd&submit=login

HTTP/1.1 200 OK
Content-Type: text/html

<b>Welcome!</b>
```

Listing 3.1: Example traces from the Bodgeit Store

3.2 Background

Web Security

Running Example We use the Bodgeit Store application[57] as our running example. The Bodgeit Store is a deliberately vulnerable web application used for teaching web security. It is an online shopping web application that allows users to register an account, login to an account, manage shopping carts and make purchases. Listing 3.1 is an example of some messages from its protocol.

Web Applications A *web application* is a distributed system based on the HTTP protocol. We refer to the participants of a web application as *endpoints*. A *client endpoint* (client for short) is a browser and a *server endpoint* (server for short) is a web principal represented by its web origin. A *message* is either an HTTP request or an HTTP response. A *web protocol* is

a specification of the sequences of messages exchanged between endpoints and the invariants satisfied by these messages.

A *session* is a set of messages pertaining to a single online activity. HTTP is a stateless protocol. A typical approach to building a stateful, session-aware web application around HTTP is to include session identifiers in every HTTP request. In our running example, the `session` field in the cookie identifies the shopping activity session. In addition, we also consider that the `username` and `password` fields identify the overall user session, and the `csrftoken` identifies the login activity session.

Server Model We assume a simple model of the server: it receives an HTTP request and extracts information from the request. Based on the extracted information and the history of requests and responses, the server constructs a response and sends it to the client.

Threat Model We consider the web attacker model and session integrity formally defined in Akhawe et al. [2]:

A *web attacker* is a malicious principle who controls a web server visited by the user. Intuitively, the web attacker can be thinked as having "root access" to this web server, and is able to retrieve arbitrary information contained in the request, and send arbitrary response to the user. As a result of interpreting the response in the user's browser, the web attacker also has access to the browser's web APIs exposed to common websites. In addition, the web attacker can send arbitrary HTTP requests to the honest servers. However, the web attacker has no special network privileges. This means the web attacker can only respond to HTTP messages directed at the web attacker's own servers.

A *session integrity* condition states that the attacker should not be able to cause benign servers to undertake potentially sensitive actions. Cross-Site Request Forgery (CSRF) attacks are typical violations of the session integrity policy. A CSRF is a type of attack by a web attacker that violates the session integrity of a web application in which the user is authenticated. For example, a login CSRF attack violates the login session by directly logging in the user without the initial login page. If the Bodgeit Store code did not check the `csrftoken` (Listing 3.1) in the second request, a malicious website could just submit a request (via the user's browser) to the store using the attacker's password and log the user in as the attacker to Bodgeit.

Our tool searches for a large variety of CSRF attacks. These CSRF attacks can take place in different forms during different steps of a web protocol. For example, the aforementioned CSRF attack happens at the user credential submission step, and the client is authenticated as the attacker. There are also CSRF attacks that construct other authorization tokens and perform the action on behalf of the user, which we will show in our case studies in Section 3.8. A key feature in our approach is the ability to accommodate the customized, non-standard application logic and generate the correct CSRF attack traces tailored to it.

Program Synthesis

In this section, we give a background overview of synthesis techniques which provides a basis for developing systems for specification synthesis. See Section 3.10 for a discussion on related work, and Gulwani at al. [37] for a comprehensive overview of such techniques. Broadly speaking, a *specification synthesis* framework takes examples of behavior as input and searches the space of possible specifications, defined by a *domain specific language (*DSL*)*, for candidates that conform to the given input. We discuss all these components further below.

Domain Specific Languages A language defines a search space for candidate programs; thus, the choice of a target language has implications on the size of the search space as well as the space of problems that the synthesizer can solve. For example, a general-purpose programming language as the target language has the appeal of being able to solve a wide range of problems. On the other hand, these languages also have an infinite search space of possible programs. A narrow domain specific language defines a smaller space of program, but may not be sufficient for the problem.

Synthesizer A synthesizer searches the space of possible programs for candidates that satisfy the provided constraints. The analyst inputs, via execution examples, form the constraints for the target program. In any synthesis application, given the computational complexity of performing a naive search, the primary concern is often designing a clever search strategy.

Domain knowledge is critical to help guide the search of candidate programs as well as rank candidate programs. Typically, the search space of possible specifications remains prohibitively large even with a DSL and synthesis systems require smart search algorithms as well as liberal use of domain knowledge. It is useful to compare the synthesis search to what typically occurs when humans author specifications. A specification author would look at examples of behavior or think of intended behavior and combine it with her domain knowledge gained with experience, finally creating an appropriate specification—an effort prone to errors. A synthesizer replaces this step with automated techniques.

Formal Verification

A formal verification tool takes as input an abstract *model* of the system, and a *property* that the analyst want the system to maintain. The tool checks if the system model satisfies the property. The result of the verification is either `YES` meaning that the model satisfies the property, or `NO` which means otherwise. Optionally, a *counter-example* is returned with the `NO` answer. If the counter-example represents an actual behavior of the system, it means the implementation of the system fails to satisfy the property. If the counter-example is spurious, it can be used to *refine* the model so that it better reflects the implementation of the system.

Researchers have created formal verification tools for proving the security of software and protocols [6, 15]. Researchers have also leveraged formal tools to analyze deployed systems, resulting in the discovery of real, exploitable vulnerabilities [4, 8, 71].

In our implementation, we use ALLOY [43] as the basis of our verification component. The word ALLOY refers to both a model description language based on the notion of relations, and a model checker that performs verification on ALLOY models. Our DSL is compiled to ALLOY for verification, and the counter-examples are translated back to traces of HTTP messages.

3.3 System Overview

To solve the problem of synthesizing specifications for vulnerability discovery, we must solve two types of problems. The *representation problem* is to design a DSL that captures the subtleties of web protocols while still remaining high-level enough for efficient checking. The *algorithmic problems* are to construct a protocol model, find vulnerabilities, and update the model using analyst feedback. The details of the algorithm depend on the representation, and hence preclude an off-the-shelf solution. We now describe how these problems are solved in WEBSYN.

Designing a Representation

We design a language, called MDL, with primitives chosen to enable succinct descriptions of web protocols. These primitives have a precise semantics and compile into the language of a formal analysis tool. Section 3.4 explains MDL along with a description of our running example, the Bodgeit store, in MDL (Figure 3.2).

Algorithmic Components

Program Synthesis The first algorithmic problem is to construct a protocol model from HTTP traces, e.g., the one in Listing 3.1. Since our model is expressed as a program, model construction can be viewed as a *program synthesis* task. Specifically, the set of all MDL programs defines a search space and synthesis from examples seeks to find programs that generate and generalize those sample executions. Section 3.5 presents our synthesis algorithm and the data structures we use to achieve the performance necessary in an interactive system.

Vulnerability Discovery The second algorithmic problem is to discover vulnerabilities in the protocol model. Our system includes template descriptions of web protocol vulnerabilities (such as CSRF). We compile a MDL program together with a vulnerability description into an ALLOY model, and reduce the vulnerability discovery problem to a model checking problem.

The ALLOY model simulates all endpoints of a protocol and includes a malicious server and malicious client. The ALLOY model contains definitions of all relevant endpoints (HTTP

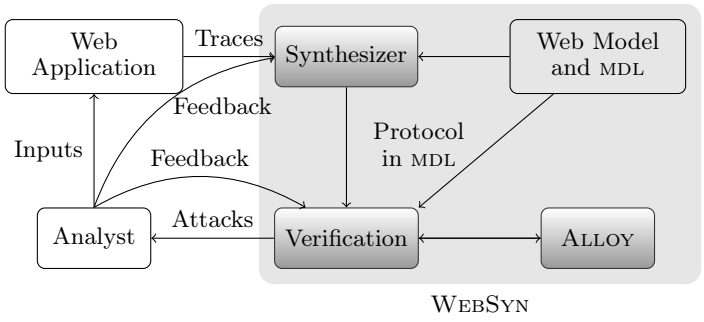

Figure 3.1: System Architecture: WEBSYN consists of a synthesizer and a verification component. It takes web application execution traces as input, and uses the synthesizer to generate protocol models. The verification component compiles protocol model and vulnerability templates into ALLOY models for vulnerability discovery, and ALLOY output to example attack traces. The analyst interacts with WEBSYN by providing inputs and refining the protocol model and the vulnerability templates.

clients, HTTP servers), messages (HTTP requests, HTTP responses), cookies, tokens etc., and the definitions of vulnerabilities such as token forgery. We describe malicious clients, malicious servers and benign clients in ALLOY using a general model of browsers and web servers. See Section 3.6 for details of this translation.

Feedback and Refinement The third algorithmic problem is to incorporate analyst feedback to update either the protocol model or the vulnerability description used by the system. We present the types of feedback in Section 3.6 and demonstrate their effects on the search space in Section 3.8.

Architecture and Exploration

The architecture of WEBSYN is shown in context in Figure 3.1. WEBSYN consists of two components. The synthesizer takes as input traces and generates an MDL program, which describes the protocol's behavior. The verifier takes as input the the MDL program, translates it to ALLOY model, run the ALLOY analyzer, and translates the counter-example back to the MDL level if the analyzer returns one. The analyst generates inputs to run the web application and feedback to feed directly to WEBSYN.

3.4 The MDL Language

In this section, we present the language for describing the protocol model: the Model Description Language (MDL). We start by using the Bodgeit Store example to informally describe

the semantics of MDL, followed by a formal definition of the key aspect of the language: the invariants. The full MDL syntax is defined in Figure B.1 in the appendix.

From a protocol execution's point of view, a MDL program describes the protocol logic of the servers in a multiparty web application, i.e., a MDL program takes a concrete HTTP request, and constructs a concrete HTTP response. See Figure 3.2 for the protocol model of the Bodgeit Store login protocol in MDL.

```
1 servers: bodgeit;
2 init:
3     bodgeit knows t1,t2;
4     client knows t3,t4;
5 messages:
6     helo(server=bodgeit, type=request),
7     rehelo(server=bodgeit, type=response,
8         fields=(jsid in setcookie, csrf in content)),
9     login(server=bodgeit, type=request,
10        fields=(rcsrf in urlparam, rjsid in cookie,
11                username in urlparam, password in urlparam)),
12    relogin(server=bodgeit, type=response);
13 invariants:
14    rehelo.jsid isa t1;
15    rehelo.csrf isa t2;
16    login.username isa t3;
17    login.password isa t4;
18    forall m1:rehelo, m2:login {
19       m1.jsid == m2.rjsid <=> m1.csrf == m2.rcsrf;
20    }
```

Figure 3.2: The MDL protocol model for the Bodgeit Store. Note that we have changed the variable names from the randomly generated original ones to more meaningful ones for the ease of understanding.

