SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps

David Sounthiraraj Justin Sahs Garret Greenwood Zhiqiang Lin Latifur Khan
Department of Computer Science, The University of Texas at Dallas
{david.sounthiraraj, justin.sahs, garrett.greenwood, zhiqiang.lin, lkhan}@utdallas.edu

***Abstract*—Many Android apps use SSL/TLS to transmit sensitive information securely. However, developers often provide their own implementation of the standard SSL/TLS certificate validation process. Unfortunately, many such custom implementations have subtle bugs, have built-in exceptions for self-signed certificates, or blindly assert all certificates are valid, leaving many Android apps vulnerable to SSL/TLS Man-in-the-Middle attacks. In this paper, we present SMV-HUNTER, a system for the automatic, large-scale identification of such vulnerabilities that combines both static and dynamic analysis. The static component detects when a custom validation procedure has been given, thereby identifying potentially vulnerable apps, and extracts information used to guide the dynamic analysis, which then uses user interface enumeration and automation techniques to trigger the potentially vulnerable code under an active Man-in-the-Middle attack. We have implemented SMV-HUNTER and evaluated it on 23,418 apps downloaded from the Google Play market, of which 1,453 apps were identified as being potentially vulnerable by static analysis, with an average overhead of approximately 4 seconds per app, running on 16 threads in parallel. Among these potentially vulnerable apps, 726 were confirmed vulnerable using our dynamic analysis, with an average overhead of about 44 seconds per app, running on 8 emulators in parallel.**

I. INTRODUCTION

The recent proliferation of smartphones has led to many software vendors extending their service to the mobile domain. There are more than 1 million Android apps in the Google Play market alone, with over 50 billion downloads [38]. For many of these apps, the secure transfer of data across the network is a prime concern. To ensure security, many apps transfer sensitive data using the HTTPS protocol (HTTP over SSL/TLS), which is designed to guarantee security, even if a malicious attacker is able to intercept and modify network traffic between the app and the server.

Unfortunately, if a client fails to properly validate SSL/TLS (henceforth SSL for brevity) certificates, it may lead to an SSL Man-in-the-Middle (MITM) vulnerability [27]. In an MITM attack, an attacker is able to intercept and modify network traffic between the client and the server. Normally, if the client properly validates certificates, the attacker cannot decrypt the network traffic. However, if the client accepts certificates without checking their signatures, or if the client accepts self-signed certificates without prompting the user, the attacker can pose as the server by presenting a fraudulent certificate. In this case, the attacker can decrypt the network traffic with her own fraudulent certificate, and can read or modify it at will.

Recently, a number of efforts have focused on analyzing the prevalence of apps vulnerable to MITM attacks in the Google Play market. In particular, Georgiev *et al.* demonstrated that SSL certificate validation is completely broken in various popular security-critical apps and libraries [32]. Similarly, Fahl *et al.* delve deep into SSL MITM Vulnerabilities (SMV in brevity) in Android apps [30]. They developed a tool called Mallodroid, which they used to statically analyze 13,500 Android apps, out of which 1,074 were found to be potentially vulnerable. From these potentially vulnerable apps, the authors chose 100 popular apps and conducted manual inspections, and found that 41 of them were truly vulnerable to SSL MITM attacks. However, a weakness of both approaches is that they rely on *manual analysis* to identify or confirm vulnerabilities. This methodology simply does not scale to large markets like Google Play.

Thus, to enable the automatic, large scale identification of SMV, we need a new set of techniques and tools. The development of these techniques is the focus of this paper. Specifically, we propose SMV-HUNTER, a set of novel techniques to overcome the challenges of automated SMV identification for Android apps. The key observation is that existing static analysis alone is not sufficiently powerful to reliably detect SMV. The output of logical conditions, runtime data dependency and user interaction cannot be determined statically. This non-determinism causes inaccuracy in static analysis. Therefore, in order to detect SMV automatically, we must include dynamic analysis, in which the target app is actually observed during execution. In such an approach, a critical step in detecting a vulnerability is to trigger the vulnerable behavior by simulating the user interaction that the app expects.

On the other hand, a purely dynamic approach would perform an exhaustive search of all possible user interface (UI) paths, which is prohibitively slow. Additionally, apps often validate or convert text input before accepting it. A purely dynamic approach would have difficulty supplying valid text, as it cannot determine what validation or conversion the app will apply to the input text. To address these issues, we propose

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2014.23205

a hybrid static-dynamic approach, in which static analysis is used to guide dynamic analysis by pruning the search space, and supply more valid text. We have also developed a novel UI automation framework which intelligently analyzes the running app by interacting with the Android window manager and providing smart input without human intervention.

In summary, this paper makes the following contributions:

- We present SMV-HUNTER, a novel system for performing automatic, large-scale analysis of SMV in Android apps. SMV-HUNTER contains a static analysis component to identify potentially vulnerable apps, identify the app entry points that lead to vulnerable behavior, and generate smart input for text fields; and a dynamic analysis component to confirm the vulnerability. Additionally, SMV-HUNTER is built to be modular. It could easily be repurposed for other tasks, such as software testing or detection of other vulnerabilities.
- We develop a fully automated framework to run Android apps in parallel on multiple emulators, while collecting vital information such as app logs, and network and system call traces.
- We demonstrate the efficacy of SMV-HUNTER with 23,418 android apps collected from the Google Play market in July 2012. Static analysis identified 1,453 apps as potentially vulnerable, of which 726 were confirmed vulnerable by dynamic analysis.

The rest of this paper is organized as follows: in §II, we review the technical background knowledge required to understand our system; in §III, we give an overview of our system; in §IV, we describe the static analysis components of SMV-HUNTER; in §V, we describe the dynamic analysis components; in §VI, we present our experimental evaluation of our system; in §VII, we review related works; in §VIII, we discuss limitations and future work; finally, in §IX, we conclude our paper.

II. BACKGROUND

A. SSL/TLS

RFCs 2818, 2246 and 3280 enumerate the rules that should be followed to establish a secure SSL/TLS connection. Compliant clients must check that the certificate chain is valid, and that the hostname is valid. A certificate chain is valid if:

- Each certificate in the chain has not expired, and
- The root certificate of the chain is from a trusted Certification Authority (CA) present in the client's default keystore (a list of trusted CAs maintained by the client), and
- If the certificate chain has more than one certificate, the chain should be validated by checking that each certificate has been signed by the CA immediately after it in the chain. If there is only one certificate, it is said to be self-signed, and it is therefore the root CA in the chain, subject to the validation above.

A hostname is valid if the name in the certificate matches the domain name of the server being connected to, possibly including wildcard matching.

