

# **Abstract Semantics for Software Security Analysis**

## *Thesis proposal*

**Kevin Zhijie Chen**

Computer Science Division, EECS  
University of California, Berkeley

kevinchn@cs.berkeley.edu

October 7, 2014

## **Abstract**

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 abstraction and algorithm 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, and 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 many formal techniques applicable in a large scale.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Completed Work: Contextual Policy Enforcement in Android Applications with Permission Event Graphs</b>	<b>2</b>
2.1	Background . . . . .	4
2.2	Overview and Running Example . . . . .	5
<b>3</b>	<b>Work-in-progress: Iterative Security Analysis of Web Protocols using Synthesized Models</b>	<b>10</b>
3.1	Introduction . . . . .	10
3.2	Background . . . . .	12
3.2.1	Web Security . . . . .	12
3.2.2	Program Synthesis . . . . .	14
3.2.3	Formal Verification . . . . .	15
3.3	System Overview . . . . .	15
3.3.1	Designing a Representation . . . . .	15
3.3.2	Algorithmic Components . . . . .	15
3.3.3	Architecture and Exploration . . . . .	16
3.4	The MDL Language . . . . .	17
3.5	Synthesizing MDL . . . . .	19
3.5.1	Data Structure . . . . .	19
3.5.2	TFG Generation . . . . .	20
3.5.3	Model Generation . . . . .	22
3.6	Verification and Feedback . . . . .	23
3.6.1	Verification . . . . .	23
3.6.2	User Feedback . . . . .	26
3.7	Implementation . . . . .	26
3.8	Evaluation . . . . .	26
3.8.1	NeedMyPassword.com . . . . .	26
3.8.2	The CAS Protocol . . . . .	29
3.8.3	Govtrack.us and Facebook Connect . . . . .	32
<b>4</b>	<b>Research Plan</b>	<b>33</b>
4.1	Discussion . . . . .	33
4.1.1	System Limitations . . . . .	33
4.1.2	Input Traces and Errors . . . . .	33
4.2	Future Directions . . . . .	33
4.3	Timeline for completion of the research . . . . .	34
<b>5</b>	<b>Additional Related work</b>	<b>34</b>
<b>A</b>	<b>Refereed Paper</b>	<b>39</b>

# 1 Introduction

Abstraction is one of the most powerful ideas invented by the modern human. By thinking abstractly, we derive general rules and concepts from the tedious details and reason about the commonalities more efficiently. In computer science, abstraction is the process of using computational or statistical structures to represent the abstract semantics of the software or the hardware. Depending on the goal, a system can have several abstraction layers, exposing different aspects and amounts of details. 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 construction algorithms that expose the program properties in concern. In my thesis, I will introduce two novel abstractions that reveals two different aspects of the program behaviors, and the algorithms to construct these abstractions. 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 half 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.

## 2 Completed Work: Contextual Policy Enforcement in Android Applications with Permission Event Graphs

In this section, we motivate the problem of enforcing temporal policies on Android applications, and summarize our work on using permission event graphs to detect complex malicious behaviors. A more detailed technical report of this work can be found in the appendix.

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 [1–3].

In this section, 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 and 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 [2, 4, 5].

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 the 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.

**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 artifact, such as a button, a system artifact, 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 EX-PSPACE-hard [6]. 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 modeling 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.

In this work, 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.

## 2.1 Background

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*.



Figure 1: User interface for the running example. The application records audio in a background service after the user has clicked the stop button.

**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 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 work, 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.

## 2.2 Overview and Running Example

We now demonstrate the concepts in this section 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 1. This interface is implemented as a `Recorder` activity and contains two buttons, REC and STOP.

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.

$$\begin{aligned} & (\neg \text{Start-Recording} \text{ U } \text{REC.onClick}) \\ & \wedge (\text{Stop-Recording} \iff \text{STOP.onClick}) \end{aligned}$$

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.

$$\begin{aligned} & \neg \text{Send-SMS} \text{ U } \text{Button.onClick} \\ & \neg \text{BroadcastAbort} \end{aligned}$$

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 depicts the permissions requested by the recorder during installation. Techniques that only examine permission requests [5, 7, 8] will only know that



Figure 2: Permissions requested by the recording application during installation.

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 [1, 9], 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 specific button causes the `READ_CONTACTS` permission to be used, while the `ACCESS_FINE_LOCATION` permission is used in a background task.

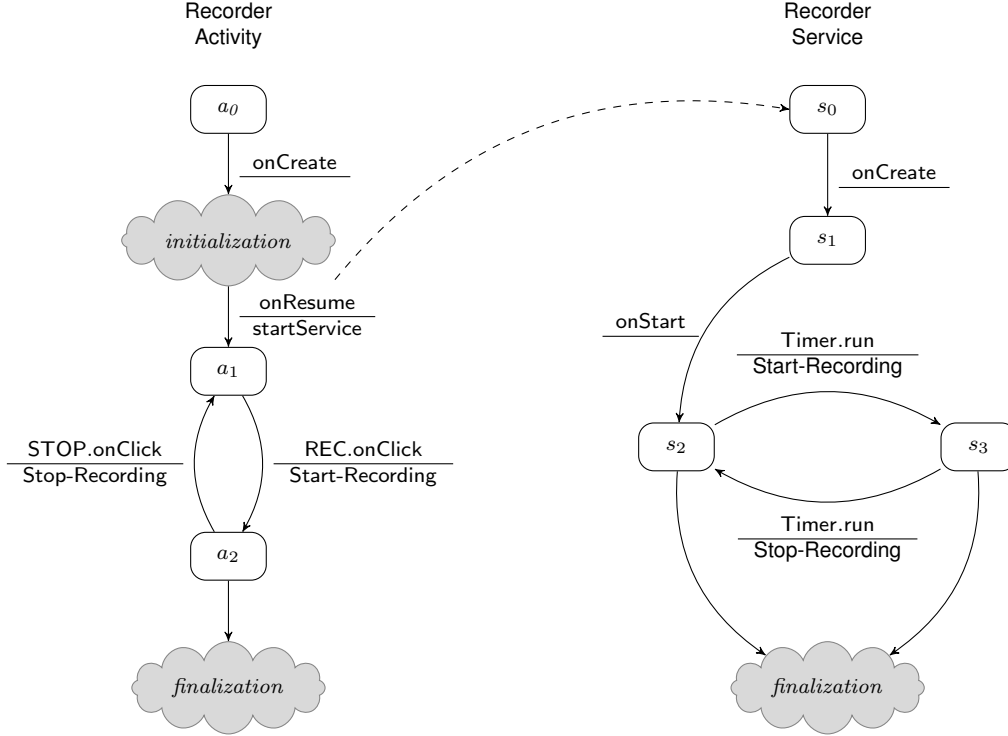


Figure 3: Permission Event Graph for the running example. Vertices represent event contexts and an edge label  $\frac{E}{A}$  represents that when the event-handler  $E$  fires, an action  $A$  is performed. Dashed edge represent asynchronous tasks.

A portion of the PEG for the running example is shown in Figure 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 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 initialization, destruction, and the service lifecycle. 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.

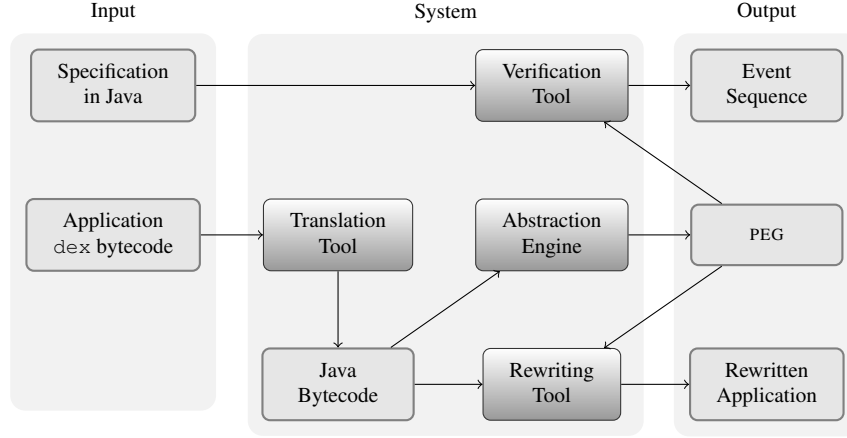


Figure 4: Pegasus architecture. The system consists of a translation and a verification tool, and an abstraction engine.

**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 summaries, 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.

### 3 Work-in-progress: Iterative Security Analysis of Web Protocols using Synthesized Models

In this section, I will introduce in more details of a work-in-progress direction that will form the other half of my dissertation.

#### 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 [10, 11], 2011 [12], 2012 [13, 14], 2013 [15] and 2014 [16].

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 [11, 15, 17, 18] 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 [19–21] 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 work, 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 [22] 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 [23]) 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 [24–26], and provides a more general framework thanks to synthesis techniques (See Section 5). We note that while synthesis from examples has been explored in some other domains [22], 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 [27–30], 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 three real world web applications. In all three instances, WEBSYN was able to find session integrity 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, *session integrity* vulnerabilities, including previously unknown vulnerabilities.

## 3.2 Background

### 3.2.1 Web Security

**Running Example** We use the Bodgeit Store application [31] 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 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. Messages are abstractly represented as sets of *tokens*. 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` parameter in the cookie identifies the shopping activity session. In addition, we also consider that the `username` and `password` parameters 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.

---

**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 1: Example traces from the Bodgeit Store



**Threat Model** We consider the web attacker model and session integrity formally defined in Akhawe et al. [11]. A *web attacker* is a malicious principle who controls a web server visited by the user. Intuitively, the web attacker can be thought 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 he or she can only respond to HTTP messages directed at his or her own servers.