A MDL program consists of three top-level expressions: the endpoint declaration (`servers` in Figure B.1), the data type declaration (`init`), the message declaration (`messages`) and the invariant declaration (`invariants`). The endpoint declaration lists the participating server endpoints. In the Bodgeit Store example, there is only one server endpoint (`bodgeit`) as shown in Line 1 of Figure 3.2. The data type declaration lists all the data types and for each data type the endpoints who know all the data with that type at the beginning of the protocol. Our running example involves 4 data types `t1-t4`, as shown in Line 3,4 of Figure 3.2. The first two are initially known by the server (i.e., `bodgeit`) and the other two are initially know by the client. The message declaration defines all the messsages involved in the protocol and the data fields carried by the message. A message is either an

HTTP request or an HTTP response. The invariant declaration defines how each endpoint constructs responses according to the received requests. The message declaration and the invariant declaration are the core of the model. We explain both in more details in the following paragraphs.

The message declaration consists of a list of messages, where each message specifies which server it is sent from or to, whether it is a request or a response, and a list of data fields as well as where the data is located in the message. We are interested in the data location because it determines how the data get handled by the clients, e.g., whether it is forwarded in the subsequent requests (i.e. cookie data), or whether it is forwarded to a different web principle (i.e. data in the URL of a HTTP redirection).

The invariant declaration consists of a list of invariants over the messages and data fields. Current we support two kinds of invariants: the data type of a data field, and a boolean expression over multiple data fields quantified by message types. These invariants help to determine how the servers construct responses for requests. Intuitively, when a new request arrives, the server will match it with a request defined in the message declaration. Once the corresponding request definition is found, the server construct the response by finding a concrete solution to the invariants.

For example, Lines 6-8 in Figure 3.2 specifies that the `bodgeit` server sends a `rehelo` response when it receives a `helo` request. Line 14 in Figure 3.2 specifies that the `rehelo` response's `jsid` field is of type `t1`, and Lines 18-20 specifies that if two messages' session IDs match(`rehelo.jsid == login.rjsid`), their CSRF tokens should be equal, too, and vice versa.

Figure 3.3 lists the full syntax of the invariant declaration.

INVARIANTS :=	**invariants**: INVDEF INVDEF ...
INVDEF :=	ISA | FORALL
ISA :=	mt1.f1 **isa** t1;
FORALL :=	**forall** m1:mt1, m2:mt2 ... { BOOLEXP }
BOOLEXP :=	m1.f1 == m2.f3 | BOOLEXP BOP BOOLEXP
BOP :=	$\wedge$ | <=>

Figure 3.3: The syntax of the invariant declaration.

3.5 Synthesizing MDL

In this section, we present the algorithm for synthesizing a MDL model. The algorithm consists of three phases: trace alignment, data propagation and model generation.

Trace Alignment

The trace alignment phase takes multiple network HTTP traces in the form of

$$T = [(s_1^1, s_1^2, ...), (s_2^1, s_2^2, ...), (s_3^1, s_3^2, ...), ..., (s_n^1, s_n^2, ...)] \tag{3.1}$$

and produces an alignment between these traces in the form of

$$M = (\{(s_1^1, s_2^1, ..., s_n^1), (s_1^2, s_2^4, ..., s_n^2), (s_1^4, s_2^3, ..., s_n^3), ...\}, <)$$

where s_j^i represents the i-th message in the original trace j, and $<$ is a partial ordering function. Here we use (...) to denote a tuple, [...] to denote an ordered list, and {...} to denote an unordered set.

The input traces are HTTP requests and responses captured during multiple runs of user demonstration. The alignment algorithm identifies for each message what are the corresponding messages in another trace that are structurally similar. There are a large collection of work on computing web page similarities for clustering, such as [40]. However, since these work focus on the *content-wise* similarity, while we are mainly interested in the similarity of the *metadata*, such as URL parameters and cookies, we define the similarity of two message to be the length of the longest common subsequence of the hostname, the URL path, the GET or POST parameter names, and the names in the `<form>` elements. From these alignment pairs, our approach compute an overall alignment so that the size of aligned message set is maximized. We then discard messages that are not aligned or aligned messages that remain constant in all traces.

At this point, we have computed the set of aligned message tuples to be included in M. The partial ordering of any two aligned messages $m_1, m_2 \in M$ is defined as

$$m_1 < m_2 \Leftrightarrow \forall s_j^i \in m_1, \forall s_j^k \in m_2, i < k \tag{3.2}$$

Data Propagation

In the data propagation phase, the aligned HTTP messages will be propagated into a set of data types. This process serves two goals: First, we want to extract the data contained in each concrete message. For example, among the messages in Listing 3.1, we want to extract the value of the CSRF token embedded in the HTML response content, and also from the content of the POST request encoded as form data. In order to achieve this, we need to handle a variety of customized embedding and encoding of the data. Secondly, we want to discover the invariants between the data fields from different messages. For example, the CSRF token encoded in the POST request should be equal to the CSRF token in the HTML page of a previous response.

We first introduce a data structure that efficiently represents the process of data propagation. This data structure will be used as the input to the model generation phase.

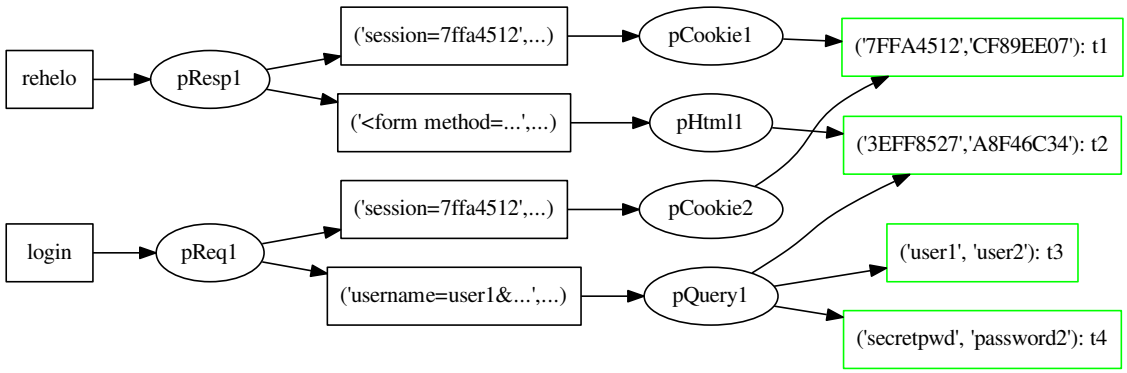

Figure 3.4: Example DFG for the Bodgeit Store

Data Structure We use *data-function graphs (*DFG*)* to keep track of how the data fields are extracted from the messages and the invariants on the data fields. A DFG is a bipartite directed acyclic graph $G = (T, A, E)$ where

1. T is the set of all data types. They are represented as a tuple of their concrete values in the example traces.
2. $A = T \times F$ is the set of function applications on data types, where F is a set of decoding functions defined in Table 3.1. Each function $f \in F$ takes one input data type ($t \in T$) and produces one or more output data types. Since a function can be applied to multiple elements in T, we use $a = (t, f) \in T \times F, t \in T, f \in F$ to uniquely denote a particular application of a function.
3. $E \subseteq T \times A + A \times T$ is the set of all the edges in G. For all $a \in A$, a's incoming edges $E_{in}(a) \subseteq T \times A$ connect a and its argument data type, and a's outgoing edges $E_{out}(a) \subseteq A \times T$ connect a and its result datatypes.

An example DFG for the Bodgeit Store is shown in Figure 3.4. We omit the irrelevant messages and the content of the non-leaf nodes in the graph for simplicity.

For each server, we take the set of its relevant messages and generate a DFG. Such a DFG has the following properties:

1. The roots of the DFG is the aligned message set in M, e.g., `rehelo` and `login` in Figure 3.4.
2. The leaves of the DFG form the set of data types in the protocol. This data type set will appear in the data type declaration of the synthesized model, e.g., the four leaf nodes in Figure 3.4.
3. The leaves reachable from a root message form the set of data fields carried by the message. For example, message `rehelo` has two fields with values (`'7FFA4512'`, `'CF89EE07'`) and (`'3EFF8527'`, `'ABF46C34'`) from two traces.

4. If a leaf node is reachable by multiple roots, it means the same data appears in multiple messages, which may imply a cross-origin sharing of knowledge, or a challenge-response-based authentication. The synthesizer will use this to infer invariants.
5. The paths from a root to a leaf represent how a field can be decoded from a message, and the reverse paths indicate the way to construct a concrete message according to the values of its fields. They will be used to construct a concrete attack trace from an abstract one, e.g., (`'user1'`,`'user2'`) is extracted by decoding the content of an HTTP response message as a URL query string.

Intuitively, the DFG mimics how a human analyst would dissect a message into small string fragments and build correlations between the messages according to the value of the fragments. Next, we are going to present how to automatically generate a DFG for each endpoint, and how to use it to synthesize the server logic.

DFG Generation In the DFG generation phase, we generate a DFG for each endpoint in the protocol using the example messages. The graph generation algorithm is shown in Algorithm 1. It is a worklist based graph construction algorithm. The worklist is initialized with the roots of the graph, i.e. the messages. Each element in the worklist is a pair (t, f) where t is a data type node to be extended, and f is the function to be applied on the data type. If the application is successful, a new application node (t, f) and a set of new data type node corresponding to the return values of $f(t)$ will be added to the graph, together with the edges that connect them. And for each returned data types t' of $f(t)$, all the possible function applications (t', f_i) will be pushed to the worklist. The propagation iterates until a fixedpoint is reached.

Propagation Functions A key factor affecting the size of the DFG search space is the functions used in the propagation, which we call the *propagation functions*. The propagation functions used are listed in Column 2 of Table 3.1. The choice of the next functions to try for each data type determines the search space of the graph. The propagation algorithm choose different subsets of functions to try depending on where the data type is from. The subset is determined by the functions that output this data type. Columns 3-6 of Table 3.1 lists which functions will be pushed to the worklist for each return value of a function. Taking the first row as an example, when (t, f) = (`login, pHttpReq`), The `while` loop in Algorithm 1 will try to parse the input string t as a HTTP request, and if successful (i.e., `pHttpReq(login)` $\neq \varnothing$), produces 11 (v, f') pairs. The first pair (v_1, f_1' in the table) is the URL of the request and function 6:`pUrl`, which means `pUrl` will be tried on this URL in a future iteration. Similarly, the last 9 pairs are indicated by v_3, f_3' in the table, which means that all functions 3-11:`pJS-pConcat` will be tried on parsing the request content.

Function `computeDFG(`M`)`
  Data: M: The result of the trace alignment.
  Result: G: The data-function graph of M.
  $finished = \varnothing$;
  forall the $m_i \in M$ **do**
    $worklist.push((m_i, parseReq))$;
    $worklist.push((m_i, parseResp))$;
  end
  while $worklist \neq \varnothing$ **do**
    $t, f \leftarrow worklist.pop()$;
    $T \leftarrow T \cup \{t\}$;
    if $(t, f) \notin finished \wedge f(t) \neq \varnothing$ **then**
      $A \leftarrow A \cup \{(t, f)\}$;
      $E \leftarrow E \cup \{(t, (t, f))\}$;
      forall the $(v, f') \in f(t)$ **do**
        $E \leftarrow E \cup \{((t, f), v)\}$;
        if $(v, f') \notin finished$ **then**
          $worklist.push((v, f'))$;
        end
      end
    end
    $finished.push((t, f))$;
  end

Algorithm 1: Data-function graph generation.