The Android OS provides built-in SSL certificate validation and hostname verification which includes a keystore, but allows developers to provide their own implementation by creating classes which implement the X509TrustManager and HostNameVerifier interfaces, respectively.

There are a number of reasons why a developer might choose to override the SSL certificate validation procedure:

- In early versions of the Android platform, there were errors in the SSL certificate validation procedure which caused some valid certificates to be rejected [9].
- If an app connects to a server whose certificate's root CA is not present in the keystore, the app cannot establish a secure connection using built-in certificate validation.
- To avoid the cost of procuring valid certificates for development, testing, and user acceptance environments, developers often use self-signed or invalid certificates.
- Some popular 3rd-party libraries such as ACRA override the built-in SSL certificate validation with vulnerable implementations [32], rendering any app that uses such libraries vulnerable.

These custom implementations often have errors, intentionally accept all self-signed certificates, or even just accept all certificates without checking anything [30]. In these cases, the app is left vulnerable to SSL MITM attacks. The objective of SMV-HUNTER is to identify these vulnerable apps in a large scale manner.

B. Android UI Composition

The visual components of Android apps are called activities, which create screens, which are analogous to windows in a typical desktop computing environment [1]. These activities create and compose UI components either by using declarations in an XML file, or programmatically at runtime. Additionally, activities can use fragments, which are reusable UI components which allow the developer to define and manage the screen at runtime, without switching activities [1].

The UI components that are composed on a screen can be classified into three broad categories: editable components, clickable components, and static components. Editable components have an internal state that can be modified, such as text boxes, check boxes and radio buttons. Clickable components are components without state that cause some action when tapped, such as buttons or links. Static components do not react to user interaction. There are other ways that actions can be triggered, such as the `Menu` or `Home` buttons, which are physically distinct from the device's touchscreen. Note that SMV-HUNTER will only interact with on-screen components.

Typically, each activity is associated with one screen, but it is possible to use a single activity for the whole app [20]. We therefore define a window as the displayed content of a screen at a given point in time. Then, a typical Android app's UI will consist of a discrete collection of windows. Specifically, we represent a UI as a directed graph G, whose nodes correspond to windows, and whose edges correspond to actions caused by clickable components. If an action does not change the

UI display, we represent this as a self-loop. This abstraction helps us represent complex UIs compactly, and allows us to formulate our UI Automation algorithm in terms of graph traversal.

III. System Overview

In this section, we first define our research problem in §III-A, then walk through the challenges in §III-B, and finally give an overview of our system in §III-C.

A. Problem Statement

The goal of SMV-HUNTER is to identify SMV in Android apps. Our solution must be fully automatic and scalable to very large datasets. Towards this end, we have four major design goals:

- ***Coverage:*** Our goal is to have a tool that can be applied to entire markets, which means it needs to be able to run as many apps as possible. Due to the reliance of many apps on proprietary components of the Android OS, this means that our solution must work with a stock Android image provided by Google.
- ***Efficiency:*** The system should be efficient enough to be run on large sets of apps. It should avoid testing code paths that static analysis shows are not vulnerable.
- ***Robustness:*** The Android emulator [4] and Android Debug Bridge (ADB) [3] provided by Google have issues with instability when run for prolonged periods [8], [10]. The emulator tends to switch to an "offline" state, becoming unresponsive. The system should be able to avoid such problems or detect them and take corrective action.
- ***Accuracy:*** The system should be able to detect vulnerable apps without false positives.

A system meeting these requirements could be used at the market level (e.g., on Google Play) to enforce stricter security requirements, or by organizations who want to enforce strict security requirements on employee's devices: software on devices could prevent non-vetted apps from being installed, and as employees request apps, they could be analyzed automatically and efficiently, rather than requiring a manually-maintained whitelist of acceptable apps.

B. Challenges and Key Techniques

Towards realizing these goals, we have identified the following challenges and techniques:

Simulating User Interaction Simulating user interaction with the app requires understanding what is being displayed on the screen and providing intelligent input. As an example, consider the login screen of an online banking app. A typical login screen will contain `username` and `password` text boxes, possibly a "`remember me`" check box, and a login button which will submit the user's credentials when clicked. The user will typically provide input to these elements in this order, starting with the `username`, and ending with tapping the login button. A useful *UI automation* component should be able to simulate this behavior without the need for human intervention or guidance.

After analyzing existing tools for UI automation, we have concluded that these challenges require new techniques. In particular, Google's Monkey tool [17] cannot accurately simulate the controlled behavior of the user because it provides randomized UI events. Another existing UI automation framework is Robotium [14], which is a popular tool used widely by Android developers for testing. This framework is tightly coupled with Android's instrumentation framework, which causes Robotium test scripts to be tightly coupled with the target apps. This makes it unsuitable as a generic UI automation solution as it requires a unique test script for each target app.

Managing Application State Recall from §II-B that we represent an Android app's UI as a directed graph. Then, during an app's execution, the UI behaves like a state machine, where the app's "state" is the current window being displayed. We track this state in order to direct our search of the program's UI. We have a set of "target" nodes that were identified as vulnerable entry points, and we wish to explore the program's code paths that start from these entry points. To prevent spending too much time exploring any single entry point, we wish to limit our search to the target nodes and their children. To enforce this limit, our system must be able to detect when the app transitions to a new state.

This key functionality is missing from many existing systems. While the framework proposed in [41] provides similar functionality by curtailing all code paths that lead to unwanted activities, it is not general and requires the customization of the Android OS. Therefore, our system exploits the `FocusChange` and `WindowChange` events provided by the Android ViewServer.

Testing Efficiency UI automation is typically a slow and resource-intensive process. In our experiments, we find that UI automation takes an average of approximately 45 seconds to traverse all of the possible UI paths on one window. In order to maintain feasibility, we use static analysis techniques to drastically reduce the number of windows we must test. Our system disassembles each app and checks for a custom implementation of the X509TrustManager or HostNameVerifier interfaces. If any such implementation is found, we construct a method call graph and trace the invocation of the potentially vulnerable code back to the appropriate window. We can then eliminate apps that do not provide such custom implementations, as they will use the (correct) built-in SSL certificate validation procedure. Among those apps that do override these interfaces, our *UI automation* component can restrict the automation to those entry-point windows identified as invoking the overridden SSL certificate validation interface.

Even with the speedup we achieve from static analysis, the sheer number of available Android apps makes sequential testing of apps prohibitively slow. To achieve enough of a speedup to make testing large markets feasible, we must test multiple apps in a parallel manner.