Typical attacks on web protocols violate the *session integrity*. 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 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 defaults to searching for session integrity violations like the ones above, but, as we will show in our case studies, a key feature that makes our tool useful is allowing the analyst to modify the session integrity condition by excluding “uninteresting” attacks.

### 3.2.2 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 5 for a discussion on related work, and Gulwani et al. [22] 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.

### 3.2.3 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 [20, 21]. Researchers have also leveraged formal tools to analyze deployed systems, resulting in the discovery of real, exploitable vulnerabilities [12, 17, 24].

In our implementation, we use ALLOY [23] 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.

### 3.3.1 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 7).

### 3.3.2 Algorithmic Components

**Program Synthesis** The first algorithmic problem is to construct a protocol model from HTTP traces, e.g., the one in Listing 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.

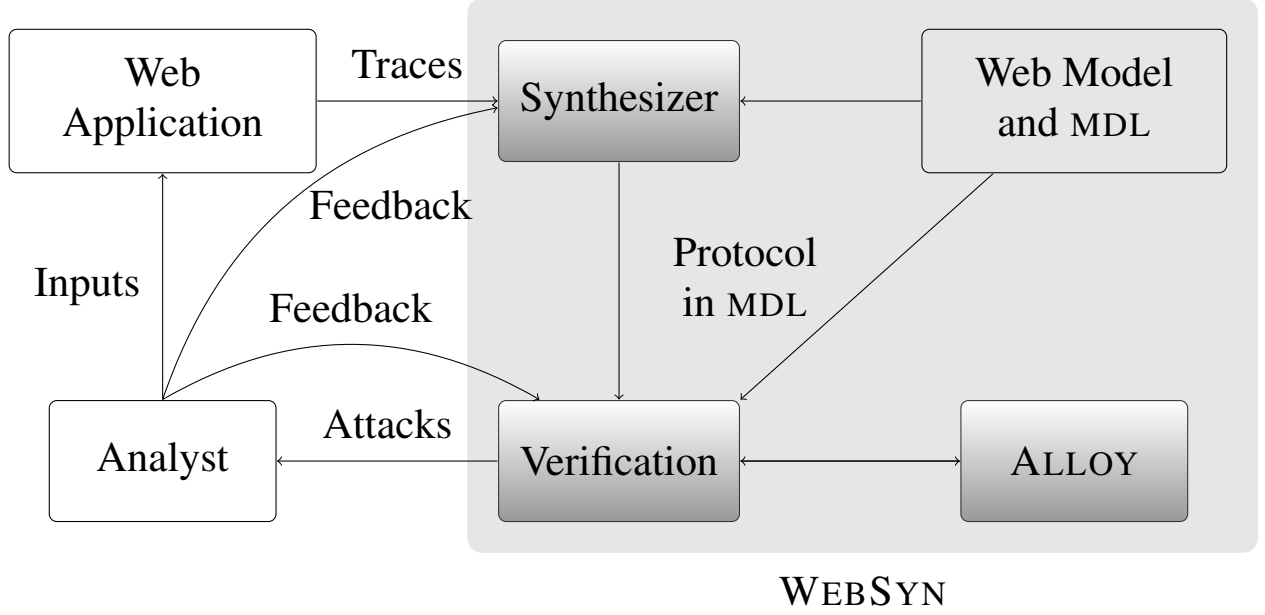


Figure 5: System Architecture: WEBSYN consists of a synthesizer and a verification component. It takes web application execution traces as input, and uses the synthesizer to generate protocol models. The verification component compiles protocol model and vulnerability templates into ALLOY models for 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.

**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 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 3.6.1 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 3.6.2 and demonstrate their effects on the search space in Section 3.8.

### 3.3.3 Architecture and Exploration

The architecture of WEBSYN is shown in context in Figure 5. 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 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 introducing the syntax of the MDL language, and then we use the Bodgeit Store example to informally describe the semantics of MDL.

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 6 for the syntax of MDL and Figure 7 for the protocol model of the Bodgeit Store login protocol in MDL.

```

specdecl := (spec epsdecl tokendekl logicdecl)
epsdecl := (eps {endpoint})
endpoint := string
tokendekl := (tks {(token {endpoint})})
token := string
logicdecl := (io {(endpoint msgdecl msgdecl)})
msgdecl := (msgname {(fieldname fieldtype valdecl)})
fieldtype := urlp | pbody | cookie | body |
             locparam | setcookie
msgname := string
fieldname := string
valdecl := selectdecl | newdecl
selectdecl := (select msgname fieldname {(fieldname fieldname)})
newdecl := (new token)

```

Figure 6: The extended BNF syntax of the MDL language.

The MDL syntax is lisp-like. A MDL program consists of three top-level expressions: the endpoint declaration (`epsdecl` in Figure 6), the token declaration (`tokendekl`), and the logic declaration (`logicdecl`). The endpoint declaration lists the participating server endpoints. In the Bodgeit Store example, there is only one server endpoint (`bodgeit`) as shown in Line 2 of Figure 7. The token declaration lists all the tokens and for each token the endpoints that know it at the beginning of the protocol. Our running example involves 4 tokens `t1`–`t4`, as shown in Line 3 of Figure 7. The first two are initially known by the server (i.e., `bodgeit`) and the other two are initially know by the client (i.e., `UA`). The logic declaration defines how each endpoint constructs responses according to the received requests. It is the core of the model. Once the logic declaration has been synthesized, the endpoint declaration and the token declaration can be trivially constructed. We explain the subexpressions in a logic declaration in the following paragraphs.

A logic declaration consists of a list of (`endpoint`, `msgdecl`, `msgdecl`) tuples, where each tuple specifies how the server endpoint `endpoint` constructs the response when it receives

---

```

1 (spec
2  (eps bodgeit)
3  (tks (t1 bodgeit) (t2 bodgeit) (t3 UA) (t4 UA))
4  (io
5    (bodgeit
6      (req-helo)
7      (resp-helo
8        (jsid setcookie (new t1))
9        (csrf body (new t2))))
10   (bodgeit
11     (req-login
12       (rcsrf urlp
13         (select resp-helo csrf
14           (csrf rcsrf)
15           (jsid rjsid)))
16       (username urlp (new t3))
17       (password urlp (new t4))
18       (rjsid cookie
19         (select resp-helo jsid
20           (csrf rcsrf)
21           (jsid rjsid)))
22     )
23   (resp-login))))

```

---

Figure 7: The MDL protocol model for the Bodgeit Store. Note that we have changed the variable names from the original randomly generated ones to more meaningful ones for the ease of understanding.

a request. Both the request and the response are specified by a `msgdecl` expression, which is a template of an HTTP message with symbolic “holes”. In the MDL language, these “holes” are called fields, and a message declaration is essentially a list of field declarations (syntactically a list of `(fieldname fieldtype valdecl)` tuples). Intuitively, when a new request arrives, the server will match it with the list of template requests in the logic declaration. To do this, the server needs to parse the concrete request into a list of `(fieldname, fieldtype, value)` tuples, and finds a template request which has the same set of fields. Once the corresponding template request is found, the server performs further validation and token-binding on the concrete value according to the `valdecl` expression. More specifically, if the `valdecl` is a `select` expression, the server will search through the history messages it has received, get the value of a history field according to the `select` constraints, and make sure it is equal to the current concrete value (otherwise the request will be rejected by the server). If the `valdecl` is a `new` expression, the server simply binds the value of this field to a specific token declared in `tokendcl`. The binding simply builds a correlation between the symbolic token name and the concrete value. After the validation and token-binding, the server constructs a concrete response according to the template response. The template response uses the same `msgdecl` expression as the template

request, with slightly different semantics: Instead of validating and binding the fields, we use the `select` expression to select a value from history and assign it to the field, and the new expression to generate a new value for the field. For example, Lines 5-9 in Figure 7 specifies that the `bodgeit` server sends a `resp-helo` response when it receives a `req-helo` request. Line 8 in Figure 7 specifies that the `resp-helo` response's `jsid` field should be constructed using a newly generated `t1` token, and Lines 12-15 specifies that the `rcsrf` field of request `req-login` should be equal to the `csrf` field of some previously seen `resp-helo` message.

As an example use of the `select` expression, Lines 13-15 in Figure 7 (`(select resp-helo csrf (csrf rcsrf) (jsid rjsid))`) refers to the `csrf` field of a previous `resp-helo` message, whose `csrf` field (and `jsid` field, resp.) is equal to the `rcsrf` field (`rjsid`, resp.) of the currently received request. Essentially, the values of `rcsrf` and `rjsid` are used as the identifiers to collect the relevant previous messages.

### 3.5 Synthesizing MDL

In this section, we present the algorithm to synthesize a MDL model. The algorithm consists of two phases, the token propagation phase and the model generation phase.

In the token propagation phase, the HTTP messages from the traces will be propagated into a set of tokens. This process serves two goals: First, we want to extract the tokens contained in each concrete message. For example, among the messages in Listing 1, we want to extract the value of the CSRF token embedded in the HTML response body, and also from the body of the POST request encoded as form data. There are also other types of encoding functions including nested or customized ones. Secondly, we want to discover the invariants between the tokens 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.

The model generation phase is responsible for determining the identifiers in a request, and discovering the invariants between the fields in the request-response pair and the fields from its previous messages. It translates the result of the token propagation phase to a model in our DSL.