No.	f	v_1	f'_1	v_2	f'_2	v_3	f'_3	v_4	f'_4
1	pHttpReq	url	6	cookie	7	content	3-11		
2	pHttpResp	content	3-11	setcookie	7	redir	6		
3	pJS	token	3-11						
4	pJson	value	3,5-11						
5	pHtml	link	6	value	3-11				
6	pUrl	host	-	path	10	query	8	frag.	3-11
7	pCookie	value	3-11						
8	pQuery	value	3-11						
9	pUrlEscaped	orig	3-8						
10	pPath	item	-						
11	pConcat	item	-						

Table 3.1: Propagation function signatures. The numbers in Columns 3-6 refer to the function indices in Column 1.

Model Generation

The model generation phase translates the DFG to a model in our DSL. First, we collect all the web origins in the trace to form the set of endpoints. Secondly, for each endpoint and

No.	f	Description
1	pHttpReq	Try parsing the input string as an HTTP request and extracting its URL, cookie string and content.
2	pHttpResp	Try parsing the input string as an HTTP response and extracting its content, setcookie string and redirection URL if it is a HTTP 302 response
3	pJS	Try parsing the input string as a Javascript source code and extracting the list of tokens in the parsed abstract syntax tree(AST).
4	pJson	Try parsing the input string as a JSON source code, flattening the parsed tree into a list of (path-to-leaf, leaf-value) pairs, and extracting the list of values.
5	pHtml	Try parsing the input string as a HTML source code and extracting a list of the URLs and form field values found in the source.
6	pUrl	Try parsing the input string as a URL and extracting the host, path, query and fragment values.
8	pQuery	Try parsing the input string as a URL query string and extracting a list of values sorted by their keys.
9	pUrlEscaped	Try parsing the input string as a URL-escaped string and unescaping it.
10	pPath	Try parsing the input as a UNIX path string and extracting each individual directory name.
11	pConcat	Try parsing the input as a concatenated string with some delimiter and extracting a list of the individual items before the concatenation.

Table 3.2: Propagation function description

each request-response pair, we infer the corresponding message definition by collecting all the leaf data type nodes on the DFG. Third, we extract all the invariants by examining the multi-root leaves on the DFG. Finally, we infer the initial knowledge set of each endpoint according to the first appearance of each data types.

Inferring the message fields The set of fields for each message is defined by the set of reachable leaves from the message in the DFG.

$$m.\text{fields} := \text{DFG.leavesof}(m)$$

In our running example, the first request and the last response carries no data. The first response carries a cookie in `jsid` and a CSRF token in `csrf`. And the second request carries a cookie in `rjsid`, a CSRF token in `rjsid` , a username in `username` and a password in `password`.

Inferring the invariants For each field in a request message, we need to determine if it is supposed to contain a previously unseen data type or a known one, and for each field in a response message, we need to determine if the server needs to construct a new value of that type or use a known one. To achieve this, we first define the *relevant ancestor messages (RAM)* of a request as

$$\text{RAM(req)} := \{m | m < \text{req} \wedge m.\text{fields} \cap \text{req.fields} \neq \varnothing\}$$

The RAM of a request is the set of all history messages who have some fields equal to a request fields. We use their fields RAM(req).fields and the fields of the request to form the set of all candidate data-fields for constructing the response message. Note that an extended, alternative definition of relevant ancestor messages, which we denote RAM*(req), is

$$\text{RAM}^*(\varnothing) := \varnothing$$
$$\text{RAM}^*(\text{req}) := \text{RAM}(\text{req}) \cup \text{RAM}^*(\text{RAM}(\text{req}))$$

This definition extends the original RAM(req) by also including its transitive dependencies, which leads to a more complete approximation of the real session history. We do not use this definition in order to trade for a faster synthesis algorithm with reduced completeness.

Next, we generate request invariants between fields in req.fields and fields in RAM(req).fields, in the form of

```
forall m1:req, m2:??, ... {
  m1.?? == m2.?? <=> m1.?? == m2.??
  ∧ m1.?? == m?.?? <=> m1.?? == m?.??
  ∧ ...
}
```

This simulates the server validating the requests according to prior knowledge, such as validating a CSRF token. We also generate response invariants between fields in resp.fields ∪ req.fields) and fields in RAM(req).fields, in the form of

```
forall m1:resp, m2:req, m3: ??, ... {
  m2.?? == m3.?? <=> m2.?? == m3.??
  ∧ ...
  ∧ m1.?? == m3.?? <=> m1.?? == m3.??
  ∧ ...
}
```

This simulates the server using the requests to identify which session this request is with regard to, as well as what are the relevant data previous recorded, such as a database query, and then using all these relevant data to construct the response. For example, from the Bodgeit Store's DFG we find that both the cookie (`t1`) and the CSRF token (`t2`) are shared between `rehelo` and `login`. Thus RAM(login).fields = rehelo.fields = $\{t1, t2\}$, and as a result, we generate the following invariant:

```
forall m1:rehelo, m2:login {
  m1.jsid == m2.rjsid <=> m1.csrf == m2.rcsrf;
}
```

Inferring the initial knowledge sets The initial knowledge sets can be trivially extracted from the message declaration: For each endpoint, its initial knowledge set is the set of data types which are first used by this endpoint. In our running example, the username `t3` and password `t4` are initially known by the user, and the cookie `t1` and the CSRF `t2`

are initially known by the Bodgeit Store. Note that there is a mistake here made by the synthesizer, since technically the username and the password should be initially known by both the server and the client, but in our case studies (Section 3.8) we show that this is irrelevant if the vulnerability we discover does not rely on this fact, and if it later becomes relevant, the analyst can correct this mistake by providing some feedback.

3.6 Verification and Feedback

In this section, we present how the synthesized protocol model is used in verification, and how the analyst can provide feedback to WebSyn and refine the synthesis search space according to the MDL model and the verification result.

Verification

The verification component takes the MDL model and embeds it into a general model of web-based distributed systems (called *the base model*). The whole instance is then used for model checking.

In this extended model, the problem of finding a vulnerability is converted into the problem of generating a trace of HTTP messages that satisfies a set of constraints. The constraints can be divided into 4 categories: the trace constraints, the endpoint constraints, the protocol constraints and the policy constraints.

The trace constraints The trace constraints define the well-formedness of an HTTP trace. Some example trace constraints include that a `Trace` instance in Alloy is a list of `Message` instances, a `Message` instance is either a `Request` instance or a `Response` instance, `Request` instrances and `Response` instances appear alternately in a `Trace`, a `Response` instance consists of URL parameters, set-cookie fields, a response content, and a redirection location, etc. Note that these trace constraints are only used to constrain the shapes of the attack traces. While we do not claim that all vulnerabilities can be exploited under these constraints, these constraints help to reduce the search space significantly. The trace constraints are part of the base model.

The endpoint constraints The endpoint constraints specify the constraints between two adjacent messages in the trace, i.e., how each endpoint reacts to an incoming message. Intuitively, these constraints enforce the following endpoint capabilities:

1. The set of endpoints consists of a benign client, a malicious client, a malicious server and a set of benign servers defined by the synthesized MDL specification.
2. The *benign client* can send arbitrary request permitted by the rule of the server. When it receives a redirection response, it immediately sends a request with the url specified by the redirection.

3. The *malicious client* inherits all the capabilities of the benign client. Additionally, it may choose to not follow the redirection when receiving a redirection response. The malicious server and the malicious client can cooperate and share the same knowledge about the data.
4. The *malicious server* accepts arbitrary requests and can construct arbitrary redirection responses.
5. For each *benign server*, it only accepts requests and send responses according the synthesized MDL specification, which is separately called *the protocol constraints* since they are not part of the base model.

The endpoint constraints are part of the base model.

The protocol constraints The protocol constraints consist of protocol specific constraints translated from the MDL specification. It follows the semantics of the MDL and enforces that the generated trace is accepted by the MDL specification.

The policy constraints The policy constraints are translated from the initial vulnerability description, which is defined as the malicious server causing the benign client to send a request containing the attacker supplied data (cross site request forgery). More precisely, we reduce the question of whether the protocol is vulnerable to the question of whether there exists an execution, while conforming to the constraints imposed by the MDL model and the base model, also satisfies the following constraints:

1. There is a redirection response from the attacker.
2. The response is sent to the benign client.
3. The response is followed by a request sending to a benign server.
4. The request's non-cookie fields are all known by the attacker.

The corresponding ALLOY predicate is shown in Figure 3.5. Note that these constraints do not try to describe and enumerate all possible CSRF attacks to a vulnerability. For example, not every CSRF attack includes a redirection, and there are even more sophisticated ones that do not conform to any of these constraints. We are only searching for vulnerabilities that can be exploited in this particular way.

```
1 pred iscsrf[e: HTTPReqMessage] {
2    (some e.prev and e.prev in Resp_redir_mal2ua)
3    (e.from = UA)
4    (e.to in (ServerEP − MalEP))
5    some (e.payload − e.cookies)
6    (e.payload − e.cookies in queryKnown[MalEP, e])
7 }
```

Figure 3.5: ALLOY predicate that checks for CSRF vulnerabilities

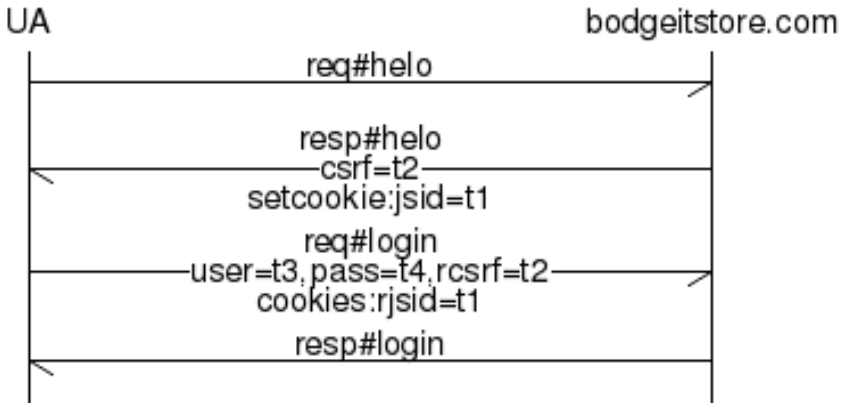

Figure 3.6: The message sequence chart illustrating the protocol synthesized for the Bodgeit Store.

The verification result The result of the verification is one of the following:

1. **Safe:** The model checker can not find an instance of `Trace` which is consistent with all the constraints. This means the protocol is safe under the synthesized model.