Large Scale Automation Another key challenge is the orchestration of multiple Android emulators for parallel execution of *UI automation*. Google's MonkeyRunner [12] tool is designed

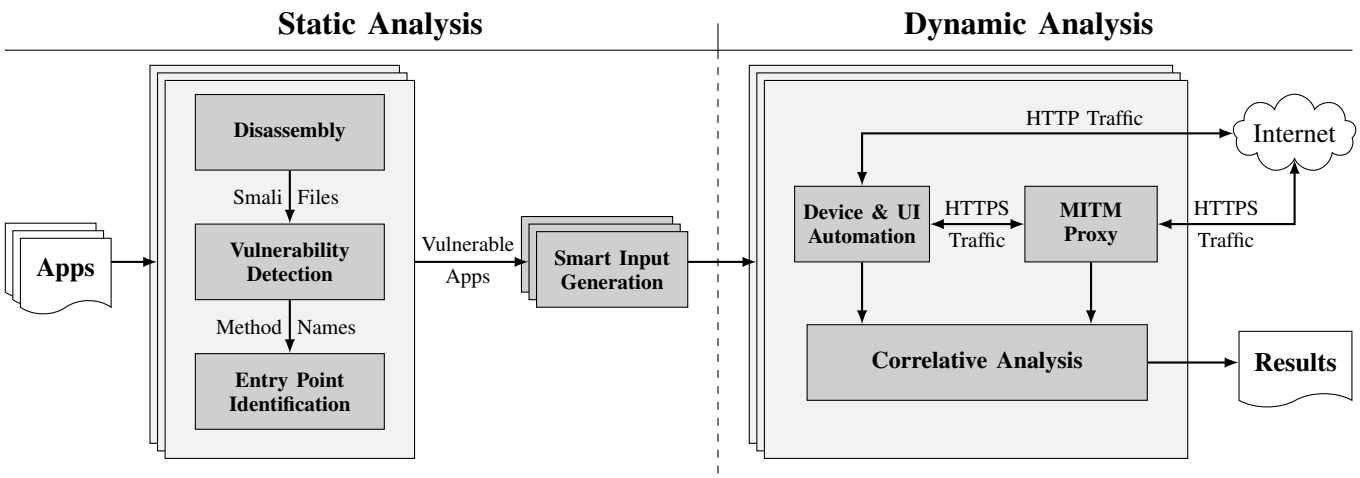

Fig. 1: System Overview

to support multiple emulators running in parallel, but it has several shortcomings that prevent its use in our large scale automation scenario. One of the more severe problems is its lack of error propagation. Most of the methods provided by the MonkeyRunner API [11] have no return value and throw no exceptions. Thus, API users are not provided any feedback in cases where the intended action fails, making the API unpredictable. For example, the `press` method sends a specified key press event to the emulator. If the key press event dispatch fails, the user will be oblivious to the failure. This lack of immediate, explicit feedback when failures occur prevents users from taking corrective action.

To address these issues, we have built a *device management* component based on the Android ADB tooling framework [3]. It manages the emulators and orchestrates the process of installing apps, executing *UI automation*, and collecting statistics such as app logs and network traffic logs, while being tolerant towards the erratic behavior of the emulator.

To our knowledge, there has been no other published work detailing device management techniques. In [37], Rastogi *et al.* mention developing a device management framework, but do not include specifics. The only comparable work publicly available is MonkeyRunner. Our system has a number of advantages over MonkeyRunner, as shown in Table I.

C. System Overview

An overview of SMV-HUNTER is presented in Figure 1. Given a large set of Android apps, SMV-HUNTER first performs static analysis on each app, determining if they are vulnerable.

	MonkeyRunner	SMV-HUNTER
Direct Failure Feedback	✗	✔
Stability over Multiple Emulators	✗	✔
Stability over Long Runtimes	✗	✔
Error Propagation	✗	✔
Targeted Towards Large-scale Use	✗	✔
Easy to Use	✔	✔
Python-based	✔	✗

TABLE I: MonkeyRunner vs. SMV-HUNTER

When potential vulnerabilities are found, further analysis traces the invocation of the vulnerable code back to an entry point window. The potentially vulnerable apps are then analyzed to generate smart input for text boxes. Next, these potentially vulnerable apps and entry windows are given to the *device management* component, which installs the apps and executes the *UI automation* on the windows. While the *UI automation* runs, the *device management* component captures logging information for later processing. As the *UI automation* triggers HTTPS traffic from the apps, this traffic passes through the proxy, which attempts an SSL MITM attack, and logs successes and failures along with identifying features and the server being connected to. Finally, the output is aggregated and processed to combine the data from the *device management* and proxy components, producing a final list of confirmed vulnerable apps.

IV. STATIC ANALYSIS

In the first phase of SMV-HUNTER, each Android app is disassembled and analyzed statically to detect potential SMV. The detailed design of this static analysis component is described below.

A. Disassembling the Apps

Android apps are distributed as packages that contain compiled Android app code [6]. This code can be decompiled to Java, or disassembled to a human-readable format called Smali [23]. In SMV-HUNTER, we disassemble the bytecode to Smali using a tool called `apktool` [2]. We choose Smali disassembly over decompilation for two reasons. First, Smali disassembly is much faster than decompiling to Java. Second, the decompilation process can be hindered by obfuscation techniques that produce code decompilers do not understand, whereas disassembly does not have this shortcoming.

B. Static SMV Detection

Once we have disassembled the app, identifying potential SMV is straightforward: we simply check whether the app overrides the X509TrustManager or HostNameVerifier interfaces.

Algorithm 1: `traverse`: Vulnerable Entry Point Identification

```
input  : seed, the constructor of a vulnerable class
         node, the current node in the MCG
         constructors, the set of traversed constructors
output : the set of vulnerable entry point methods
1  begin
2      if parents(node) ≠ ∅ then
3          for parent ∈ parents(node) do
4              traverse (seed,parent,constructors)
5      else
6          for method ∈ methods(class(node)) do
               /* method is never called. Continue traversing from its class' constructor.
                  */
7              if method is the class' constructor ∧ method ∉ constructors then
8                  constructors ← constructors ∪ method
9                  traverse(seed,method,constructors)
               /* method is a constructor that is never called; report it as an entry
                  point.                                                          */
10             else if method ∈ constructors then
11                 output(method,seed)
```

Apps that do not override these interfaces either do not use SSL or use the built-in SSL support without modification, and can therefore be considered secure. Apps that do override these interfaces do so at great risk, often introducing vulnerabilities.

We have identified the most common vulnerable implementations of the X509TrustManager and HostNameVerifier interfaces by manually analyzing 1,000 random apps, 252 of which were vulnerable.