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

#### 3.5.1 Data Structure

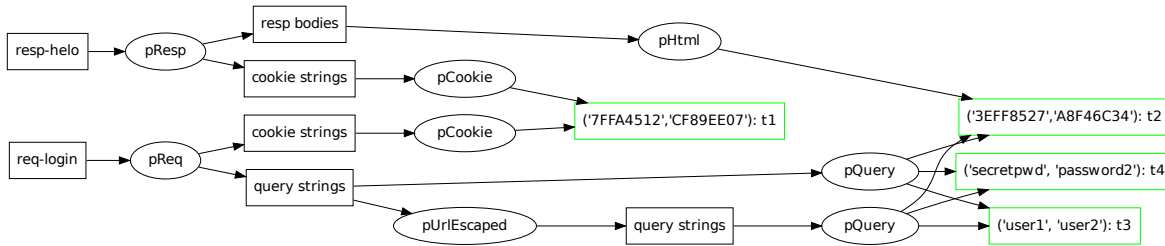


Figure 8: Example TFG for the Bodgeit Store

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

1.  $T$  is the set of all tokens. 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 tokens, where  $F$  is a set of token propagation functions defined in Table 1.
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 tokens, and  $a$ 's outgoing edges  $E_{out}(a) \subseteq A \times T$  connect  $a$  and its result tokens.

An example TFG for the Bodgeit Store is shown in Figure 8. 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 TFG. Such a TFG has the following properties:

1. The roots of the TFG is the set of the server's relevant messages, e.g., `resp-helo` and `req-login` in Figure 8.
2. The leaves of the TFG form the set of tokens known by the server at the end of the protocol execution. This token set is the tokens that will appear in the token declaration expression  $((\text{tks} \dots))$  of the model, e.g., the four leaf nodes in Figure 8.
3. The leaves reachable from a root message form the set of tokens carried by it. They determine the field declaration for the message, e.g., message `resp-helo` has two fields with values  $('7FFA4512', 'CF89EE07')$  and  $('3EFF8527', 'ABF46C34')$  from two traces.
4. The paths from a root to a leaf represent how a token can be extracted from a message, and the reverse paths indicate the way to construct a concrete message according to the tokens. They will be used to construct a concrete attack trace from an abstract one, e.g.,  $('user1', 'user2')$  is extracted by decoding the body of an HTTP response message as a URL query string.

Intuitively, the TFG 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. The structure of the TFG can also be viewed as a generalization of the concrete messages. Next, we are going to present how to automatically generate a TFG for each endpoint, and how to use it to synthesize the server logic.

### 3.5.2 TFG Generation

In the TFG generation phase, we generate a TFG for each endpoint in the protocol using the example messages. The graph generation algorithm is shown in Algorithm 1. We introduce its details in the following paragraphs.

We extract the messages from or to the same server and group the corresponding messages in the traces as a tuple, the result of which is:

$$M = [(s_1^1, s_1^2, \dots), (s_3^1, s_3^2, \dots), (s_4^1, s_4^2, \dots), \dots]$$

**Function** `computeTFG( $M$ )`

**Data:**  $M$ : List of messages from or to the same endpoint. Each message is represented as a tuple of the corresponding concrete message from each trace

**Result:**  $G$ : The token-function graph of  $M$ .

$finished = \emptyset$ ;

**forall the**  $m_i \in M$  **do**

$worklist.push(m_i, parseReq|Resp)$  ;

**end**

**while**  $worklist \neq \emptyset$  **do**

$t, f \leftarrow worklist.pop()$  ;

$T \leftarrow T \cup \{t\}$  ;

**if**  $(t, f) \notin finished \wedge f(t) \neq \emptyset$  **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:** Token-function graph generation

**Graph Generation** The generation algorithm 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 token node to be extended, and  $f$  is the function to be applied on the token. If the application is successful, a new application node  $(t, f)$  and a set of new token 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 token  $t'$  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 TFG 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 1. The choice of functions to try for each token determines the search space of the graph. The propagation algorithm choose different subsets of functions to try depending on where the token is from. The subset is determined by the functions that output this token. Column 3 of Table 1 lists which functions will be pushed to the worklist for each return value of a function. For example, the first row specifies that the algorithm will use `pCookie` to try parsing the `SetCookie` field returned by the `pHttpRequest` function.



No.	Function	Next functions
1	pHttpReq	(url:(6),cookie:(7),body:(3-11))
2	pHttpResp	(body:(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),fragment:(3-11))
7	pCookie	(value:(3-11))
8	pQuery	(value:(3-11))
9	pUrlEscaped	(orig:(3-8))
10	pPath	(item:())
11	pConcat	(item:())

Table 1: Parser function signatures. The numbers in the Column 3 refer to the function indices in Column 1.

### 3.5.3 Model Generation

In the model generation phase, we synthesize the different parts that form a valid model in MDL. First, we collect all the web origins in the trace to form the set of endpoints. Secondly, for each endpoint and each relevant request-response pair, we infer the corresponding ( $ep \ reqT \ respT$ ) expression in the  $io$  expression of MDL. Finally, we infer the initial knowledge set of each endpoint according to the  $io$  expression.

**Inferring the  $eps$  expression** The inference of the  $eps$  expression is trivially done by scanning through all the request messages and collect the hostnames of their URLs. In our running example, there is only one server involved, which is the Bodgeit Store website.

**Inferring the  $io$  expression** The inference of the  $io$  expression is divided into subtasks of inferring the set of fields for each message, including the types of the fields and where the values come from.

**Inferring the message fields** The set of fields for each message is defined by the set of reachable leaves for it in the TFG.

$$m.fields := TFG.leavesof(m)$$

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

**Inferring the field invariants** For each field in a request message, we need to determine if it is supposed to contain a previously unseen token or a known one, and for each field in a response message, we need to determine if the server needs to construct a new token or use a known one.

To achieve this, we first define the *relevant ancestor messages (RAM)* of a request as

$$RAM(req) := \{m \mid m < req \wedge m.fields \cap req.fields \neq \emptyset\}$$

The RAM of a request is the set of all history messages that can be identified by some fields in the request, and we use their fields  $RAM(req).fields$  and the fields of the request to form the set of all candidate tokens for constructing the response message. More specifically, we generate a token expression for each field in

$req.fields - RAM(req).fields$ , a new expression for each field in  $resp.field - RAM(req).fields - req.fields$ , and a select expression for each field in  $req.fields \cap RAM(req).fields$  and  $resp.fields \cap (RAM(req).fields \cup req.fields)$ . For example, from the Bodgeit Store’s TFG we find that both the cookie token ( $t1$ ) and the CSRF token ( $t2$ ) are shared between two messages. Thus  $RAM(req_{login}).fields = resp_{helo}.fields = (t1, t2)$ , and as a result, we generate select expressions for the respective fields for  $t1$  and  $t2$  in  $req_{login}$ , and new expressions for the respective fields for  $t3$  and  $t4$  in  $req_{login}$ .

**Inferring the initial knowledge sets** The initial knowledge sets can be trivially extracted from the `io` expression: For each endpoint, its initial knowledge set is the set of tokens that are used as the arguments of its token expressions. The initial knowledge set of the client is the set of tokens that are used as the arguments of all new expressions but not used as the argument of any token expression. In our running example, the username token  $t3$  and password token  $t4$  are initially owned by the user, and the cookie token  $t1$  and the CSRF token  $t2$  are initially owned by the Bodgeit Store. Note that there is a mistake here made by the synthesizer, since technically the username token and the password token should be initially owned by both the server and the client, i.e.  $(t3 \text{ UA})$  should be  $(t3 \text{ UA bodgeit})$ , 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.

### 3.6.1 Verification

The verification component takes the MDL model and embeds it into a general model of web-based distributed systems. 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 invariants. The invariants can be divided into 4 categories: the trace invariants, the endpoint invariants, the protocol invariants and the policy invariants.

**The trace invariants** The trace invariants define the basic structure of an HTTP trace. Some example trace invariants 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` instances and `Response` instances appear alternately in a `Trace`, a `Response` instance consists of URL parameters, set-cookie fields, a response body, and a redirection location, etc.

**The endpoint invariants** The endpoint invariants specify the constraints between two adjacent messages in the trace, i.e. how each endpoint reacts to an incoming message. Intuitively, these invariants 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 tokens.
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 elaborated as the protocol variants.

**The protocol invariants** The protocol invariants consist of protocol specific invariants 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 invariants** The policy invariants is translated from the initial vulnerability description, which is defined as the malicious server causing the benign client to send a critical request (cross site request forgery). The corresponding `ALLOY` predicate that checks if an HTTP request belongs to a CSRF attack is:

---

```

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 queryTokens[MalEP, e])
7 }
```

---

It basically says that an HTTP request is part of a CSRF attack if:

1. Its previous message is a redirection message from the attacker (Line 2).
2. It is sent by the benign client (Line 3).
3. It is sending to a benign server (Line 4).
4. Its non-cookie tokens are all known by the attacker (Line 5-6).

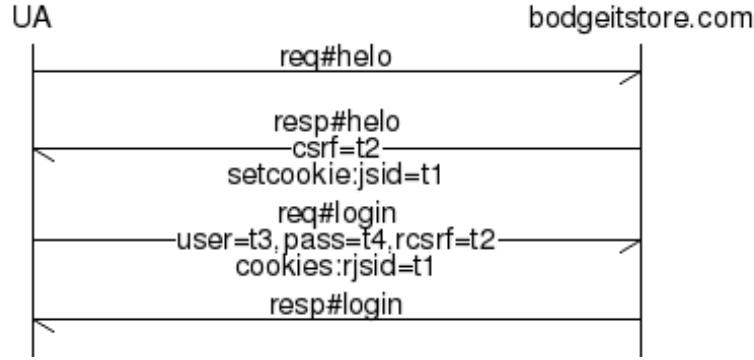


Figure 9: The message sequence chart illustrating the protocol synthesized for the Bodgeit Store.