2. **Unknown:** The search space is too large and the model checker fails because of a timeout.
3. **Vulnerable:** The model check is able to find an instance of `Trace` that satisfies all the invariant. This means that there exists a session integrity vulnerablilty in the synthesized model.

For all the three case, a synthesized protocol will be visualized to the analyst in a message sequence chart, as shown in Figure 3.6. Additionally, in the case when the protocol is vulnerable, a trace will be presented to the analyst as the demonstration of the attack. Both the protocol and the attack trace are interactive so that the analyst can inspect them and provide feedback to refine the synthesis.

For our running example, the verification returns `Safe` because the login activity is properly protected by the CSRF token. However, if we remove the CSRF token, the verification will return an attack trace exhibiting a login CSRF attack, as illustrated in the message sequence chart in Figure 3.7.

User Feedback

In the user feedback component, the analyst inspects the results by reading the message sequence charts (e.g. Figure 3.6 and 3.7) or replaying the attack trace to the actual web service, and reaches a conclusion on the correctness of the synthesized protocol and whether the attack trace is spurious. If the synthesis is not accurate or the attack trace is spurious, the analyst provides more hints to the synthesizer. Our system accepts three types of hints. We elaborate on each of them and their effects on the search space in the following paragraphs.

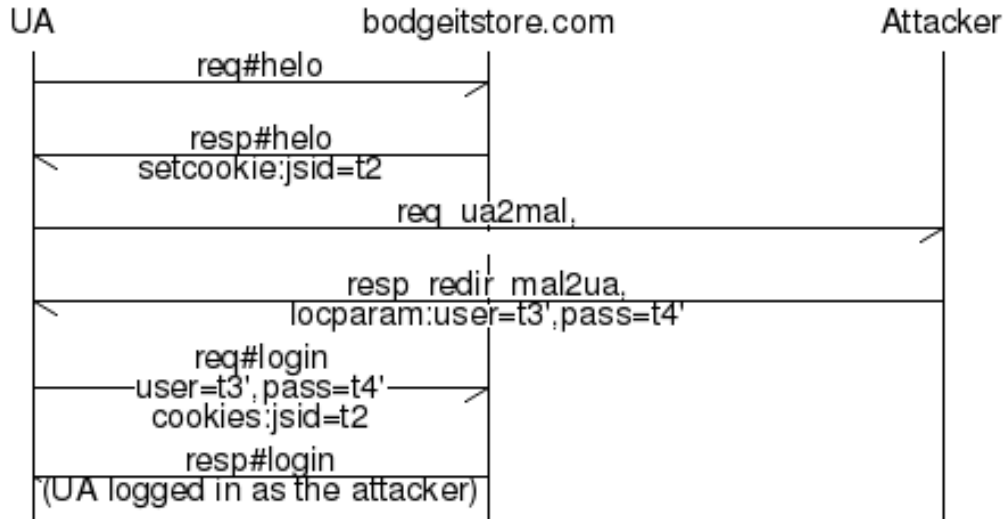

Figure 3.7: The message sequence chart for a login CSRF attack on the Bodgeit Store.

1. Input hints, in which a new input value is provided as an alternative to the input that are previously constant, or an annotation is provided to mark one of the current data types as unimportant. For example, if initially the analyst demonstrated the use of the Bodgeit store using different accounts but bought the same item, the analyst could create another demonstration in which the analyst buys a different item, thus making the item to buy and its price a new data type. This is called a *positive* input hint. On the other hand, if the analyst think that the CSRF token is guessable, the analyst could mark it as unimportant, thus removing the CSRF token from the model when searching for CSRF attacks, and as a result, the model checker would return an attack trace that ignores the CSRF token. This is called a *negative* input hint.
2. Scope hints, which are boolean answers to one or more URLs indicating whether similar URLs should be excluded or included in the example traces. A regular expression will be synthesized from these answers. For example, if the analyst were to buy two items in two demonstrations, and if the synthesizer included the HTTP messages related to loading the images of the items, the analyst could mark the image URLs as a *negative* scope hint, which means that URLs like these should be excluded from the synthesized specification.
3. Target hints, which are boolean answers to whether one or more messages should be considered as a critical message. By default, all messages in the synthesized specification are considered critical. However, sometimes we want to only focus on CSRF attacks that end at a particular HTTP request, in order to reduce the search space, and sometimes we want to assume a particular HTTP request is safe and proceed to find CSRF attacks targeting other HTTP messages. We use *positive* and *negative* target hints to handle the above cases, respectively.

No user feedback is needed for the running example, but in the case studies section (Section 3.8) we will demonstrate how it can help reduce the search space and adjust the search strategy.

3.7 Implementation

We have implemented a prototype system of WEBSYN. It consists of 2000 lines of python code and 150 lines of the initial ALLOY code. The additional ALLOY model compiled from MDL ranges from 250 to 1100 lines of code. We use the standard Selenium IDE and WebDriver to record user demonstrations and generate more traces with alternative inputs [60]. We use the BrowserMob proxy [50] to capture the network traces and output the trace files in the HTTP Archive (HAR) format. HAR is a standard format and analysts can rely on other tools such as the Firefox DevTools, the OWASP Zed Attack Proxy, Fiddler or the Burp Suite [54] to generate HAR traces. Our code is freely available online[16] including the original HAR traces, intermediate results and final results mentioned in the evaluation section.

3.8 Evaluation

We used WEBSYN to identify vulnerabilities in three real world applications. These applications represent simple, medium and complex protocols and cover all three kinds of user feedback. The CAS protocol case study also provides a comparison with previous work on manually writing its formal model. We rediscover two (previously found manually) vulnerabilities and discover four previously unknown vulnerabilities. We also include a fourth case study to demonstrate the limitation of our system. We summarize the configuration and the performance of each iteration in Table 3.3. We performed all the experiments on a desktop machine with Intel i5 670 3.4GHz CPU and 8GB memory. Each of the case studies involve at most 3 iterations until we find an "interesting" attack, and we present the user interactions in details to demonstrate that the user interactions are actually very simple but effective. The performance is moderate considering it is an *offline* analysis without interrupting the web services. From the performance evaluation we also learn that the analyst's simple feedback can have significant impact on the size and shape of the search space.

NeedMyPassword.com

NeedMyPassword.com is an online password manager. Next, we present our experience of testing it with WEBSYN.

The initial iteration The analyst provides execution traces containing examples of system behavior as input to WEBSYN. For the initial iteration, we consider execution traces that

Name	Websites	New Hints	#Msgs	#Types	#vars	#clauses	Verif. Time (s)	Attack?
NMP	nmp.com	None	8	16	1454	109187	7.20	Y(New)
		Target(-)	8	16	1454	109217	9.53	N
		Input(+)	8	22	1778	121922	8.16	Y
CAS	regclass.edu auth.edu	None	12	18	2022	186261	7.17	Y
		Target(-)	12	18	2022	186297	41.71	Y
		Target(+)	12	18	2022	186297	8.63	Y
		None	12	18	1998	170923	>7200	N
GOV	govtrack.us facebook.com	None	48	164	74750	25312696	>7200	N
		Scope(-)	24	86	16140	2293730	699.91	Y(New)
		Target(-)	24	86	16140	2293802	2399.77	Y(New)
		Target(+)	24	86	16140	2293874	149.15	Y
JIGO	localhost paypal.com	None	32	56	23678	4338001	-	-

Table 3.3: Configuration and performance of the case studies. The first column lists the names of our case studies. The second column lists the servers involved. The third column list the new hints provided by the analyst in each iteration. For each type of the hint, we use "+" to indicate a positive answer, and "-" to indicate a negative answer. The fourth column lists the number of message types in the synthesized model. The fifth column lists the number of data types. The sixth column lists the number of primary variables of the SAT instances generated by the ALLOY analyzer. The seventh column lists the number of CNF clauses in the SAT instances. The eighth column lists the verification time. We bound the verification to take up to 7200 seconds. The last column lists whether we find any attack traces. The protocol synthesizer terminates within 5 seconds in all the case studies.

open(*needmypassword.com*)
type(cssselect=#username,*account*1)
type(cssselect=#password,*password*1)
click(cssselector=input.submit)
type(cssselect=#recname,*name*1)
type(cssselect=#usr,*usr*1)
type(cssselect=#pwd,*pwd*1)
click(cssselect=.enter_password)

Figure 3.8: Recording of first user demonstration on NeedMyPassword.

demonstrate the process of logging in and adding a new record of username and password to the database. See Figure 3.8 for the details of the execution trace.

The analyst also provides credentials for a second NeedMyPassword account (*account*2 and *password*2). The example generation component generates two traces using the two sets of user accounts, and synthesizes an MDL model. Then, the verification component embeds

the MDL model into the basic web model and the ALLOY checker searches for vulnerabilities. The verification component quickly returns with a login CSRF attack.

The login page of NeedMyPassword.com does not include a CSRF token, and as a result a malicious server could make a benign user log in to NeedMyPassword using the attacker's account (Figure 3.10). We were not aware of this vulnerability when we started this experiment. Depending on how careful a user is while adding new credentials to the NeedMyPassword database, this could be a severe vulnerability.

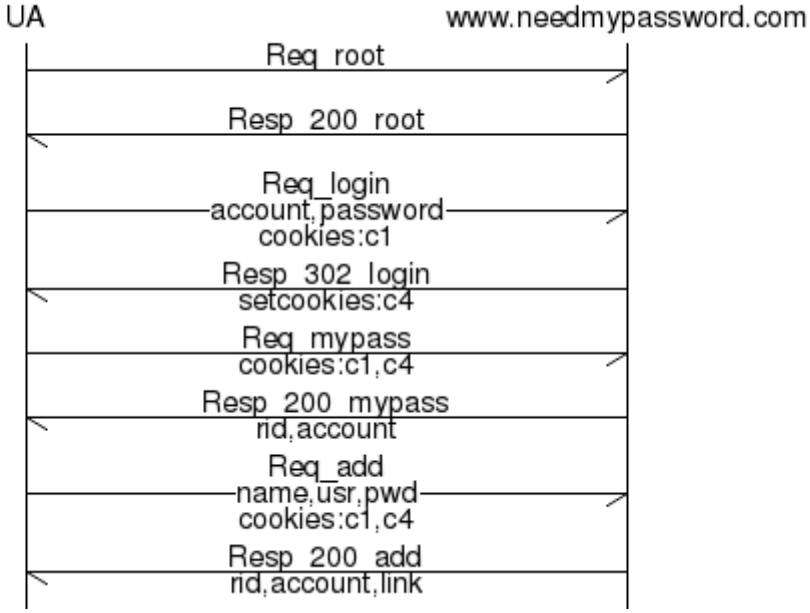

Figure 3.9: Synthesized protocol for NeedMyPassword.com.