We have identified four common patterns that result in SMV:

X509TrustManager based:

1) *No-op:* The most common implementation of the X509TrustManager interface is the "no-op" implementation which asserts all certificates are valid without looking at them.
2) *Trusting Self-signed Certificates:* In this pattern, the implementation checks if the certificate chain consists of a single certificate, as is the case in self-signed certificates. In this case, it uses checkValidity to check that the certificate has not expired, but does not verify the certificate's signature or ask the user if they want to trust a self-signed certificate. For example, the Apache HTTPClient library's wiki section [18] has an example implementation called EasySSLProtocolSocketFactory which follows this pattern. This implementation has been copied and pasted into many apps (including the Chase banking app mentioned in [32]), despite the code containing comments that warn against using it in production.
3) *checkValidity Only:* In this final pattern, the implementation iterates through the certificate chain, using checkValidity to check that each certificate has not expired, but does not do any other validation.

HostNameVerifier based:

4) Host name verification is most commonly implemented as a "no-op" implementation, often by using the AllowAllHostnameVerifier provided by the Apache HTTPClient library [22].

C. Vulnerable Entry Point Identification

A typical Android app will have many entry points (e.g., activities and services [15]), making dynamic analysis that exhaustively executes them prohibitively slow. However, many (sometimes most) of these entry points lead to code paths that do not involve making HTTPS connections. Therefore, SMVHUNTER identifies those entry points that lead to the invocation of the vulnerable code identified during static analysis. To achieve this, we construct a method call graph (MCG) for each app, and trace each vulnerable method back to the entry point that ultimately causes its execution. Because we only construct a graph of methods contained in the compiled app (and do not include methods found in the Android libraries, for instance, as this would require deep static analysis of the entire Android operating system code), we use the modified MCG traversal procedure in Algorithm 1 to identify vulnerable entry points.

To find entry points that execute a particular vulnerable method, we start at that method (called **seed** in the algorithm), and we traverse into its parents (the methods that call it), and into their parents, and so on, until we reach a method that has no parents. In a typical MCG traversal procedure, this would be the end of the traversal. However, it is often the case that a method we reach is only ever called by system code, so when

Algorithm 2: Smart Input Generation

```
input  : app, the app to be analyzed
output : input text for text boxes
1  begin
2      map ← ∅
3      for act ∈ getActivities(app) do
4          layout ← getLayoutForActivity(act)
           // loop over EditText elements in the layout
5          for elem ∈ getUIElemsByType(layout, "EditText") do
6              elem_id ← getUIElemID(elem.name) // translate the element's name to an id
7              input_type ← getUIElemInputType(elem_id) // get the input type if there is one
8              if input_type found then
9                  map ← map ∪ {elem_id ↦ input_type}
10                 continue to the next element
               // if there is no type annotation,
               // trace the access of the element through the activity's code
11             instance_var ← getInstanceVariableForID(act, elem_id)
12             reads ← getAccessLocations(instance_var)
13             for {read ∈ reads | "getText" ∈ read} do // when the getText function is called
                   // search for and record type casts
14                 text_var ← variable containing the result of the getText call
15                 type_cast ← getTypeCastOperation(text_var)
16                 if type_cast found then
17                     map ← map ∪ {elem_id ↦ type_cast}
18                     continue to the next element
19                 else
20                     continue to the next read
21     return translateTypeToInput (map)
```

we reach a method with no parents, we jump to the constructor of that method's class, and continue traversing from there. This allows us to continue traversing when the developer instantiates an object and passes that object to the operating system. Only when we reach a constructor that is never called in the app code do we stop traversing. These constructors are therefore only called by system code, and are the entry points to the app.

The identified vulnerable entry points correspond either to activities or services. Services are non-UI components mostly associated with long-running background processes that are unlikely to trigger SSL connections, so we only trigger activities declared in the app's manifest file [6].

D. Smart Input Generation

Android apps often perform validation on text input, or convert from text to some other datatype (such as integers or floating point numbers). If the dynamic analysis tool does not supply valid input during execution, the app will not perform the desired operation, and may even crash. Previous work in dynamic analysis has relied on providing randomized input [33] or hand-crafted tables matching visual labels to valid input [37]. SMV-HUNTER instead uses static analysis techniques to leverage information available in the apps' metadata and code to determine the form of valid input. In particular, SMV-HUNTER uses two sources of information: developer-supplied input type annotations and type casts in the code. The input type annotations are used by developers mainly to control the keyboard that appears when a user selects the input field, and to restrict the characters that the user is able to input.

As shown in Algorithm 2, SMV-HUNTER generates smart input by attempting to assign a data type for each text field. Once a type has been assigned, the system can use a simple table to provide typical input of that type. The type assignment process begins by looping over every activity in the targeted app (line 3). Each activity will declare a layout with a call to `setContentView`. The system extracts this call from code and loads the associated layout XML file (line 4). From the layout file, the system extracts UI elements, specifically elements of the `EditText` type, and loops over these elements, extracting the element's ID and input type annotation (lines 5-7). If there is a type annotation, the system uses that and moves on to the next UI element (lines 8-10).

If there is no annotation, SMV-HUNTER attempts to extract type information from the activity's disassembled code. To do this, it first finds variables that reference the elements by ID (line 11). Next, the system collects all parts of the code that access these variables (line 12), and for each call to the element's `getText` function (line 13), it tracks usage of that value through any type cast operations (lines 14-15), and uses any such type casts as type labels (line 17). Finally, it converts

AndroidManifest.xml

```
<activity android:label="@string/app_name" android:name="com.example.testproject.MainActivity">
```

MainActivity.smali

```
const/high16 v0, 0x7f03
invoke-virtual p0, v0, Lcom/example/testproject/MainActivity;->setContentView(I)V
```

R$layout.xml

```
.field public static final activity_main:I = 0x7f030000
```

activity_main.xml

```
<EditText android:id="@id/integer_field" />
...
<EditText android:id="@id/phone_field" android:inputType="phone" />
```

R$id.smali

```
.field public static final integer_field:I = 0x7f080000
```

MainActivity.smali

```
1  const/high16 v0, 0x7f08
2  invoke-virtual {p0, v0}, Lcom/example/testproject/MainActivity;->findViewById(I)Landroid/view/View;
3  iput-object v0, p0, Lcom/example/testproject/MainActivity;->integer:Landroid/widget/EditText;
4  ...
5  iget-object v3, p0, Lcom/example/testproject/MainActivity;->integer:Landroid/widget/EditText;
6  invoke-virtual {v3}, Landroid/widget/EditText;->getText()Landroid/text/Editable;
7  move-result-object v3
8  invoke-interface {v3}, Landroid/text/Editable;->toString()Ljava/lang/String;
9  move-result-object v3
10 invoke-static {v3}, Ljava/lang/Integer;->parseInt(Ljava/lang/String;)I
11 move-result v3
12 invoke-static {v3}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
13 move-result-object v0
```

```
1  integer = (EditText) findViewById(R.id.integer);
2  ...
3  Integer parsedInt = Integer.parseInt(integer.getText().toString());
```

Fig. 2: Sample Code showing Type Information

these type annotations to input strings (line 21). To the best of our knowledge, SMV-HUNTER is the first such system that can provide intelligent input to UI elements.