**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 invariants. 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 vulnerability 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 9. 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.

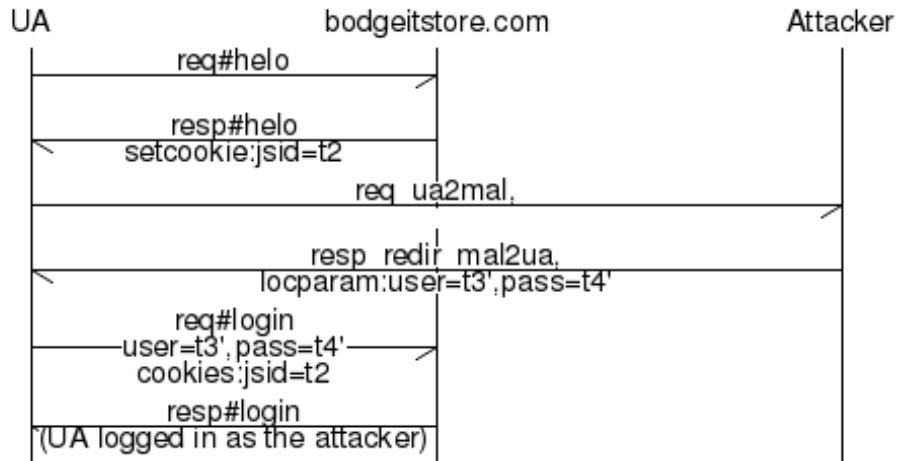


Figure 10: The message sequence chart for a login CSRF attack on the Bodgeit Store.

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 10.

### 3.6.2 User Feedback

In the user feedback component, the analyst inspects the results by reading the message sequence charts (e.g. Figure 9 and 10) 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.

1. Input hints, in which a new input value is provided as an alternative to the tokens that are previously constant, or an annotation is provided to mark one of the current tokens as unimportant.
2. Scope hints, which are binary 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.
3. Target hints, which are binary answers to whether one or more messages should be considered as a critical message.

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 [32]. We use the BrowserMob proxy [33] 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 [34] to generate HAR traces.

## 3.8 Evaluation

We used WEBSYN to identify vulnerabilities in three real world applications. We rediscover two (previously found manually) vulnerabilities and discover four previously unknown vulnerabilities. We summarize the configuration and the performance of each iteration in Table 2. 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.

### 3.8.1 NeedMyPassword.com

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

Name	Websites	New Hints	#Msgs	#Tokens	#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

Table 2: 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 token 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.

**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 demonstrate the process of logging in and adding a new record of username and password to the database.

The analyst also provides credentials for a second NeedMyPassword account (*account2* and *password2*). 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 12). 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.

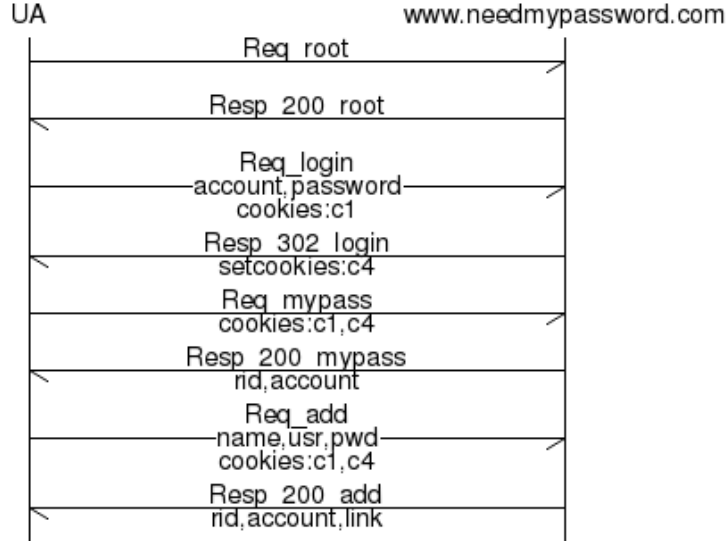


Figure 11: Synthesized protocol for NeedMyPassword.com

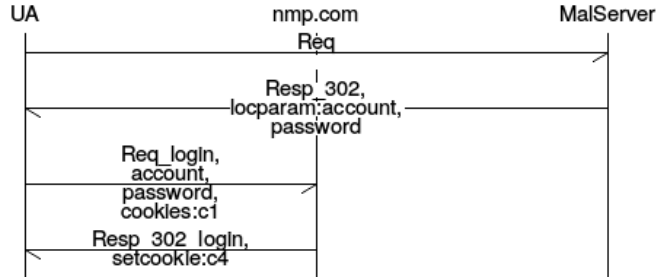


Figure 12: The session fixation attack for NeedMyPassword.com

**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

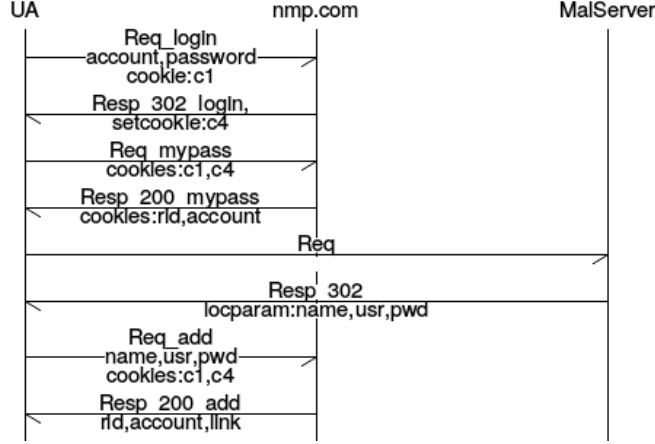


Figure 13: The second CSRF attack for NeedMyPassword.com

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 11 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 addition step too (Figure 13). 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 [35]. 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.

### 3.8.2 The CAS Protocol

Next, we analyze the Central Authentication Service protocol (CAS) [36]. The CAS protocol was originally developed at Yale University and at least eighty universities currently deploy it [37]. Akhawe et al. manually wrote down the protocol model and identified a session fixation attack (later fixed) [11].

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 14 shows the vulnerable protocol and we present its MDL model in the Appendix.



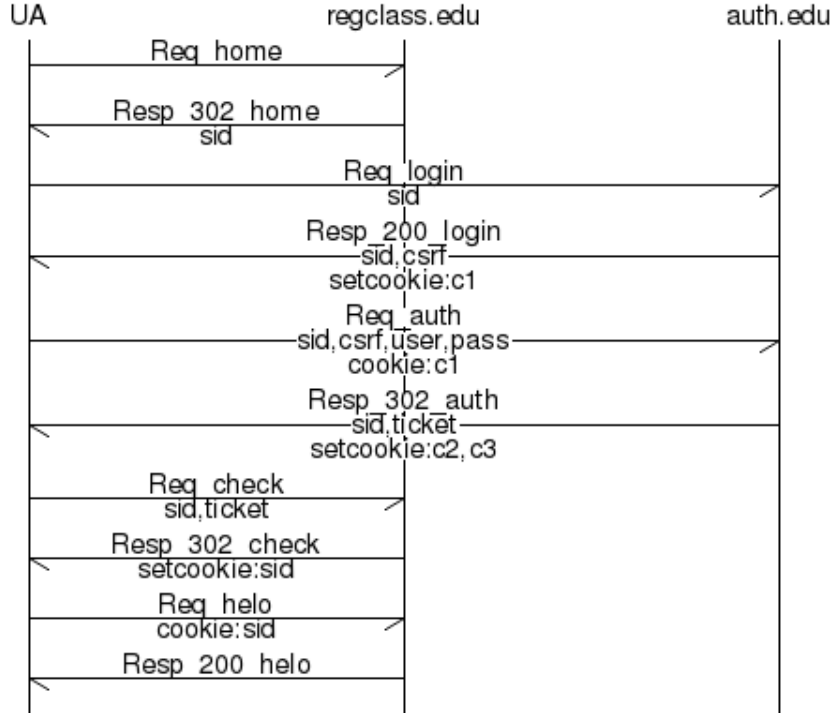


Figure 14: The vulnerable CAS protocol

**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 session-riding attack (Figure 15). Interestingly, this attack 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.

**Negative target hint** The verification on the vulnerable protocol with the modified attack condition returns an attack trace as shown in Figure 16. The attack trace basically says that the attacker 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 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 of the negative one above. We explicitly tell the model checker to find attack traces with the vulnerable message. After this change, the verification time reduces from 41.71 seconds to 8.63 seconds. Positive hints like these could be feasible for an analyst who is familiar with the protocol and the messages affecting critical resources or access.