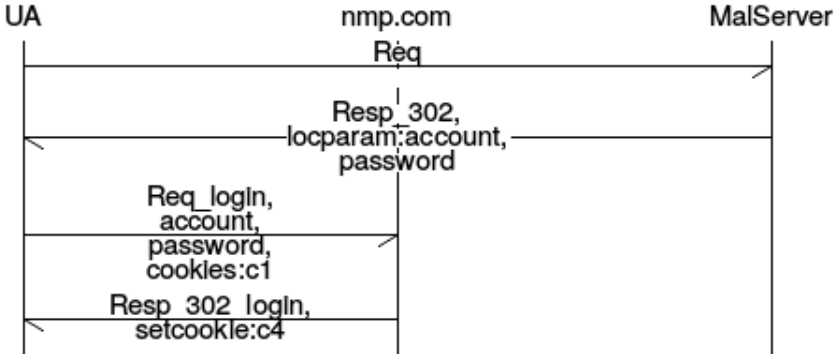

Figure 3.10: The session fixation attack for NeedMyPassword.com

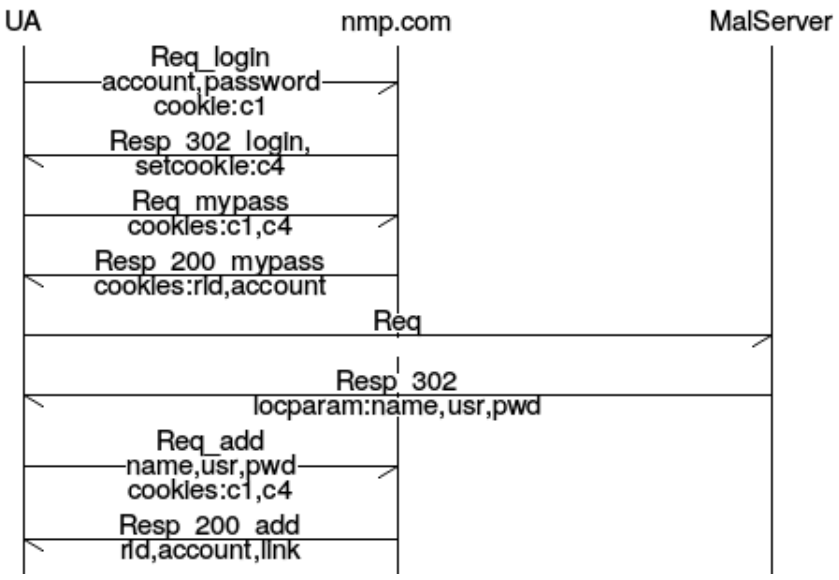

Figure 3.11: The second CSRF attack for NeedMyPassword.com

Negative target hint To continue with the search, we omit the login request (a target hint) and rerun the synthesizer to generate a new vulnerability description. The new description essentially excludes all the attack traces in which the CSRF requests are this particular request. The model checker returns `Safe`, but the analyst can still inspect the protocol and give more hints to refine the model.

Positive input hint The analyst can provide additional example execution traces; we considered an additional input trace demonstrating the step of adding credentials to the NeedMyPassword database. As a result of the extended input set, WEBSYN synthesizes a new, more-general protocol that represents a larger search space (See Figure 3.9 for the MSC of the synthesized protocol).

This time the verification component returns with a counterexample showing that there is a CSRF vulnerability in the password management step too (Figure 3.11). The CSRF vulnerability allows a malicious website to insert a new password record into a user's NeedMyPassword database. We also generated traces for editing and deleting existing password records and found that they are all vulnerable to the same kind of CSRF attacks.

We note that previous work *manually* identified the second CSRF vulnerability that we found [48]. This demonstrates the power of our approach; not only was WEBSYN able to rediscover known vulnerabilities from system execution traces, but its systematic analysis was also able to identify new vulnerabilities in protocols that have been manually vetted.

Accuracy We manually analyzed the website in order to evaluate the accuracy of our synthesized model. We found that the synthesized model failed to recognized that the

account credentials are shared between the user and the server. We also found that the cookie `c1` is actually irrelevant to the protocol and it is OK to not include it in the attack, while the synthesizer defaults to conservatively generate an invariant that enforce it to appear in every subsequent HTTP requests. Neither of them affected the correctness of the verification result.

The CAS Protocol

Next, we analyze the Central Authentication Service protocol (CAS)[51]. The CAS protocol was originally developed at Yale University and at least 80 universities currently deploy it[44]. Akhawe et al. manually wrote down the protocol model and identified a session fixation attack (later fixed) [2].

To further validate the fidelity of WEBSYN, we attempted to recreate and automatically identify this vulnerability. We captured example traces at our university by logging into a class registration system (twice) using the CAS protocol. We manually removed the fix to the Akhawe et al. vulnerability by deleting relevant nonces. Our system is able to synthesize two protocols from these two sets of traces. Figure 3.12 shows the vulnerable protocol.

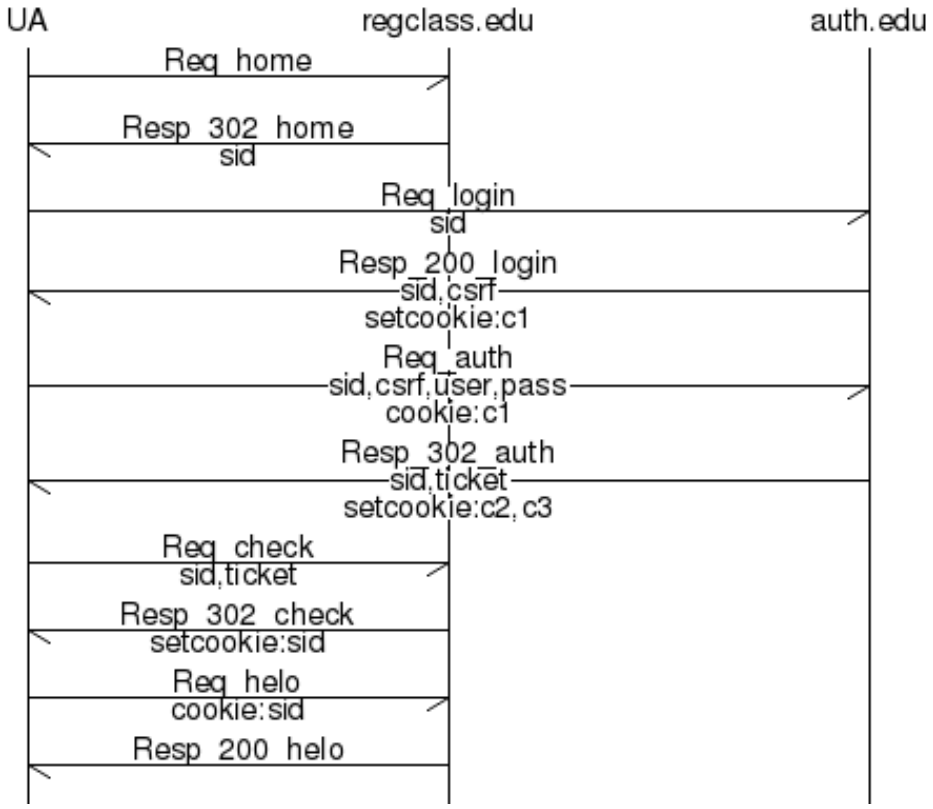

Figure 3.12: The vulnerable CAS protocol.

The initial iteration We first embed the vulnerable protocol into our base model and check for attack traces. The initial attack trace returned by the model checker is a valid CSRF attack that reuses the attacker's session ID in the victim's client (Figure 3.13. Interestingly, this attack that was missed by prior manual analysis. Similar to the NeedMyPassword.com case study, we provide a negative target hint to exclude this attack, and continue the search for more attacks.

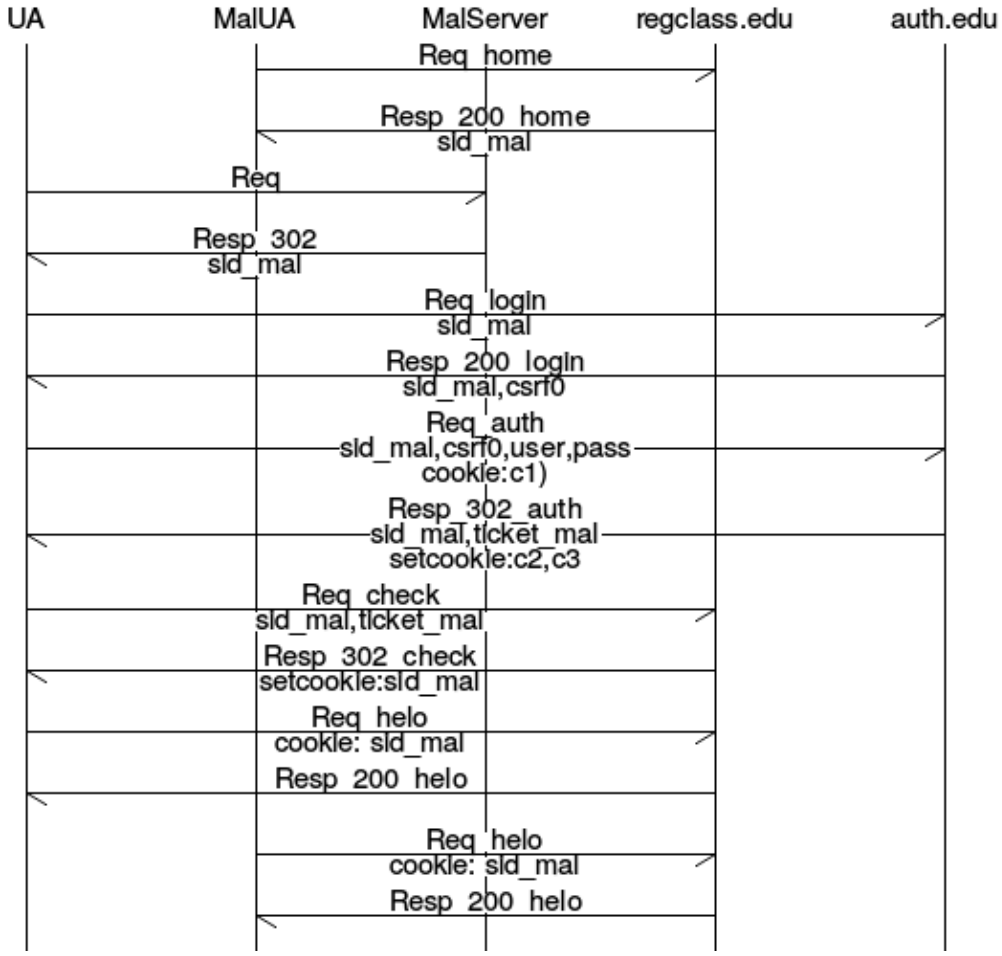

Figure 3.13: The first attack trace for the CAS protocol

Negative target hint The verification on the vulnerable protocol with the modified attack condition returns an attack trace as shown in Figure 3.14. The attack trace basically says that the attack can authenticate with the authentication server first and get a ticket. Instead of redirecting to `Req_check` in the attacker's browser, the attacker sends the link to a benign user who ends up logging in as the attacker on the user's browser. If the user is not aware of this, he or she could end up registering classes or paying tuition for the attacker. This finding

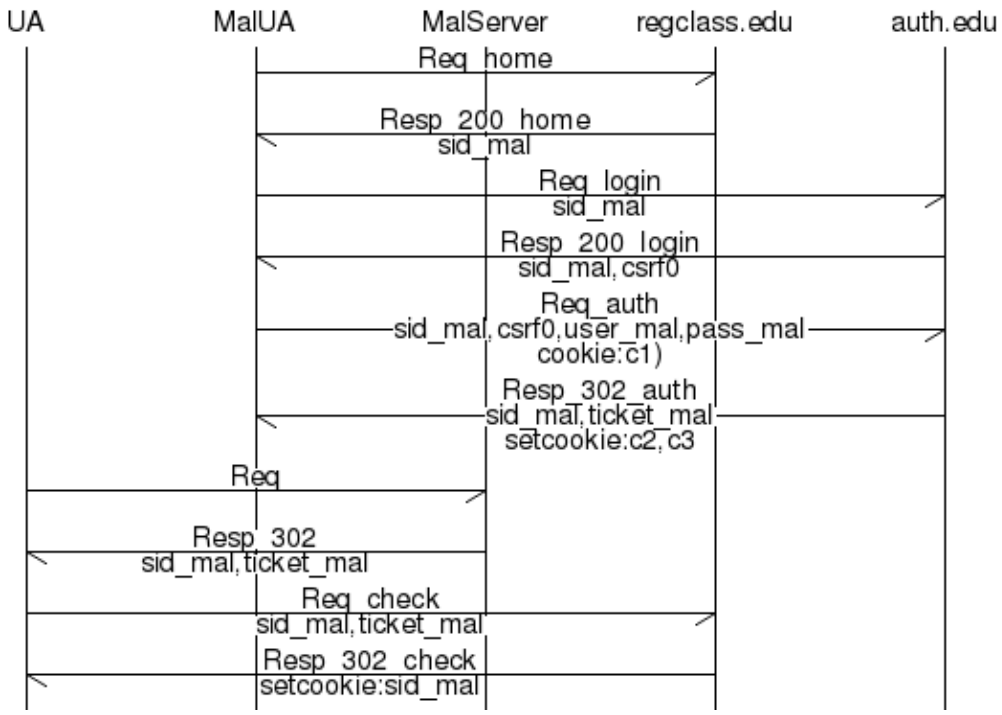

Figure 3.14: The second attack trace for the CAS protocol

validates the performance of WEBSYN. Recall that the vulnerability was previously found using significant manual analysis. Furthermore, the manual analysis missed the vulnerability discovered in the initial iteration.