Figure 2 shows some typical sample code from which SMV-HUNTER can extract type information. In particular, there are two `EditText` fields: one which expects an integer, but provides no input type annotation, and one which expects a phone number, and uses the appropriate input type annotation. To extract these types, the system first looks at `AndroidManifest.xml` to find the activity name (MainActivity). Using this name, it next looks in `MainActivity.smali`, looking for calls to `setContentView`, and extracting the ID being passed as an argument (`0x7f03` in this case). The system then looks in `R$layout.xml` to find the name associated with that ID (`activity_main`). Finally, SMV-HUNTER opens the associated file (`activity_main.xml`), and searches for `EditText` fields, and extracts their names, and any input type annotations. In the case of the field named `phone_field`, there is now enough information to associate a type with the field: it is of type `phone`.

The other field, named `integer_field`, does not supply any type annotation, so the system must rely on code analysis to determine its type. SMV-HUNTER first looks the name up in the file `R$id.smali` to find its associated numeric ID (`0x7f080000`). Next, it looks in the disassembled code, specifically `MainActivity.smali`, in order to to associate the ID with a variable name. In line 1-3 of `MainActivity.smali`, SMV-HUNTER traces the use of the ID through a call to `findViewById`, which returns an object which is then associated with the name `integer`. Later in the code, the system then uses data flow analysis provided by Androguard [28] to find places where this name is accessed (line 5), then searches for the `getText` method call (line 6), and traces the result (in register `v3`) to a call to `parseInt` (line 10). Then, the system can associate the Integer type with the name `integer_field`.

V. DYNAMIC ANALYSIS

In the second phase of SMV-HUNTER, the *device management* component runs each app in an emulator, triggering the *UI automation* component and collecting logging information. Meanwhile, a proxy monitors all HTTPS traffic and attempts to launch an MITM attack, logging successes and failures. The output of each of these components is collected and aggregated by the *correlative analysis* component.

A. Device Management

The *device management* component forms the core of our dynamic analysis. It is responsible for managing emulators, monitoring their state, installing apps, and running *UI automation*. To ensure completeness and efficiency, it must:

Algorithm 3: `schedule`: Application Scheduling

input : **apps**, a list of apps to be tested

```
1 begin
2     for app ∈ apps do
          /* get an emulator from the management thread; this is a blocking call */
3         emulator ←getEmulator ()
4         install (emulator,app)
          /* for each vulnerable entry point from static analysis */
5         for activity ∈ getEntryPoints(app) do
6             startActivity (emulator,activity)
7             automateUI (emulator,activity)
8         uninstall (emulator,app)
9         releaseEmulator (emulator)
```

- Manage multiple emulators in parallel,
- Understand and detect the internal state of each emulator ("online" versus "offline"), and take corrective action by restarting any emulator that has gone "offline",
- Handle emulator crashes and other errors,
- Dynamically manage a pool of emulators which may shrink or grow during execution,
- Schedule and distribute app testing across running emulators, and
- Collect and manage log data including installation and uninstallation details, Android OS-level logs, and network traffic.

To address these requirements, the *device management* component consists of two main threads: an emulator management thread, and an app scheduling thread.

Emulator Management The emulator management thread manages two pools of emulators (each emulator in their own thread): the running pool and the free pool.

When the system is started, it registers a `DeviceChangeListener` callback with ADB. When an emulator is started or dynamically added to the system, it enters the "online" state, and the management thread adds the emulator thread to both the running pool and the free pool. If the emulator ever enters the "offline" state or crashes, the management thread removes the thread from both pools, stops the emulator, and starts a new emulator in its place. Thus, the running pool contains all emulators that are in the "online" state. The free pool contains emulators that are ready to be used; these emulators are "online", but not currently testing an app. When the scheduler requests an emulator, it is removed from the free pool and returned to the scheduler. When the scheduler finishes a job, it returns the emulator to the emulator management thread, which adds it back to the free pool.

App Scheduling The app scheduling thread manages a list of apps to be tested, and processes them as detailed in Algorithm 3. As shown in the algorithm, the scheduler iterates over each app to be tested (line 2), first getting an emulator from the management thread (line 3), then installing the app on that emulator and running *UI automation* on each vulnerable activity identified by static analysis (lines 4-7). Then, the scheduler uninstalls the app and returns the emulator to the management thread (lines 8, 9). This algorithm is simplified, however; the scheduler also monitors the size of the running pool, and creates a thread for each emulator, distributing apps among them, so that the emulators execute in parallel (i.e. the loop of line 2 is parallelized). Additionally, the scheduler handles any errors reported by the emulator. If installation fails, it retries once, then abandons that app. If failure occurs during *UI automation*, it moves to the next entry point.

B. UI Automation

UI automation is a key component in SMV-HUNTER. It emulates the user's interaction with the Android app, driving the app's execution in ways that are likely to lead to vulnerable code being executed. The system explores code paths that originate in each vulnerable entry point identified during static analysis. The *UI automation* component is shown in Figure 3.

The *UI automation* component has three goals: understanding the interface as it is displayed, providing intelligent input to the app, and understanding and managing the state of the app.

Understanding the Interface The first step in automating the UI is to decompose the UI into its component elements. For each of these elements, the system extracts properties such as the coordinates that define its boundaries, and what form of input (e.g., text or tap) it expects. With this information, the system crafts the appropriate input events to send to the app. For example, if the UI component is a button or a checkbox, a click event with the appropriate coordinates is generated; if the UI component is a textbox, text input events are generated.