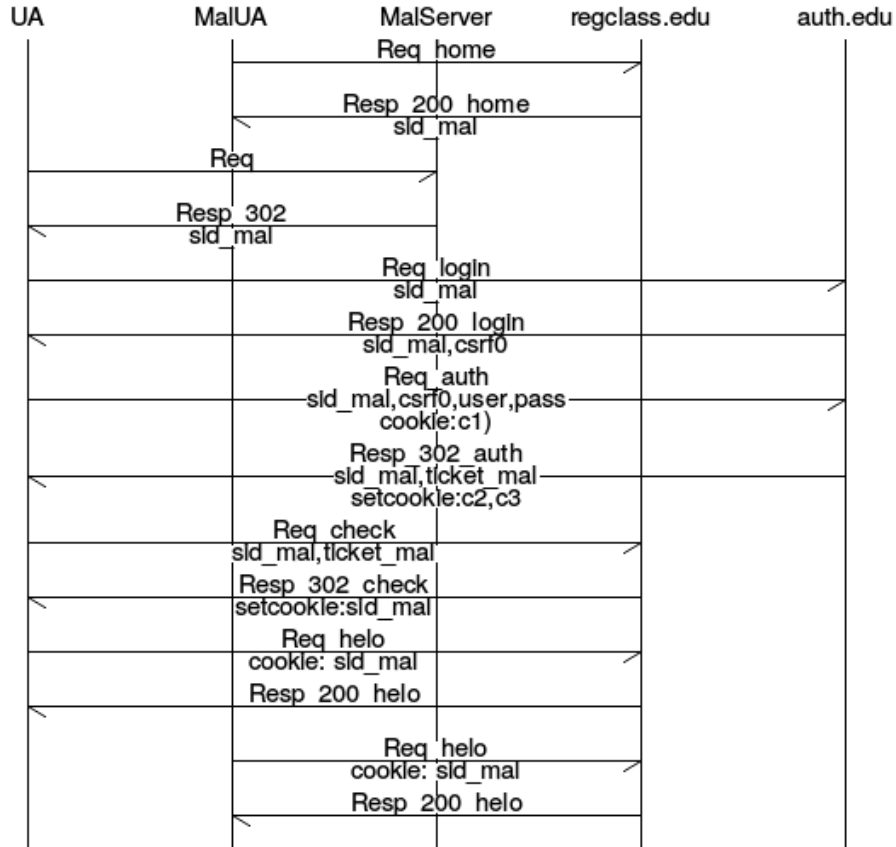


Figure 15: The first attack trace for the CAS protocol

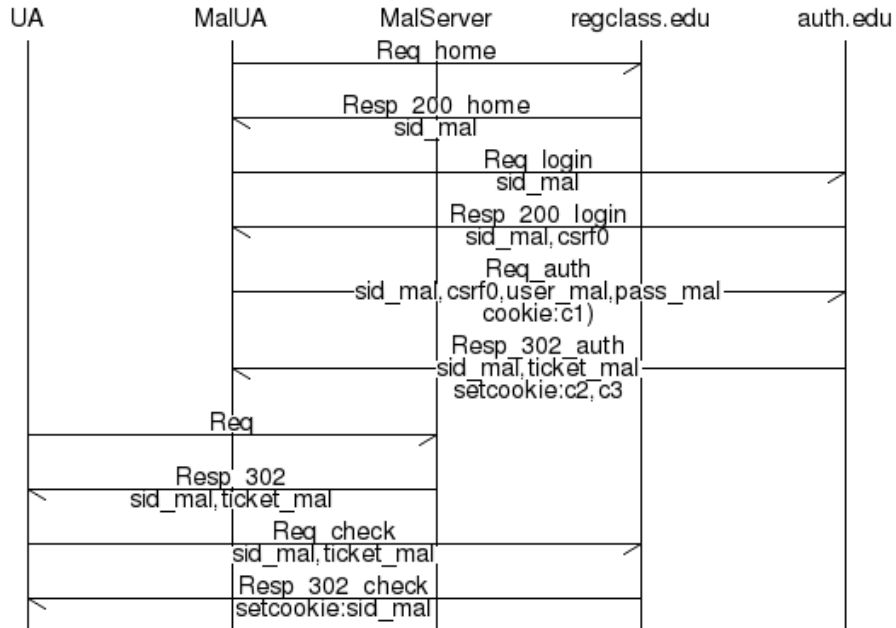


Figure 16: The second attack trace for the CAS protocol

**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.

### 3.8.3 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 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 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.

Both the attacks discovered in this case study are new, previously-unknown vulnerabilities, suggesting that WEBSYN can be a useful tool for protocol analysis.

## 4 Research Plan

In this section, we first discuss the limitations of our system and the robustness of the algorithm. The discussion serves as a motivation of the planned work introduced in the following sections.

### 4.1 Discussion

#### 4.1.1 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 deals with value differences, we assume there are no control flow differences at the level of our abstraction. 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 vulnerabilities. However, there is a large variety of attacks that fall in this category, and we are working on extending both the synthesizer and the security specification in order to incorporate other types of attacks.

#### 4.1.2 Input Traces and Errors

It is possible that analyst provided 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 [38], as well as machine learning techniques [39], 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.

### 4.2 Future Directions

Table 3 summarizes the proposed directions that we plan to work in the future, as well as the progress so far. We also introduce each direction in more details in the following paragraphs.

**Input Traces** The input traces provide the richest source of information for the synthesis algorithm. The current implementation performs static analysis on the HTML and JavaScript files transmitted between the server and the client. In the future, we plan to add support for dynamic client-side execution and client-to-client communication monitoring, in order to provide more precise behavioral information to the synthesizer.

**The DSL** The DSL represents the basic assumptions of the server logic model. It also determines the search space of the synthesis and verification. The current implementation of the DSL relies heavily on the low-level `select` expression in order to model server-side validation and token association. In the future we plan to add more high-level language constructs in order to reduce






Direction	Current implementation	Progress	Future plan
Input Traces	Static analysis		Dynamic analysis
DSL	Low-level constructs		High-level constructs
Synthesizer	Maximum validation		Partial validation
Verification	Minimum constraints		Domain-specific constraints
Evaluation	General CSRF and SSO		Password managers, payments, etc

Table 3: Progress and Plan Summary

the search space as well as model additional aspects of the server logic, such as the communication between the servers.

**The synthesizer** The current implementation of the synthesis algorithm assumes maximum validation, i.e., the server will try to validation all the tokens passed to it in each message. However, a lot of recent vulnerabilities show that the server often fails to validate all the tokens even through it has sufficient knowledge to perform the validation [40]. In the future, we plan to extend the synthesizer so that it also considers these cases and generate the server logic accordingly.

**The verification** The current implementation uses ALLOY as the underlying verifier and translates the specification in the DSL into ALLOY models. The preliminary evaluation shows that the current search space encoded by the translated model does not scale very well. In the future we plan to redesign the ALLOY modeling and use additional domain specific constraints in order to further reduce the search space.

**The Evaluation** In order to demonstrate the generality of our approach, we also plan to evaluation our system on a large variety of web applications including online payment services and password managers.

### 4.3 Timeline for completion of the research

Table 4 shows the timeline for completion of the research.

## 5 Additional 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 [40], single-sign-on web services [12] and protocol SDKs [15]. The latter also demonstrates a systematic process of the interaction between the security analyst and a

Deadline	Work	Progress
-	Contextual Policy Enforcement inn Android Applications with Permission Event Graphs	completed
Sep. 2014	Iterative Security Analysis of Web Protocols using Synthesized Models (Prototype)	completed
Mar. 2015	Extend the system based on the proposed directions above	ongoing
May 2015	Thesis writing	planned
May 2015	Dissertation talk	planned

Table 4: Timeline for completion of my research

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 [24] 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 AuthScans 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 [25] propose a blackbox technique that identifies a number of behavioral patterns from network traces and generates test cases. The patterns are expressed as a graph-based model, and are applicable only to the detection of logic vulnerabilities. Invariant detectors like Daikon [26] inductively generates invariants from traces. The basic constructs of the invariants are low-level operations such as arithmetic or boolean operations. Our approach is similar to the above work in that we also use an inductive approach to generate specification. But our DSL based models are more expressive than the graph model and at a higher semantic level than Daikon. The difference in the choices of the abstraction level reflects the difference of vulnerabilities each approach focuses on, and the different efficiency trade-offs according to the goals.

Lie et al. [41] proposed a method to extract specifications automatically from program code using program slicing. Aizatulin et al. [19] proposed model extraction using symbolic execution. SLAM [20] and CEGAR [21] 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 [29,42], system predicates over input-output [43,44], program template structure and constraints [45], natural language [46], and user demonstration [38]. Gulwani et al. provide a comprehensive overview of such techniques [22]. 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 [11] enables automatic security analysis of specifications; our work focuses on automatic *generation* of these specifications.

## References

- [1] W. Enck, “Defending users against smartphone apps: Techniques and future directions,” in *International Conference on Information Systems Security*, pp. 49–70, Springer, 2011.
- [2] W. Enck, D. Ocateau, P. McDaniel, and S. Chaudhuri, “A study of Android application security,” in *Proc. of the USENIX conference on Security*, pp. 21–21, USENIX Association, 2011.
- [3] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *Proc. of the workshop on Security and privacy in smartphones and mobile devices*, SPSM ’11, pp. 3–14, ACM, 2011.
- [4] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” in *Proc. of the ACM conference on Computer and communications security*, pp. 235–245, ACM, 2009.
- [5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in *Proc. of the Conferenc. on Computer and Communication Security*, pp. 627–638, ACM, 2011.
- [6] R. Jhala and R. Majumdar, “Interprocedural analysis of asynchronous programs,” in *Symposium on Principles of Programming Languages*, vol. 42 of *POPL*, pp. 339–350, ACM, Jan. 2007.
- [7] K. Au, Y. Zhou, Z. Huang, and D. Lie, “PScout: Analyzing the Android permission specification,” in *Proc. of the ACM Conference on Computer and Communications Security*, pp. 217–228, ACM, 2012.
- [8] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: attacks and defenses,” in *Proc. of the USENIX Security Conference*, USENIX Association, 2011.
- [9] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “RiskRanker: scalable and accurate zero-day Android malware detection,” in *Proc. of Mobile Systems, Applications, and Services*, pp. 281–294, ACM, 2012.
- [10] S. Hanna, R. Shin, D. Akhawe, P. Saxena, A. Boehm, and D. Song, “The emperor’s new APIs: On the (in) secure usage of new client-side primitives,” in *Proceedings of the Web*, vol. 2, 2010.
- [11] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, “Towards a formal foundation of web security,” in *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pp. 290–304, IEEE, 2010.
- [12] R. Wang, S. Chen, and X. 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*, pp. 365–379, IEEE, 2012.
- [13] S.-T. Sun and K. Beznosov, “The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems,” Aug 2012.
- [14] S.-T. Sun, K. Hawkey, and K. Beznosov, “Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures,” *Computers & Security*, 2012.
- [15] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, “Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization,” in *Proceedings of the USENIX Security Symposium*, USENIX, 2013.
- [16] Y. Zhou and D. Evans, “Sscan: Automated testing of web applications for single sign-on vulnerabilities,” in *Proceedings of the 23rd conference on USENIX security symposium*, USENIX Association, 2014.
- [17] C. Bansal, K. Bhargavan, and S. Maffei, “Discovering concrete attacks on website authorization by formal analysis,” in *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pp. 247–262, IEEE, 2012.
- [18] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffei, “Keys to the cloud: formal analysis and concrete attacks on encrypted web storage,” in *Principles of Security and Trust*, pp. 126–146, Springer, 2013.
- [19] M. Aizatulin, A. 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*, pp. 331–340, ACM, 2011.
- [20] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of C programs,” in *ACM SIGPLAN Notices*, vol. 36, pp. 203–213, ACM, 2001.