Positive target hint To show the impact of analyst feedback on verification runtime, we tried using positive target hints instead the negative one above. We explicitly tell the model checker to find attack traces with the vulnerable message. After this change, the verification time was reduced from 41.71 seconds to 8.63 seconds. Positive hints like these could be feasible for a analyst who is familiar with the protocol and the messages affecting critical resources or access.

Verification on the fixed protocol We run the model checker with no hints on the fixed protocol. The verification lasts more than 2 hours, which is the timeout we set for all experiments. Due to the nature of SAT solvers we use, we do experience a significant increase in solver time in the absence of attacks. But the analyst always has the option of terminating the search and providing more hints.

Accuracy We compared our automatically synthesized model with the manually written model in Figure 4 of Akhawe et al.[2]. The synthesized MDL model contains two unnecessary

data fields, i.e., `c2` and `c3`. But similar to the previous case study, the verification phase is still able to find the vulnerability despite the noise.

Govtrack.us and Facebook Connect

Govtrack.us is a website for easily tracking the activities of the United States congress. It provides some social features where the user can associate his or her govtrack.us account with a Facebook account, and also login with Facebook accounts.

In this case study, we use the Facebook Connect API to associate govtrack.us accounts with Facebook accounts. The input execution trace includes logging into govtrack.us, using the "connect with Facebook" feature in govtrack.us, and logging into Facebook.

The initial iteration At first, the synthesizer generates a protocol with 48 HTTP messages, which is too large for the model checker, so the verification times out. When visualized as a message sequence chart, the analyst can easily see a lot of irrelevant XHR calls[1] when loading the Facebook homepage.

Negative scope hint Since these are irrelevant to the account association functionality of govtrack.us, the analyst picks two of these URLs, and tell the synthesizer that they are out of scope. The synthesizer generalizes these two URLs into a regular expression and excludes all similar messages. As a result, the protocol synthesized from the second iteration contains only 24 messages.

In the second iteration, our system successfully synthesized an attack trace, which says that the malicious server can initiate the association process between govtrack.us and Facebook when the user visits the malicious server. To exploit this vulnerability, the attacker would need the Facebook user to click the "Allow" button.

Negative target hint To continue searching for additional attacks, the analyst omits the association initiation request from the scope. The system returns another synthesized attack, which says that the malicious server can initiate the account association request to Facebook using the attacker's browser, and submit the final association request to govtrack in the benign user's browser.

The result of this attack is that the benign user's govtrack account binds with the attacker's Facebook account, and the attacker can login to the benign user's govtrack account using its "login with Facebook" feature.

Positive target hint We perform the same substitution from negative target hint to a positive one, and witness a significant reduction of the verification time from almost 40 minutes to 149.15 seconds. Even without such positive target hints, recall that our analysis is offline, and does not interrupt the web service.

[1] `https://en.wikipedia.org/wiki/XMLHttpRequest`

Both the attacks discovered in this case study are new, previously-unknown vulnerabilities, suggesting that WebSyn can be a useful tool for protocol analysis. We have reported the vulnerability to the developers of govtrack. They have acknowledged the vulnerability and promised to fix it promptly.

Jigoshop and Paypal Payments

In this section, we introduce a case study where our current implementation fails to synthesize a useful specification for identifying logic vulnerabilities like the ones in Wang et al.[73]. Jigoshop[2] is an online e-commerce application based on Wordpress. Its checkout process supports third-party payment systems such as the Pay-by-Paypal button.

We recorded two traces using the same Jigoshop and Paypal account but different items to buy. According to manual inspection of the traces, we suspect that the confirmation from Paypal to Jigoshop is not properly validated. The attacker could intercept the charge request from Jigoshop to Paypal and change the amount of the charge to a smaller value. If the Jigoshop failed to verify that the amount charged by Paypal is the same as the amount requested, the attacker could buy items with a much cheaper price.

However, we could not find any such vulnerabilities from the synthesized specification. This is because the security policy in our current implementation is not useful in this case, there is no benign client involved, no redirection is needed in order to launch such attacks, and a successful attack is not about submitting a valid charge request, but submitting a charge request with less amount. Writing such a security policy would require an understanding of the application-specific semantics of the data types, which can not be generalized. For example, it is very hard to automatically recognize the difference between changing the price of the item and changing the serial number of the item, without incorporating the domain knowledge about e-commerce applications.

3.9 Discussion

In this section, we present a discussion of some properties of our system and the future work.

System Limitations

Our implementation does have a number of limitations. First, our protocol synthesizer assumes a particular server logic detailed in Section 3.2. If the server does not follow this high-level model, we will not be able to synthesize a meaningful model. Second, our synthesizer currently only infer invariants from the traces' value differences. We simply ignore the HTTP messages that only appear in one trace. Finally, due to the our server logic assumptions and our current constraints on the attack traces, our system only synthesizes CSRF attacks on session integrity. However, there is a large variety of ways to launch such

[2] `https://www.jigoshop.com/`

attacks depending on the application specific logic, and one possible future work is to extend both the synthesizer and the security specification in order to incorporate other types of attacks. Our main contribution is in the general synthesis-based approach—the starting security specification to use is a separate problem.

System Extensibility

In order to support detecting other kinds of vulnerabilities, WebSyn needs to be extended in order to recognize relevant execution differences and synthesize more complex invariants. More specifically, we have looked into open redirect vulnerabilities. An *open redirect* vulnerability exists when an application takes a URL as a parameter and redirects a user to the parameter value without proper validation. For example, in the CAS protocol case study (Section 3.12), `Reg_login` contains the URL of `Req_check`, and the `auth.edu` server will redirect the user to this URL with the `ticket` attached. If the `auth.edu` server did not validate this redirection URL and just blindly redirected the user to any URL it received, it would be an open redirect vulnerability in `auth.edu`. An attacker could leverage this vulnerability to steal secret information such as the value of the `ticket`, or bypass referrer-based request validation. To support modeling such semantics, the analyst would need to introduce a new value difference in the demonstration traces, i.e., using a server data type instead of the constant `regclass.edu` to perform the CAS protocol with `auth.edu`. However, this poses a challenge to our data propagation algorithm because the new server's encoding of data could be very different from `regclass.edu`, and it is much harder to extract equivalent fields if they are not embeded in the messages in a similar way. We have recorded such a set of traces, and ran our current implementation unmodified on them, the propagation algorithm was able to extract the same set of data types for `auth.edu`, but failed to apply propagation functions other than `pResp` and `pReq` on other servers' messages, due to the significant structural difference.

Input traces and Errors

It is possible that the example execution traces do not fully specify the system behavior (contain ambiguity). To increase precision under this scenario, the analyst can provide additional example traces, as we have shown in our first case study.

It may also be possible that the analyst's feedback is noisy, and contains some hints that are incorrect or inconsistent. In this case, the synthesizer will fail to generate a model that excludes this trace. In future work, we plan to investigate advanced inductive algorithms such as version space algebra [47], as well as machine learning techniques [52], to add a probabilistic strategy in the synthesis. In other words, models that only fit a subset of the example traces can still be assigned a non-zero probability.

Search Space Exploration Unlike the classic Abstract-Check-Refine paradigm, the exploration performed by WebSyn is distinct from that of most other verification tools. Web-

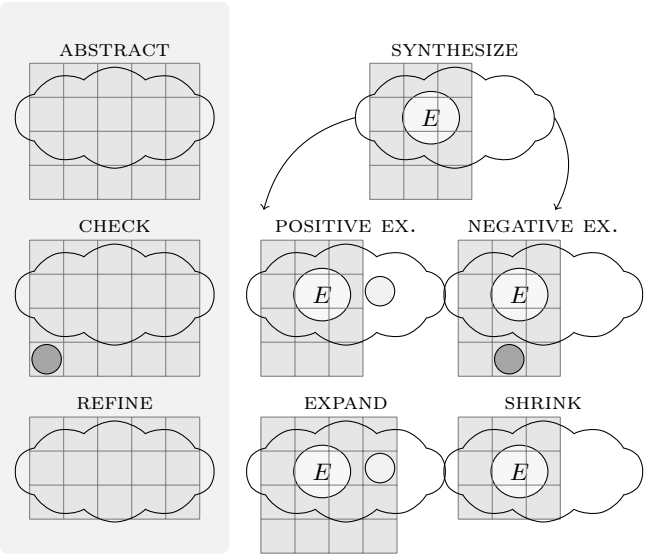

Figure 3.15: Conceptual view of how WebSyn explores a protocol with a comparison to counterexample guided refinement.