To identify the window's elements and extract their properties, the system utilizes the Android ViewServer [39], an internal component of the Android app tooling framework. The ViewServer provides a set of commands that can be used to query the Android WindowManager, which handles the display of UI elements, and the dispatch of input events

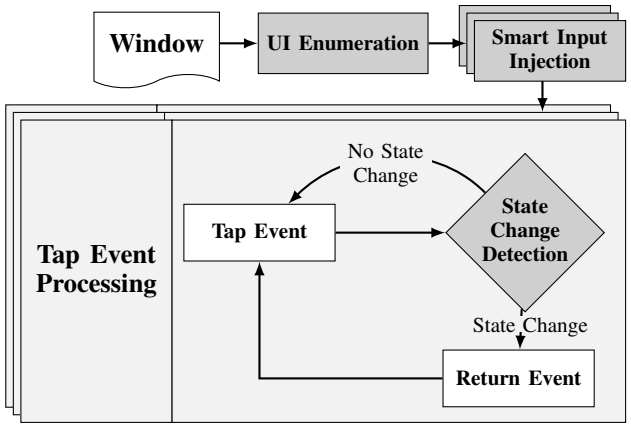

Fig. 3: UI Automation Component

to the appropriate element. Specifically, our system queries the ViewServer to retrieve the entire view hierarchy, which contains all of the UI elements and their properties, such as coordinates and editability properties.

Generating Input Events Once the system has identified the elements of the UI and the type of input they require, it must generate the input events to send to the app. For text fields, we use the text supplied by the *smart input generation* component (see §IV-D). Once the input event has been crafted, the system uses the `input` command available through ADB. This command supports both text input events and tap events at specific coordinates.

Overall, the process of automating a window has two phases. First, the system fills all editable text fields by iterating through them, generating tap events that focus them, then inputting the smart input generated by static analsysis. Then, the system iterates through all clickable elements and generates tap events at the appropriate coordinates. Between each tap, the *UI automation* component waits for a response from the *state management* component (described below) to respond. When it receives a response, the system proceeds to the next element.

Application State Management We utilize the API provided by Android's ViewServer component to obtain information about the app's current state and to detect state changes. The ViewServer provides `WindowChange` and `FocusChange` events, which are triggered when the state changes. By registering handlers for these events, our system is notified of any state transition. The *UI automation* component waits after each tap event is processed so that the handlers have time to react to a state change. When the app transitions to a new state, the *UI automation* component generates a "back button" event, which causes Android to pop the current window from its stack, returning to the target window.

Android allows for "non-cancellable" dialogs and similar UI components that temporarily disable the effect of the back button. In these cases, the back button event generated by the *UI automation* component has no effect, so the system checks for a state change before resuming normal operation. If the state remains unchanged, additional tap events are generated; these should click on any "`OK`" or "`Cancel`" buttons, dismissing the dialog and returning to the target window. If the state remains unchanged after three such tap events, the system terminates the app, abandoning the current activity and moving on to the next entry point.

C. MITM Proxy

To execute an SSL MITM attack, we must intercept all HTTPS traffic between the emulators and the Internet. When running multiple emulators, the sheer number of connections can overload standard proxy software. One widely-used MITM attack proxy is Mallory [19], but in our experiments, it could not handle many simultaneous connections, and tended to crash silently when overloaded or run too long. Because of this, we designed our system using the Burp Suite proxy [21], which generates a single self-signed certificate which it then uses to sign certificates for each attack. Burp allows users to write scripts that modify or log traffic. We use this feature to log successful HTTPS connections (i.e. successful attacks) to a database. We found the Burp Suite proxy to be more stable than Mallory, but it appears to process connections one-by-one, storing incoming connections in a queue until the current connection is completed. With multiple emulators, this leads to many connections timing out at the app level. To mitigate this, we therefore use `iptables` [25] to bypass the proxy for all non-HTTPS traffic, as shown in Figure 1, reducing the load on the proxy, and allowing our system to scale to the required level.

D. Correlative Analysis

The MITM proxy can detect vulnerabilities by successfully attacking apps, but it cannot map vulnerabilities back to the apps that were attacked: it just sees network traffic. Therefore, we have a *correlative analysis* component to map successful attacks to the apps that were attacked, using logs generated by the *device management* component and the MITM proxy. Because we test several apps in parallel, there may be multiple successful attacks at approximately the same time. Additionally, network delays may cause timestamps to differ slightly between the *device management* component and the MITM proxy. Therefore, we cannot use simple timestamps to match attacks to apps. Because the emulators are all running on the same machine and sharing the same network interface, they do not get unique MAC addresses or IP addresses, so we cannot use addresses to match attacks to apps. However, because the MITM proxy logs impersonated domain names, we can use DNS lookups to strengthen fuzzy time matching.

The *correlative analysis* component works as follows: First, installation timestamps from the *device management* component are used to map each app to the block of time it was running. Second, network logs from the *device management* component are searched for DNS queries, which are used to map these time blocks to Internet domains. Finally, MITM proxy logs are used to generate a second mapping from time blocks to Internet domains. When a time block from the MITM proxy overlaps a time block from the *device management* component with the same domain, the associated app is marked as vulnerable.

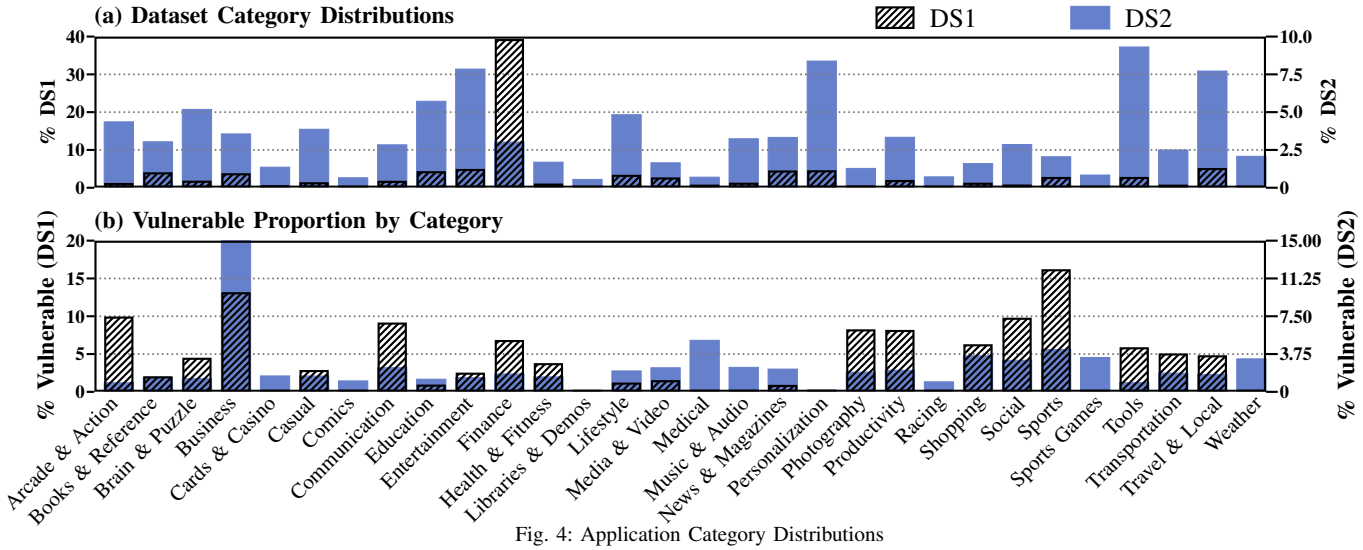

Fig. 4: Application Category Distributions

VI. Evaluation

In this section, we present our experimental results. We ran our static analysis on two Dell T7500 machines, each with Intel Xeon E5620 8-core processors and 24 GB memory. To simplify our job scheduler and to avoid overwhelming the MITM proxy, we ran dynamic analysis on only one of these machines. Therefore, timings presented are run on 16 cores for static analysis, and 8 emulators (one per core) for dynamic analysis.