- [21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Computer aided verification*, pp. 154–169, Springer, 2000.
- [22] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD ’10, (Austin, TX), pp. 1–2, FMCAD Inc, 2010.
- [23] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.
- [24] G. Bai, J. Lei, G. Meng, S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. Dong, “Authscan: Automatic extraction of web authentication protocols from implementations,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.
- [25] G. Pellegrino and D. Balzarotti, “Toward black-box detection of logic flaws in web applications,” in *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, February 2014.
- [26] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.
- [27] A. Solar-Lezama, *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.
- [28] R. Singh and S. Gulwani, “Synthesizing number transformations from input-output examples,” in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV’12, (Berlin, Heidelberg), pp. 634–651, Springer-Verlag, 2012.
- [29] S. 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, (New York, NY, USA), pp. 317–330, ACM, 2011.
- [30] E. Torlak and R. 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*, pp. 135–152, ACM, 2013.
- [31] Psiinon, “The bodgeit store.” Online, 2010. <https://code.google.com/p/bodgeit/>.
- [32] “Ide and webdriver,” 2014. <http://seleniumhq.org/>.
- [33] Lightbody, “The browsermob proxy.” Online, 2014. <http://bmp.lightbody.net/>.
- [34] OWASP, “Testing tools.” Online, 2014. [https://www.owasp.org/index.php/Appendix\\_A:\\_Testing\\_Tools](https://www.owasp.org/index.php/Appendix_A:_Testing_Tools).
- [35] Z. Li, W. He, D. Akhawe, and D. Song, “The emperors new password manager: Security analysis of web-based password managers,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
- [36] D. Mazurek, “The CAS protocol.” Online, 2005. <http://www.jasig.org/cas/protocol>.
- [37] JASIG, “The CAS protocol deployment.” Online, 2010. <http://www.jasig.org/cas/deployments>.
- [38] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, “Programming by demonstration using version space algebra,” *Machine Learning*, vol. 53, no. 1-2, pp. 111–156, 2003.
- [39] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, “A machine learning framework for programming by example,” in *Proceedings of The 30th International Conference on Machine Learning*, pp. 187–195, 2013.
- [40] R. Wang, S. Chen, X. Wang, and S. Qadeer, “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*, pp. 465–480, IEEE, 2011.
- [41] D. Lie, A. Chou, D. Engler, and D. Dill, “A simple method for extracting models from protocol code,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 192–203, IEEE, 2001.
- [42] W. R. Harris and S. Gulwani, “Spreadsheet table transformations from examples,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’11, (New York, NY, USA), pp. 317–328, ACM, 2011.

- [43] S. Srivastava, S. Gulwani, and J. 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, (New York, NY, USA), pp. 313–326, ACM, 2010.
- [44] S. Gulwani, V. A. Korthikanti, and A. Tiwari, “Synthesizing geometry constructions,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’11, (New York, NY, USA), pp. 50–61, ACM, 2011.
- [45] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, “Path-based inductive synthesis for program inversion,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, (New York, NY, USA), pp. 492–503, ACM, 2011.
- [46] A. Cozzie and S. T. King, “Macho: Writing programs with natural language and examples,” Tech. Rep. 2142.33791, University of Illinois at Urbana-Champaign, 2012.

## A Refereed Paper

For the “Contextual Policy Enforcement in Android Applications with Permission Event Graphs” part, we attach the camera-ready version of our NDSS 2013 paper for further reference.

# Contextual Policy Enforcement in Android Applications with Permission Event Graphs

Kevin Zhijie Chen<sup>†</sup>, Noah Johnson<sup>†</sup>, Vijay D’Silva<sup>†</sup>, Shuaifu Dai<sup>†</sup>, Kyle MacNamara<sup>†</sup>, Tom Magrino<sup>†</sup>,  
Edward Wu<sup>†</sup>, Martin Rinard<sup>‡</sup>, and Dawn Song<sup>†</sup>

<sup>†</sup>University of California, Berkeley\*

<sup>‡</sup>Massachusetts Institute of Technology

## Abstract

*The difference between a malicious and a benign Android application can often be characterised by context and sequence in which certain permissions and APIs are used. We present a new technique for checking temporal properties of the interaction between an application and the Android event system. Our tool 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. Our work centres around a new abstraction of Android applications, called a Permission Event Graph, which we construct with static analysis, and query using model checking. We evaluate application-independent properties on 152 malicious and 117 benign applications, and application-specific properties on 8 benign and 9 malicious applications. In both cases, we can detect, or prove the absence of malicious behaviour beyond the reach of existing techniques.*

## 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 behaviour [12, 14, 19].

In this paper, we focus on detecting malicious behaviour that can be characterised by the temporal order in which an application uses APIs and permissions. Consider a malicious audio application and 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 [14, 15, 18]. 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 behaviour.

We present Pegasus, a system for specifying and automatically enforcing policies concerning API and permission use. Pegasus combines static analysis, model checking, and

---

\*The 4-7th authors are now at Peking University, UC Santa Barbara, Cornell University and University of Washington, Seattle, respectively. The work was done when the authors were in UC Berkeley.

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, which can be detected syntactically. Thus an attempt to evade our system may only result in drawing greater scrutiny to the application.

## 1.1 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 behaviour 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 behaviour.

**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 behaviour 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 [25]. 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 behaviour, 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 analyses 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.

## 1.2 Content and Contributions

In this paper, we study the problem of detecting malicious behaviour 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 behaviour 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 behaviour, which was previously discovered by manual analysis.

The paper is organised as follows: We summarise background on Android and introduce our running example in Section 2. PEGs are formally defined in Section 3 and can be constructed using the algorithm in Section 4. The details of our system appear in Section 5, followed by our evaluation in Section 6. We conclude in Section 8 after discussing related work in Section 7.

## 2 Background and Overview

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

### 2.1 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 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 paper, 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.

### 2.2 Overview and Running Example

We now demonstrate the concepts in this paper with a running example, as well as how we envision the system being used. Consider a malicious audio recording applica-



**Figure 1.** User interface for the running example. The application records audio in a background service after the user has clicked the stop button.

tion, which eavesdrops on the user. On startup, the application displays the interface shown in Figure 1. This interface is implemented as a *Recorder activity* and contains two buttons, REC and STOP.

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 behaviour such as surreptitious recording? How can we automatically detect such behaviour?

**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.

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

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,



**Figure 2.** Permissions requested by the recording application during installation.

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.

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

The three formulae 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 formulae. 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 depicts the permissions requested by the recorder during installation. Techniques that only examine permission requests [1, 18, 20] 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 [12, 23], will not identify the behaviours 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 analysed 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 specific 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 3. Every vertex represents an event context. There are two types of edges. Solid edges represent synchronous behaviour, and dashed edges represent asynchronous behaviour. 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 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 lifecycle. 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 behaviour: 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.

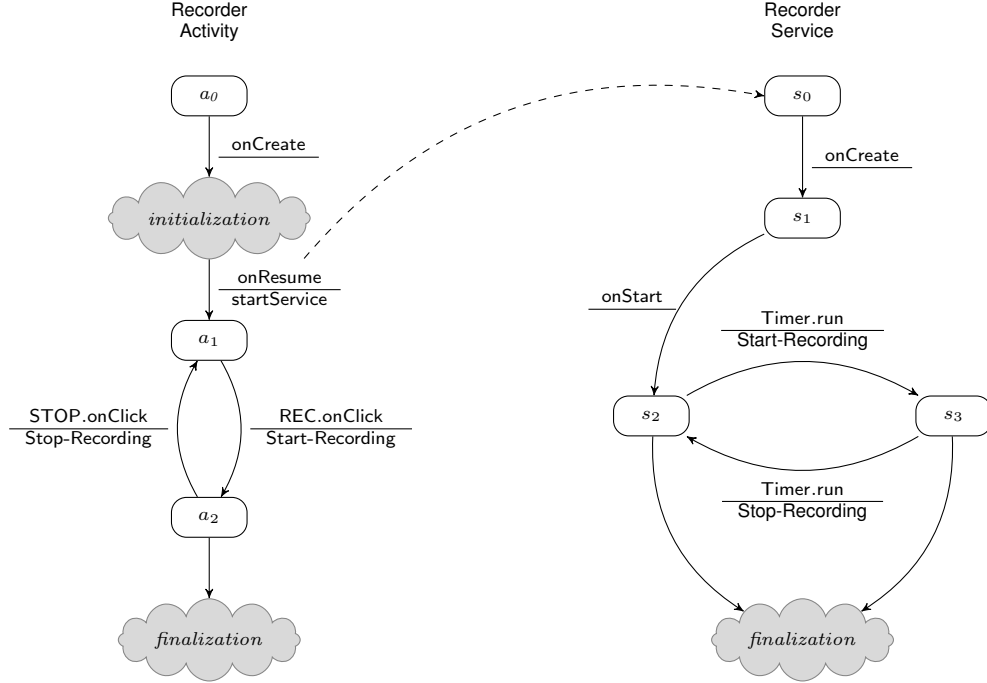
There are several kinds of properties that an analyst can check. We have developed a cookbook of application-independent properties, such as background audio or video recording. An analyst can write application-specific properties to check that an application functions as expected. For example, clicking `REC` should start recording, and `STOP` should stop recording. An analyst can also pose questions about the behaviour 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 behaviour. If the behaviour is potentially malicious, the analyst will have to reproduce it at runtime. If the behaviour is benign, the analyst will have to strengthen the property that is checked to narrow the search for malicious behaviour.