Syn employs a *Synthesize-Check-Refine* strategy. Figure 3.15 illustrates the comparison of the two in terms of the shape change of the search space. The cloud represents traces of a protocol implementation and the grid represents a protocol model. In a classic refinement process, an overapproximation of a system is constructed, analyzed to find a spurious error, which is then eliminated in the refinement step. In WebSyn, a protocol model is synthesized as a program. The model is only as good as the examples (E) provided by the analyst and may neither over- nor under-approximate the search space. Based on analyst feedback, this model may either *expand* to include more behavior, or *shrink* to eliminate spurious behavior. The two operations are independent, and the analyst may trigger both, leading to a novel, interactive exploration strategy that zooms in on relevant protocol behavior.

3.10 Related Work

Our work is motivated in part by Wang et al.'s series of mostly manual security analysis on cashier-as-a-service based web stores[73], single-sign-on web services[71] and protocol sdks[72]. The latter also demonstrates a systematic process of the interaction between the security analyst and a formal analysis tool. Their approach still requires a lot of manual efforts and the extent of manual efforts is neither well-defined nor minimized. In our approach, the iterative vulnerability discovery process is semi-automated, and the core analysis is formally encoded as a synthesis problem.

AuthScan[4] is a system for automatic extraction and checking of web authentication protocols. Although WebSyn and AuthScan share similar goals, our approach is very different.

AuthScan relies on symbolic execution and fuzz testing and AuthScan's TML language aims to provide a useful intermediate language for spi-calculus descriptions. In contrast, our approach is centered around the design of a high-level DSL to enable the interaction between human and computer. Instead of using low-level constructs such as "send/receive" as in TML, WebSyn's MDL directly captures the semantics of the web in the language constructs such as cookies and HTTP redirections.

Pellegrino and Balzarotti[56] propose a blackbox technique that identifies a number of behavioral patterns from network traces and generates test cases. Our approach differs from their approach in the constructed model, the desired security goal, and the algorithm to explore the search space. In their work, the model is a graph constructed from a collection of network traces. Three patterns are defined and searched on the graph, which are called singleton nodes, multi-step operations and waypoints. For each concrete subgraph matching these patterns, their system generate test cases that break these patterns, such as repeatedly visiting singleton nodes, breaking multi-step operations, or detouring the way points. These alternations to the patterns corresponds to a set of predefined logic flaws. A large amount of such test cases are generated, which includes a significant portion of false positives, and they are verified dynamically with a testing oracle. In our approach, the synthesize specification corresponds to a single path on their graph model, and we zoom into the nodes to synthesize the invariant for both the same data type across different messages and the co-appearance of two data types within the same message. Our security policy concerns CSRF vulnerabilities. The search for such vulnerabilities are directly performed on the synthesized model, which allows us to only produce a much smaller amount of counter-examples with lower false positives.

Invariant detectors like Daikon [29] inductively generate invariants from traces. The basic constructs of the invariants are low-level operations such as arithmetic or boolean operations, while the invariants in our DSL have additional semantics inherited from the base web model. Our approach is similar to the above work in that we also use an inductive approach to generate specifications. The difference in the choices of the abstraction level reflects the difference of vulnerabilities each approach focuses on, and different efficiency trade-offs.

Lie et al. [49] proposed a method to extract specifications automatically from program code using program slicing. Aizatulin et al. [1] proposed model extraction using symbolic execution. SLAM[6] and CEGAR[15] use predicate abstraction to construct models of program, and refine the model with counterexamples. All these proposals require access to the whole system and do not easily apply to distributed web protocols.

At the core of our system is the synthesis of the protocol description in a DSL. Program synthesis aims to develop technologies that can translate expressions of user intent into programs. For example, researchers have recently proposed techniques to automatically synthesize programs using input-output examples [36, 39], system predicates over input-output [64, 38], program template structure and constraints [65], natural language [18], and user demonstration [47]. Gulwani et al. provide a comprehensive overview of such techniques [37]. We present how modern inductive synthesis techniques can provide a framework

to automatically generate and model-check specifications of web protocols. Automatic generation of protocol specifications helps ease testing and analysis of these protocols. Previous work [2] enables automatic security analysis of specifications; our work focuses on automatic *generation* of these specifications.

Chapter 4

Conclusion

In the first part of the dissertation, we have presented a new approach to specifying and detecting malicious behavior in Android applications. Our conceptual contribution is the Permission Event Graph (PEG) a new, domain-specific program abstraction, which captures the context in which events fire, and the context-sensitive effect of event handlers. We devised a new static analysis procedure for constructing PEGs from Dalvik bytecode, and our implementation models of the Android event-handling mechanism and several APIs. Our system Pegasus can check security specifications that characterize interactions between user-driven events and application actions. Given the rapidly increasing popularity and sophisticated functionality of mobile applications, we believe that analysis systems such as Pegasus will improve the capabilities of security analysts.

Our work leads to several questions. One question is to incorporate existing techniques to improve the precision and efficiency of analyses used to construct and analyze PEGs. A particularly interesting question is to determine if counterexample-driven refinement can be used to improve both verification and rewriting. A second problem is to identify applications PEGs in other contexts, such as to measure the complexity and usability of user-interfaces, and statically provided permission information. Answering such questions is left as future work.

In the second part of the dissertation, we presented WebSyn, an interactive system that enables modeling and verification of web protocol implementations without access to the source code. WebSyn operates in the multi-lingual, black box environment of web development by using execution traces and analyst feedback to construct and refine protocol models. Our key insight is to leverage modern program synthesis techniques in inferring the protocol models. Our system eases the burden of manually translating the implementations of web applications into protocol models. Using several proof-of-concept case studies, we have demonstrated how WebSyn can discover both previously known vulnerabilities as well as new vulnerabilities from real world applications. Our research is the first step to explore the benefits of program synthesis and domain languages for enhancing application security.

Appendix A

Application, Action and Event Details

Table A.1 lists the sample applications used to evaluate Pegasus. Table A.2 lists the security actions and events used in the specifications.

Name	Description
Who's Calling?	An application that uses text-to-speech to announce the caller ID of an incoming call.
Share Contacts	An application that allows users to send and receive contacts via SMS.
Geotag	An application that embeds the userâĂŹs current GPS location into uploaded pictures.
Find My Phone	Responds with the GPS location of the device upon receiving a message containing a specific keyword.
Simple Recorder	An application that recording audio from the microphone.
Diet SD Card	An application that recommends files to delete from the SD card.
SMS Cleaner Free	An application that allows the user to delete SMS messages based on a variety of features.
SyncMyPix	An application that syncs pictures from Facebook friends to the contact list in the phone.
SMS Replicator Secret	Spyware that automatically forwards SMS messages to a number selected by the user installing the application.
ADSms	A malicious application that stealthily sends premium SMS messages and kills certain background processes.
ZitMo	A malicious application that intercepts SMS messages in order to harvest two-factor authentication codes issued by banks.
HippoSMS	A malicious application that stealthily deletes all incoming SMS messages.
DroidDream (Steamy Window)	A trojan packaged with a benign application that stealthily sends SMS messages, installs new applications and receives commands from a C&C server.
Zsone (iMatch)	A malicious application that sends SMS messages to premium numbers.
Geinimi (Monkey Jump 2)	A trojan packaged with a benign application that stealthily sends SMS messages, installs new applications and receives commands from a C&C server.
Spitmo	A trojan that intercepts and filters incoming SMS messages.
Malicious Recorder	Mimics a benign sound recorder but also records audio when an SMS message is received.

Table A.1: Sample applications used in the evaluation: 8 benign applications (top) and 9 malicious applications (bottom).

Name	Description
Access-Contacts	Access the contacts list.
Insert-Contacts	Insert to the contacts list.
Send-SMS	Send SMS.
Access-GPS	Access GPS location information.
Start-Recording	Start recording audio.
Stop-Recording	Stop recording audio.
BroadcastAbort	Abort an ordered broadcast (used by many malicious applications to prevent SMS messages from being displayed to the user).
Access-SD	Access the SD card.
Kill-Background-Processes	Kill background processes.
Read-IMEI	Read IMEI information.
Access-Internet	Access the Internet.
REC.onClick	The REC button's `onClick` handler.
STOP.onClick	The STOP button's `onClick` handler.
Clean.onClick	The Clean button's `onClick` handler in Diet SD Card.
Locate.onClick	The Locate button's `onClick` handler in Geotag.
Send.onClick	The Send button's `onClick` handler in Share Contacts.
Insert.onClick	The Insert button's `onClick` handler in Share Contacts.
Select-Contact.onClick	The Select-Contact button's `onClick` handler in SMS Cleaner Free.
Button.onClick	The general `onClick` handler for any button.
AdTask.onPreExecute	The `onPreExecute` handler from a ASyncTask created by Google Ads library in Geotag.
AdTask.doInBackground	The `doInBackground` handler from a ASyncTask created by Google Ads library in Geotag.
PhoneCall.onReceive	The `onReceive` event from a broadcast receiver in the Who's Calling? application.

Table A.2: Security actions and events used in specifications.

Appendix B

Full MDL Syntax Definition

Figure B.1 lists the full syntax of the MDL language.

PROTOCOL :=	SERVERS INIT MESSAGES INVARIANTS
SERVERS :=	**servers**: s1,s2,...;
INIT :=	**init**: s1 **knows** t1,t2,...; s2 **knows** t2,t3,...; ...;
MESSAGES :=	**messages**: MSGDEF MSGDEF ...
MSGDEF :=	**request**|**response** (server=s1, type=mt1, fields=(f1 **in** setcookie|content|urlparam|locparam|cookie, f2 **in** ...));
INVARIANTS :=	**invariants**: INVDEF INVDEF ...
INVDEF :=	ISA | FORALL
ISA :=	mt1.f1 **isa** t1;
FORALL :=	**forall** m1:mt1, m2:mt2 ... { BOOLEXP }
BOOLEXP :=	m1.f1 == m2.f3 | BOOLEXP BOP BOOLEXP
BOP :=	$\wedge$ | <=>

Figure B.1: The syntax of the MDL language.

Bibliography

[1] M. Aizatulin, A.D. Gordon, and J. Jürjens. "Extracting and verifying cryptographic models from C protocol code by symbolic execution". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM. 2011, pp. 331–340.

[2] Devdatta Akhawe et al. "Towards a formal foundation of web security". In: *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*. IEEE. 2010, pp. 290–304.

[3] K.W.Y. Au et al. "PScout: Analyzing the Android Permission Specification". In: *Proc. of the ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 217–228.

[4] G. Bai et al. "AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2013.