In §VI-A, we discuss our datasets. In §VI-B and §VI-C, we present the results of the static and dynamic analysis phases. In §VI-D, we analyze the vulnerable apps we discovered. Finally, in §VI-E, we analyze the updated state of vulnerable apps.

A. Characteristics of Our Datasets

We used two mutually exclusive datasets in our experiments. The first dataset (denoted DS1) was collected by crawling the Google Play market using finance-specific query terms. This dataset is therefore biased towards banking-related apps, as vulnerabilities in these apps will usually have the most devastating consequences. This dataset contains 3,165 apps. The second dataset (denoted DS2) was collected without giving any subject priority, leading to an unbiased selection of apps. This dataset contains 20,316 apps. We initially used DS1 as a smaller testing set during development, based on the intuition that banking and finance-related apps are more likely to make secure connections. The distributions of the datasets are shown in Figure 4 (a). DS1's mostly finance-related apps tend to have simpler interfaces than general apps. For example, DS2 may contain game apps that have highly-customized complex interfaces. The separation of DS1 provides baseline performance measures.

B. Static Analysis

We performed static analysis on both datasets, identifying apps that have vulnerabilities, and identifying the entry points that lead to those vulnerabilities. The results of static analysis are given in Table II.

Time Requirements On average, static analysis took approximately 4 seconds per app, which can be separated into three components:

- *Disassembly:* As discussed in §IV, we use `apktool` [2] to decompile apps into an intermediate format called Smali [23], which closely resembles the JVM bytecode instructions, but is more readable and structured. This process takes 0.42 seconds per app, on average. This is much better than decompiling to Java source code (using the `ded` decompiler [35]), which takes 276 seconds per app, on average, and is much less reliable.
- *Vulnerable Entry Point Identification:* The static analysis process to detect vulnerable entry points took 24 hours to analyze both datasets (3.63 seconds per app, on average).
- *Smart Input Generation:* To generate smart input for text fields, SMV-HUNTER takes approximately half an hour to process DS1 and DS2 (1.2 seconds per app).

Space Requirements Storing the downloaded apps and the results of their disassembly and analysis requires a lot of disk space, as detailed in Table II. Extrapolating from these results, we estimate that analyzing the more than 1 million apps on the Google Play market [38] would require over 8 TB of storage containing over 420 million files. To scale to this level, we

	DS1	DS2
Vulnerable Apps	221	1322
Vulnerable windows	1670	7043
Disassembly	23.5 minutes	2.4 hours
Entry Point Identification	3.2 hours	20.5 hours
Apps with Detectable Text Fields	87	417
Detected Text Fields	600	5599
Annotated Text Fields	289	3532
Type Casts	92	263
Space Requirements	26G	176G
Smali Files	1.3 million	8.7 million

TABLE II: Static Analysis Statistics

	Applications						windows			
	Vulnerable		Install Failed		Run Failed		Time/window		Launch Failed	
Category	DS1	DS2	DS1	DS2	DS1	DS2	DS1	DS2	DS1	DS2
Arcade & Action	3	11	0	9	1	3	16.00	25.19	1	48
Books & Reference	2	12	0	10	0	1	64.62	20.40	0	3
Brain & Puzzle	2	19	0	8	1	5	36.50	31.02	4	50
Business	12	149	0	4	1	4	41.48	38.15	4	57
Cards & Casino	0	6	0	0	1	2	8.00	33.15	1	2
Casual	1	17	3	19	0	0	22.30	32.50	0	23
Comics	0	2	0	0	0	0	-	54.14	0	0
Communication	4	19	0	3	0	1	46.00	43.31	0	74
Education	1	20	1	32	0	3	34.67	35.23	0	18
Entertainment	3	31	2	11	0	6	122.50	59.90	0	54
Finance	64	15	2	8	3	2	48.37	42.99	479	140
Health & Fitness	1	7	0	3	1	2	8.00	68.57	1	46
Libraries & Demos	0	0	0	0	0	0	-	44.44	0	0
Lifestyle	1	28	0	10	0	3	45.14	48.39	0	47
Media & Video	1	11	0	3	0	1	90.00	32.59	0	23
Medical	0	10	0	2	0	0	-	41.92	0	11
Music & Audio	0	22	0	3	0	2	31.50	45.70	0	97
News & Magazines	1	21	1	28	0	6	87.67	39.97	0	45
Personalization	0	4	0	0	0	0	-	60.57	0	4
Photography	1	7	0	4	0	0	23.50	30.22	1	17
Productivity	4	20	1	7	0	0	65.28	50.10	2	54
Racing	0	2	0	3	0	0	-	45.69	0	45
Shopping	2	16	0	2	0	1	72.40	52.57	7	40
Social	2	25	1	12	0	2	70.73	49.88	0	55
Sports	11	24	0	2	0	2	23.11	31.35	1	32
Sports Games	0	8	1	2	0	0	22.50	20.95	0	34
Tools	4	24	0	5	0	1	42.84	40.41	0	33
Transportation	1	13	0	8	1	1	42.00	31.73	3	31
Travel & Local	6	37	5	29	1	14	45.58	38.23	18	96
Weather	0	19	0	1	0	0	-	31.47	0	4
Total:	127	599	17	178	10	62	49.10	42.78	522	1182

TABLE III: Dynamic Analysis Statistics

recommend using a distributed computing framework such as Hadoop [7], which would provide the ability to distribute the analysis over large clusters of computers, making such a task feasible.

Statistics Of 260,395 windows, static analysis identified 8,713 as being potentially vulnerable entry points. Of these, 607 were the default window for its app (i.e. the first window shown when the app is started). Smart input generation statistics are shown in Table II.