To summarise, a vocabulary based on events and actions allows for describing a new family of benign and malicious behaviour 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 summarises 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.

### 3 An Abstraction of Android Applications

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



**Figure 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.

### 3.1 Transition Systems from Android Apps

An Android application defines an infinite-state transition system, which describes the runtime behaviour 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

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

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

$$Trans \subseteq State \times State$$

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

$$T = (State, Trans, Init, stmt)$$

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



### 3.2 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 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.  $\triangleleft$

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  $aTrans \subseteq aState \times Event \times \mathcal{P}(API) \times aState$ , 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  $bTrans \subseteq \mathcal{P}(API) \times aState$ , 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 analyse PEGs. To derive the PEGs of an application efficiently, our abstraction engine uses the symbolic encoding introduced next.

### 3.3 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

$$s : ModeVars \rightarrow \{\text{true}, \text{false}\}$$

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  $\varphi$  over *ModeVars* represents the set of Boolean encodings that make  $\varphi$  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 [2].

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

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

consisting of a *precondition* *Pre*, a set of API labels *A*, and a *postcondition* *Post*. An event-modality  $(Pre, A, Post)$  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*. Notice that events are not part of an event-modality.

Event-modalities encode abstract states and actions. We also have to encode events. An *event-map* is a function

$$event-map : Handler \rightarrow Cube$$

that maps each event-handler to a cube representing the set of abstract states in which that event-handler may fire. A *symbolic encoding* of a PEG is a tuple

$$sPEG = (aInit, EventModality, event-map)$$

consisting of a cube  $aInit$  representing initial states, a set  $EventModality$  of event-modalities, and an event-map.

**Example 2.** Consider a Boolean variable  $TimerEnabled$  and a label Start-Recording. The timer-related behaviour in Figure 3 can be encoded using the value true for  $TimerEnabled$  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.

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

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  $\neg TimerEnabled$ .  $\triangleleft$

## 4 The Abstraction Engine

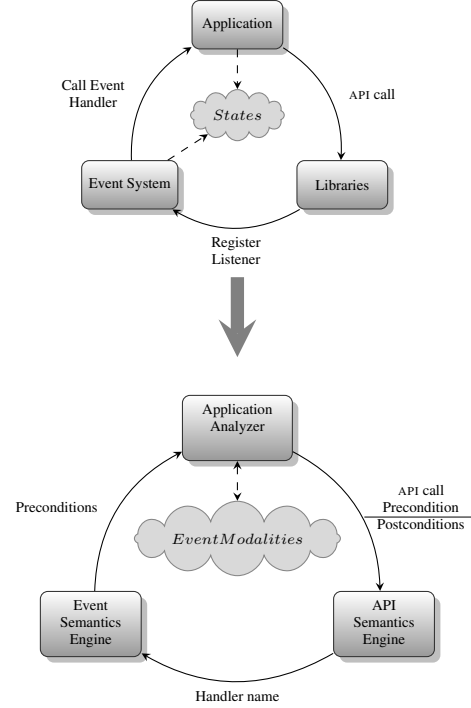
The contribution of this section is a procedure and architecture for constructing PEGs from Android applications. Our implementation of this procedure combines a model of the Android event system and APIs with fixed point iteration in a lattice to derive PEGs.

### 4.1 The Core Algorithm

The interaction between an application, the event mechanism and libraries in an Android application is summarised in the upper part of Figure 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 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 analyser, model event generation and destruction with an *event semantics engine*, and model APIs with an *API semantics engine*.

Our static analyser 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 analyser. If the API modifies the system state, the name of the API is given to the



**Figure 4.** Intuition behind the abstraction engine. The system computes event modalities by combining a static analyser, which abstracts application semantics, an API semantics engine, which abstracts API calls, and an event semantics engine, which abstracts the event system.

event semantics engine. The event semantics engine computes the set of preconditions for that event handler to fire, and the static analyser had to determine whether to analyse 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 formalise these components.

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

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  $next(h, A)$  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 analyser, 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

$$\begin{aligned} reach-sem_h : \mathcal{P}(\mathcal{P}(\text{API}) \times \text{API} \times \text{Cube}) \\ \rightarrow \mathcal{P}(\mathcal{P}(\text{API}) \times \text{API} \times \text{Cube}) \end{aligned}$$

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.

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

Note that *reach-sem<sub>h</sub>* 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 *reach-sem<sub>h</sub>* and projecting out the action, postcondition pairs that reach the exit point of the event handler. Formally, *app-sem* satisfies the condition below.

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

A pair  $(A, q)$  is produced by *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:

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

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.

## 4.2 Implementation of the Engines

A contribution we make, en route to computing PEGs, is to engineer a static analyser, 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 [5, 21, 30].

**Event Semantics Engine.** The event semantics engine implements the *event-sem* function. It receives a method han-

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

**Table 1.** Event handling APIs supported by the event semantics engine.

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 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* analyse the Android API source code. Instead, we model every API call we have found necessary to support during analysis. Figure 5 summarises 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.

## 5 Pegasus

We design and implement Pegasus, an analysis system that combines the abstraction procedure in Section 4 with

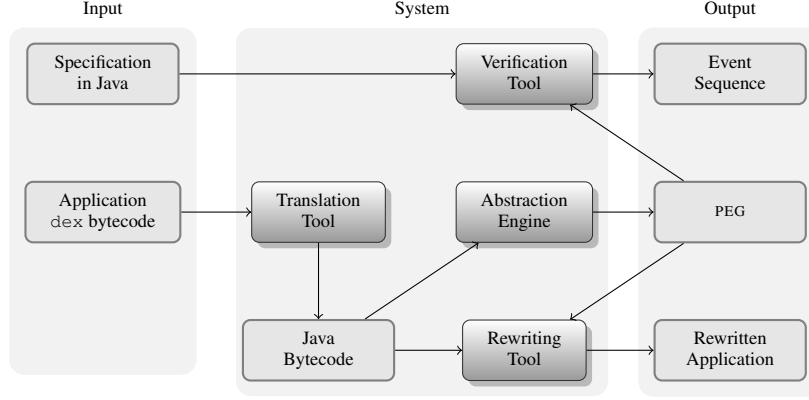
analysis of PEGs and rewriting of Android applications.

**System Overview.** Figure 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.

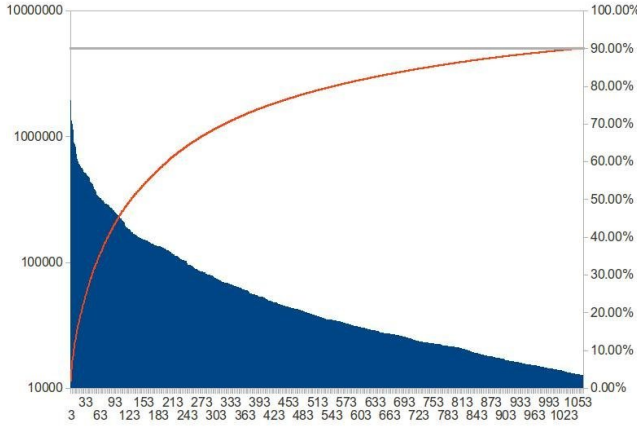
The *abstraction engine* takes as input Java bytecode and generates an PEG as output. The PEG is fed to the *verification tool*, along with a specification to check for conformance. If certain application behaviour cannot be analysed (for instance, due to unresolved reflection), the *rewriting tool* 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 semantics 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 behaviours never occur. Researchers have



**Figure 6.** Pegasus architecture. The system consists of a translation and a verification tool, and an abstraction engine.