[5] Thomas Ball and Sriram K. Rajamani. *SLIC: A Specification Language for Interface Checking (of C)*. Tech. rep. MSR-TR-2001-21. Microsoft Resesarch, 2001.

[6] Thomas Ball et al. "Automatic predicate abstraction of C programs". In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 203–213.

[7] T. Ball et al. "Automatic Predicate Abstraction of C Programs". In: *Proc. of the Symposium on Programming Language Design and Implementation*. ACM, 2001, pp. 203–213.

[8] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. "Discovering concrete attacks on website authorization by formal analysis". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE. 2012, pp. 247–262.

[9] Chetan Bansal et al. "Keys to the cloud: formal analysis and concrete attacks on encrypted web storage". In: *Principles of Security and Trust*. Springer, 2013, pp. 126–146.

[10] Nels E. Beckman et al. "Proofs from tests". In: *Proc. of the International Symposium on Software Testing and Analysis*. ACM, 2008, pp. 3–14. ISBN: 978-1-60558-050-0.

[11] E. Bodden, P. Lam, and L. Hendren. "Object representatives: a uniform abstraction for pointer information". In: *Proc. of the 1st International Academic Research Conference of the British Computer Society (Visions of Computer Science)*. 2008, pp. 391–405.

[12] Patrice Chalin et al. "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *Proc. of the International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 342–363.

[13] Kevin Zhijie Chen et al. "ASPIRE: Iterative Specification Synthesis for Security". In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[14] Kevin Zhijie Chen et al. "Contextual Policy Enforcement in Android Applications with Permission Event Graphs." In: *Network and Distributed System Security (NDSS) Symposium*. 2013.

[15] Edmund Clarke et al. "Counterexample-guided abstraction refinement". In: *Computer aided verification*. Springer. 2000, pp. 154–169.

[16] *Code Release*. `http://websyn.weebly.com/`. 2014.

[17] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks". In: *Journal of Logic and Computation* 2 (1992), pp. 511–547.

[18] Anthony Cozzie and Samuel T. King. *Macho: Writing Programs with Natural Language and Examples*. Tech. rep. 2142.33791. University of Illinois at Urbana-Champaign, 2012.

[19] Charlie Curtsinger et al. "ZOZZLE: fast and precise in-browser JavaScript malware detection". In: *Proc. of the USENIX conference on Security*. USENIX Association, 2011, pp. 3–3.

[20] Rocco De Nicola and Frits Vaandrager. "Action versus state based logics for transition systems". In: *Proc. of the LITP Spring school on theoretical computer science on Semantics of systems of concurrent processes*. Springer, 1990, pp. 407–419.

[21] L. Desmet et al. "The S3MS.NET run time monitor". In: *Electronic Notes in Theoretical Computer Science* 253.5 (2009), pp. 153–159.

[22] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-sensitive interprocedural points-to analysis in the presence of function pointers". In: *Proc. of the Symposium on Programming language design and implementation*. PLDI '94. ACM, 1994, pp. 242–256.

[23] William Enck. "Defending Users against Smartphone Apps: Techniques and Future Directions". In: *International Conference on Information Systems Security*. Springer, 2011, pp. 49–70.

[24] William Enck, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification". In: *Proc. of the ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[25] William Enck et al. "A study of Android application security". In: *Proc. of the USENIX conference on Security*. USENIX Association, 2011, pp. 21–21.

[26] William Enck et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *Proc. of the USENIX Conference on Operating Systems Design and Implementation*. USENIX, 2010, pp. 1–6.

[27] Ú. Erlingsson. "The inlined reference monitor approach to security policy enforcement". PhD thesis. Cornell University, 2004.

[28] Úlfar Erlingsson and Fred B. Schneider. "SASI enforcement of security policies: a retrospective". In: *Proc. of the Workshop on New security paradigms*. ACM, 2000, pp. 87–95. ISBN: 1-58113-149-6.

[29] Michael D Ernst et al. "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1 (2007), pp. 35–45.

[30] Adrienne Porter Felt et al. "A survey of mobile malware in the wild". In: *Proc. of the workshop on Security and privacy in smartphones and mobile devices*. SPSM '11. ACM, 2011, pp. 3–14.

[31] Adrienne Porter Felt et al. "Android permissions demystified". In: *Proc. of the Conferenc. on Computer and Communication Security*. ACM, 2011, pp. 627–638.

[32] Adrienne Porter Felt et al. "Permission re-delegation: attacks and defenses". In: *Proc. of the USENIX Security Conference*. USENIX Association, 2011.

[33] S.J. Fink et al. "Effective typestate verification in the presence of aliasing". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17.2 (2008), p. 9.

[34] Matthew Fredrikson et al. "Efficient runtime policy enforcement using counterexample-guided abstraction refinement". In: *Proc. of the Conference on Computer Aided Verification*. Springer, 2012, pp. 548–563.

[35] Michael C. Grace et al. "RiskRanker: scalable and accurate zero-day Android malware detection". In: *Proc. of Mobile Systems, Applications, and Services*. ACM, 2012, pp. 281–294.

[36] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. URL: http://doi.acm.org/10.1145/1926385.1926423.

[37] Sumit Gulwani. "Dimensions in program synthesis". In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD '10. Lugano, Switzerland: FMCAD Inc, 2010, pp. 1–2. URL: http://dl.acm.org/citation.cfm?id=1998496.1998498.

[38] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. "Synthesizing geometry constructions". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 50–61. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993505. URL: http://doi.acm.org/10.1145/1993498.1993505.

[39] William R. Harris and Sumit Gulwani. "Spreadsheet table transformations from examples". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 317–328. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993536. URL: http://doi.acm.org/10.1145/1993498.1993536.

[40] Taher H Haveliwala et al. "Evaluating strategies for similarity search on the web". In: *Proceedings of the 11th international conference on World Wide Web*. ACM. 2002, pp. 432–442.

[41] Thomas A. Henzinger et al. "Lazy abstraction". In: *Proc. of the Symposium on Principles of Programming Languages*. POPL. ACM, 2002, pp. 58–70.

[42] Lin-Shung Huang et al. "Clickjacking: Attacks and Defenses." In: *USENIX Security Symposium*. 2012, pp. 413–428.

[43] Daniel Jackson. "Alloy: a lightweight object modelling notation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 256–290.

[44] JASIG. *The CAS Protocol Deployment*. Online. http://www.jasig.org/cas/deployments. 2010.

[45] Ranjit Jhala and Rupak Majumdar. "Interprocedural analysis of asynchronous programs". In: *Symposium on Principles of Programming Languages*. Vol. 42. POPL 1. ACM, Jan. 2007, pp. 339–350.

[46] Ranjit Jhala and Rupak Majumdar. "Software model checking". In: *ACM Computing Surveys* 41.4 (Oct. 2009), 21:1–21:54. ISSN: 0360-0300.

[47] Tessa Lau et al. "Programming by demonstration using version space algebra". In: *Machine Learning* 53.1-2 (2003), pp. 111–156.

[48] Zhiwei Li et al. "The emperor's new password manager: Security analysis of web-based password managers". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014.

[49] D. Lie et al. "A simple method for extracting models from protocol code". In: *Proceedings of the 28th Annual International Symposium on Computer Architecture*. IEEE. 2001, pp. 192–203.

[50] Lightbody. *The BrowserMob Proxy*. Online. http://bmp.lightbody.net/. 2014.

[51] D. Mazurek. *The CAS Protocol*. Online. http://www.jasig.org/cas/protocol. 2005.

[52] Aditya Menon et al. "A machine learning framework for programming by example". In: *Proceedings of The 30th International Conference on Machine Learning*. 2013, pp. 187–195.

[53] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: extending Android permission model and enforcement with user-defined runtime constraints". In: *Proc. of the Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332. ISBN: 978-1-60558-936-7.

[54] OWASP. *Testing Tools*. Online. https://www.owasp.org/index.php/Appendix_A:_Testing_Tools. 2014.

[55] É. Payet and F. Spoto. "Static analysis of Android programs". In: *Proc. of the Conference on Automated Deduction*. Springer, 2011, pp. 439–445.

[56] Giancarlo Pellegrino and Davide Balzarotti. "Toward Black-Box Detection of Logic Flaws in Web Applications". In: *Network and Distributed System Security (NDSS) Symposium*. NDSS 14. San Diego (USA), Feb. 2014.

[57] Psiinon. *The Bodgeit Store*. Online. `https://code.google.com/p/bodgeit/`. 2010.

[58] Franziska Roesner et al. "User-driven access control: Rethinking permission granting in modern operating systems". In: *the 33rd IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 224–238.

[59] R. Sekar et al. "Model-carrying code: a practical approach for safe execution of untrusted applications". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 15–28.

[60] *IDE and WebDriver*. `http://seleniumhq.org/`. 2014.

[61] S. Shoham et al. "Static specification mining using automata-based abstractions". In: *IEEE Transactions on Software Engineering* 34.5 (2008), pp. 651–666.

[62] Rishabh Singh and Sumit Gulwani. "Synthesizing number transformations from input-output examples". In: *Proceedings of the 24th international conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 634–651. ISBN: 978-3-642-31423-0.

[63] Armando Solar-Lezama. "Program synthesis by sketching". PhD thesis. University of California, Berkeley, 2008.

[64] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. "From program verification to program synthesis". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 313–326. ISBN: 978-1-60558-479-9. DOI: `10.1145/1706299.1706337`. URL: `http://doi.acm.org/10.1145/1706299.1706337`.

[65] Saurabh Srivastava et al. “Path-based inductive synthesis for program inversion”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 492–503. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993557. URL: http://doi.acm.org/10.1145/1993498.1993557.

[66] San-Tsai Sun and Konstantin Beznosov. “The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems”. In: (Aug. 2012).

[67] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. “Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures”. In: *Computers & Security* (2012).

[68] Emina Torlak and Rastislav Bodik. “Growing solver-aided languages with Rosette”. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM. 2013, pp. 135–152.

[69] Raja Vallée-Rai et al. “Soot - a Java bytecode optimization framework”. In: *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research*. CASCON ’99. IBM Press, 1999, pp. 13–.

[70] Moshe Y. Vardi. “From Philosophical to Industrial Logics”. In: *Proc. of the Indian Conference on Logic and Its Applications*. Springer, 2009, pp. 89–115.

[71] Rui Wang, Shuo Chen, and XiaoFeng Wang. “Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 365–379.

[72] Rui Wang et al. “Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization”. In: *Proceedings of the USENIX Security Symposium*. USENIX. 2013.

[73] Rui Wang et al. “How to shop for free online–Security analysis of cashier-as-a-service based Web stores”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 465–480.

[74] Yajin Zhou and Xuxian Jiang. “Dissecting android malware: Characterization and evolution”. In: *the 33rd IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 95–109.

[75] Yuchen Zhou and David Evans. “SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities”. In: *Proceedings of the 23rd conference on USENIX security symposium*. USENIX Association. 2014.