**Figure 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.

```

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 7.** Specification for the running example.

developed a numerous languages for safety properties. The SLAM project used a C-like specification language called SLIC [3], because it was convenient to use specification language with similar syntax to the analysed programs. Similarly, we write specification monitors in Java.

A specification for the running example is shown in Figure 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 behaviours associated with the current event modality. If it finds the `RecordStart` behaviour 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 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 [1, 18], 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 analyse 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 analyses may be unable to precisely analyse programs that use them. We rewriting applications to include runtime checks to account for cases where static analysis does not succeed.

When the abstraction engine fails to analyse 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 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 analysed statically. This occurs when we believe the behaviour that was not statically analysed is benign. Such manually aided rewriting allows us to reduce the overhead of runtime checks.

In Section 6 we evaluate the number of unresolved transitions in the applications we analysed. 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 [31], a compiler and static analysis framework for Java bytecode.

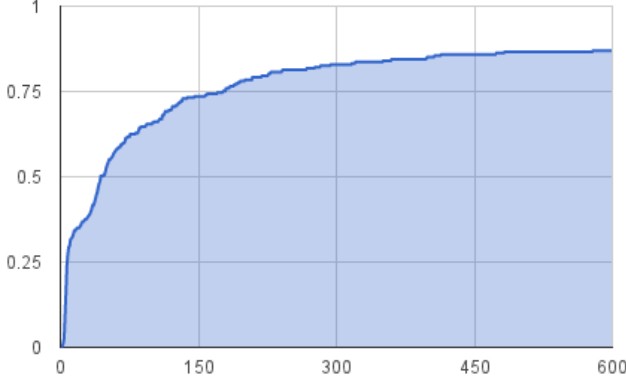
## 6 Evaluation

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.

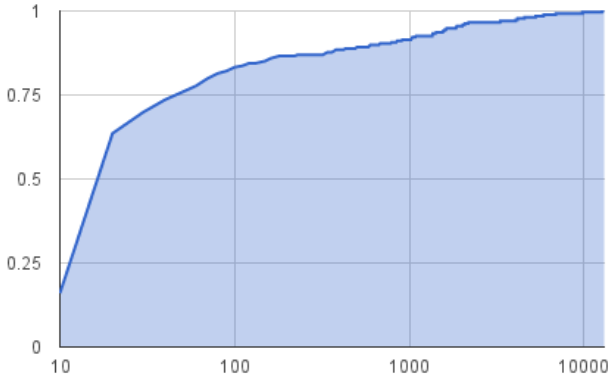
### 6.1 Generic Specification Checking

We run Pegasus on 152 malicious and 117 benign Android applications, and measure the execution and the size of the PEG. Figure 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 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 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 behaviours above are always bracketed by user interaction, such as a button click. The result of verification is shown in Table 2. We see that malicious applications performing sensitive operations without user consent more frequently than benign applications.



**Figure 8.** CDF of abstraction time.

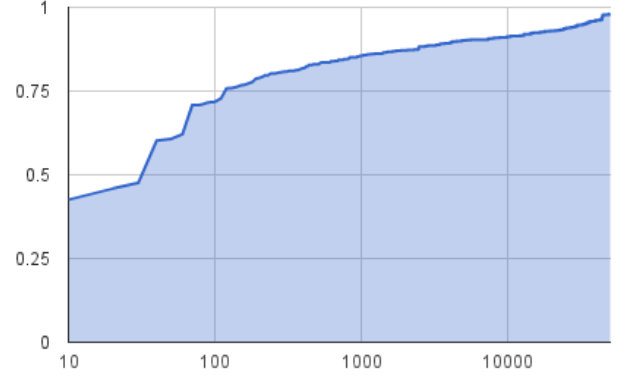


**Figure 9.** CDF of the verification time.

## 6.2 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 behaviours, to demonstrate Pegasus used as a diagnostic tool. Table 4 in Appendix A 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 application classes that together exercise most of the core functionality supported in the Android system. These applications implement a variety of behaviours 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 behaviours.

**Specifications.** For these applications, we constructed application-specific properties after installing each application, reading its documentation to understand its intended



**Figure 10.** CDF of PEG size measured by the number of states.

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

**Table 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.

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 3 summarises the results of PEG generation. The last two columns of Table 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 the counterexample trace to determine if the violation represented a feasible behaviour 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.

Name	Size KB	EM #	Intents		Reflection	
			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 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 properties we checked are discussed in detail in Section 6.3. 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 behaviour. 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.

### 6.3 Case Studies

We now present case studies illustrating properties one can check with Pegasus. Table 5 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 formulae. Note that Pegasus does not actually take LTL formulae 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 formulae 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 [9] for an in-depth discussion of such issues.

#### 6.3.1 Benign Applications

**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

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

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

$$\neg \text{Access-GPS } \mathbf{U} \left( \bigvee \begin{array}{l} \text{Locate.onClick} \\ \text{AdTask.onPreExecute} \end{array} \right)$$



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 that contacts information is only accessed after a phone call is received.

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

Pegasus returns a counterexample which shows that the application retrieves and caches the contact list when it starts up. We verified this behaviour 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.

$\neg \text{Send-SMS} \cup \text{Access-Contacts}$

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

$\neg \text{Send-SMS} \cup \text{Send.onClick}$

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

$\neg \text{Insert-Contacts} \cup \text{Insert.onClick}$

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.

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

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.

$\neg \text{Send-SMS} \cup \text{Receive-SMS}$

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.

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

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 behaviour is innocuous, so we refined the property as below.

$\neg \text{Access-Contacts} \cup \left( \bigvee \begin{array}{l} \text{Result.onClick} \\ \text{Sync.onClick} \end{array} \right)$

The application satisfies the refined property.

### 6.3.2 Malicious Applications

**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 a timer can trigger recording.

**ZitMo.** 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.** We checked two properties of this application.

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

$\neg \text{BroadcastAbort}$

Both properties are violated, because the application maliciously 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 proper-

ties.

$\neg$ Kill-Background-Processes  
 $\neg$ Read-IMEI  
 $\neg$ 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.

## 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 paper. We only attempt to place our work in the context of either seminal or very recent papers 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 [32] for a hardware-oriented survey of industrial formats for temporal logics, and [6] 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 [17]. Monitoring for security policies has been implemented in EFSA [29] and PSLang [16], and with a focus on mobile security. In the S3MS project [10], The Apex [27] 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 [22]. Techniques from JAM can be used to improve our rewriting tool.

**Static Analysis.** The design of our abstraction engine combines ideas from abstract interpretation [7] and software model checking. The closest related work is the SLAM toolkit [2], 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 have been extended to *lazy abstraction* [24], which interleaves the abstraction and checking process, and to YOGI [4], which combines testing and theorem proving to construct abstractions. Most developments that follow SLAM, such as those surveyed in [26] 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 [25]. 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 [11]. These are obstacles that will have to overcome if we wish to improve our analysis.

**Android Security.** TaintDroid [13] 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 [14] to detect a range of malicious behaviour. A semantically richer static analysis has been applied to Android in [28], but the focus is on common bugs rather than security properties.

The Stowaway system [18] combines static analysis with a permission map to identify applications requesting more permissions than they use. Permission-based approaches [1, 15, 27] use a map from API calls to Android permissions. Pegasus uses 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.

**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 behaviour.

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 behaviour. 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 applica-

tions — thereby making the code more conspicuous. Thus, much as the ZOZZLE defense against heap spraying attacks in Javascript [8], 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.

## 8 Conclusion

We have presented a new approach to specifying and detecting malicious behaviour 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 detect and prevent security specifications that characterises safe interaction 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 analyse 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.

## 9 Acknowledgements

We are deeply indebted to Úlfar Erlingsson for his assistance, feedback and encouraging support. We thank the anonymous reviewers for their comments.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0842695, No. 0831501 and the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

- [1] K. Au, Y. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proc. of the*

- ACM Conference on Computer and Communications Security*, pages 217–228. ACM, 2012.
- [2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the Symposium on Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
- [3] T. Ball and S. K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
- [4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 3–14. ACM, 2008.
- [5] 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)*, pages 391–405, 2008.
- [6] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proc. of the International Symposium on Formal Methods for Components and Objects*, pages 342–363. Springer, 2005.
- [7] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.
- [8] C. Cutsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: fast and precise in-browser JavaScript malware detection. In *Proc. of the USENIX conference on Security*, pages 3–3. USENIX Association, 2011.
- [9] R. De Nicola and F. 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*, pages 407–419. Springer, 1990.
- [10] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET run time monitor. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, 2009.
- [11] M. Emami, R. Ghiya, and L. 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, pages 242–256. ACM, 1994.
- [12] W. Enck. Defending users against smartphone apps: Techniques and future directions. In *International Conference on Information Systems Security*, pages 49–70. Springer, 2011.
- [13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation*, pages 1–6. USENIX, 2010.
- [14] W. Enck, D. Ocateau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of the USENIX conference on Security*, pages 21–21. USENIX Association, 2011.
- [15] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of the*

- ACM conference on Computer and communications security, pages 235–245. ACM, 2009.
- [16] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2004.
  - [17] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the Workshop on New security paradigms*, pages 87–95. ACM, 2000.
  - [18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the Conference on Computer and Communication Security*, pages 627–638. ACM, 2011.
  - [19] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. of the workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14. ACM, 2011.
  - [20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proc. of the USENIX Security Conference*. USENIX Association, 2011.
  - [21] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective tpestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.
  - [22] M. Fredrikson, R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, and V. Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proc. of the Conference on Computer Aided Verification*, pages 548–563. Springer, 2012.
  - [23] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proc. of Mobile Systems, Applications, and Services*, pages 281–294. ACM, 2012.
  - [24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of the Symposium on Principles of Programming Languages*, POPL, pages 58–70. ACM, 2002.
  - [25] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *Symposium on Principles of Programming Languages*, volume 42 of *POPL*, pages 339–350. ACM, Jan. 2007.
  - [26] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, Oct. 2009.
  - [27] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. of the Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.
  - [28] É. Payet and F. Spoto. Static analysis of Android programs. In *Proc. of the Conference on Automated Deduction*, pages 439–445. Springer, 2011.
  - [29] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *ACM SIGOPS Operating Systems Review*, 37(5):15–28, 2003.
  - [30] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.
  - [31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.
  - [32] M. Y. Vardi. From philosophical to industrial logics. In *Proc. of the Indian Conference on Logic and Its Applications*, pages 89–115. Springer, 2009.

## A Application, Action and Event Details

Table 4 lists the sample applications used to evaluate Pegasus. Table 5 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 users 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 4.** 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 5.** Security actions and events used in specifications.