C. Dynamic Analysis

For the dynamic analysis process, we used eight emulators running Android OS 4.1 to test the apps in parallel. This demonstrates that the framework can manage and automate multiple emulators at once. Overall, this process took 18.81 hours to analyze both datasets (2.91 hours for DS1 and 15.90 hours for DS2). While running, we recorded 12 emulator crashes, and observed that each emulator crashed (or went to an "offline" state) at least once, illustrating the instability of the Android emulator when run for prolonged periods.

Often, activities depend on data entered on previous windows. For example, any window after a login window is dependent on the result of that login. When this information is missing because we launch the activity directly, the activity behaves in one of two ways: it will either fail gracefully by redirecting to the skipped window or it will simply crash. In total, we performed dynamic analysis on 8,713 entry points, of which, 1,705 crashed on launch.

Detailed statistics collected during dynamic analysis are given in Table III, which shows: successfully attacked apps, apps that failed to install, apps for which all activities failed to launch, the average time to process each window, and the number of windows that failed to launch, per dataset and per app category. Interestingly, we can see that apps in the business category are significantly more likely to be vulnerable (this is also apparent in Figure 4 (b)). Another notable datum is that apps in the finance category are significantly more likely to have windows that fail to launch. This is likely due to apps requiring login credentials: if you launch a window that normally comes after the login screen, these apps often crash.

D. Vulnerable Apps

Table III shows the distribution of vulnerable apps by category. This is also shown in Figure 4 (b), in which each category's bar represents the relative likelihood of an app from that category being vulnerable. Clearly, the business category is the most vulnerable: from Figure 4 (b), we see that approximately 12-15% of apps in the business category are vulnerable in both datasets. Note that the small number of

Category	Unavailable (%) DS1	Unavailable (%) DS2	Still Vulnerable (%) DS1	Still Vulnerable (%) DS2
Arcade & Action	0	18.18	100	81.82
Books & Reference	0	0	100	100
Brain & Puzzle	0	26.32	100	73.68
Business	0	34.9	91.67	63.09
Cards & Casino	-	0	-	100
Casual	0	5.88	100	82.35
Comics	-	0	-	100
Communication	50	5.26	50	84.21
Education	0	0	100	100
Entertainment	33.33	12.9	66.67	80.65
Finance	7.81	6.67	40.63	73.33
Health & Fitness	0	0	100	85.71
Libraries & Demos	-	-	-	-
Lifestyle	0	10.71	100	85.71
Media & Video	100	45.45	0	54.55
Medical	-	30	-	70
Music & Audio	-	4.55	-	81.82
News & Magazines	0	9.52	100	80.95
Personalization	-	25	-	75
Photography	0	0	0	85.71
Productivity	25	10	75	80
Racing	-	0	-	100
Shopping	0	6.25	100	93.75
Social	0	8	100	88
Sports	0	12.5	100	83.33
Sports Games	-	0	-	100
Tools	0	8.33	100	66.67
Transportation	0	7.69	100	84.62
Travel & Local	0	10.81	100	86.49
Weather	-	0	-	100
Total:	7.87	16.03	64.57	78.63

TABLE IV: Vulerability Statistics for Re-downloaded Apps

non-finance apps in DS1 introduce a large amount of noise in the relative likelihood calculations.

E. Revisting Vulnerable Apps

This project was conducted over a one-year window, allowing us to revisit vulnerable apps, and check whether they had been patched. Therefore, we recently attempted to re-download all 726 confirmed-vulnerable apps from both datasets and analyze the updated versions. Table IV shows the availability of updated versions, and the results of the more recent analysis. The "Unavailable" column shows how many apps we could not re-download. The "Still Vulnerable" column shows the proportion of vulnerable apps that were re-downloaded and found to still be vulnerable.

Overall, 14.6% of apps were unavailable for re-downloading, and 76.17% were still vulnerable, showing that SMV are still very prevalent.

VII. Related Work

Static Analysis There has been a sizeable volume of work focused on using static analysis of Android apps to detect malware, privacy leaks and clone apps, among other things [26], [31], [34], [35], [42], [43]. Octeau *et al* [35] developed a Dalvik decompiler, `ded`, and used a combination of automated tests and manual inspection to analyze apps. They studied the prevalence of advertising libraries and sensitive information leaks. Although `ded` is accurate, it is too slow to be suitable for large scale analysis: in [35], they report their analysis taking almost 500 hours to analyze 21 million lines of code retrieved from just 1,100 free apps

Zhou *et al.* [43] performed a systematic, large-scale study of Android malware using static analysis. Their DroidRanger system used various heuristics to perform static analysis more efficiently. They analyzed approximately 200 thousand apps from various Android markets and reported 211 malicious apps, including two "zero-day" attacks.

Fahl *et al.* [30] performed static analysis on 13,500 popular Android apps, yielding 1,074 apps potentially vulnerable to SMV. They then performed manual analysis on a sample of 100 potentially vulnerable apps, yielding 41 vulnerable apps.

Dynamic Analysis There has been significantly less work related to the dynamic analysis of Android apps. In [29], Enck *et al.* developed TaintDroid, a light-weight system to perform "taint analysis" on Android, which tracks data dependencies in running Android apps and reports when sensitive data is leaked. They report only a 14% performance overhead, but all their testing was completely manual. In [36], Portokalidis *et al.* present a system that traces execution on-device, then "replays" the execution in an emulator, allowing resource-intensive security checks to be run without incurring an on-device overhead. Like TaintDroid, this system has a small (about 15%) performance overhead, but has no support for automation, as it is intended to be a service for live malware detection. In [40], Yan and Yin present DroidScope, a unified analysis platform that seamlessly provides the ability to capture OS-level and Java-level information. As with the other systems, DroidScope lacks automation abilities.

UI Automation In [33], Hu and Neamtiu used automatic randomized (using Monkey [17]) testing to generate test cases and detect bugs. More recently, the SmartDroid system [41] tackles some of the problems of revealing UI-Based trigger conditions in Android apps, but it requires customization of the Android OS. Specifically, it is based on a customization of the Android Open Source Project [5], which does not include proprietary libraries that are included in the commercial version. Among the 726 apps our system identified as definitively vulnerable to SSL MITM attacks, 370 (51%) required the proprietary Google Maps libraries. These apps would therefore fail to install on the SmartDroid system.

Concurrent with our work, Rastogi *et al.* [37] developed AppsPlayground, a malware and privacy leak detection system based on automated UI exploration. The key novelty of their system is their method of intelligently driving UI event generation. This method is substantially similar to the *UI automation* component of our system, differing mostly in the generation of text input, which uses hand-written rules that map UI component labels to inputs, and in some dynamic search space optimizations that avoid retesting the same window. However, their system performs an unguided exploration of all UI paths, whereas our system's static analysis component drastically reduces the search space.

Table V compares our *UI automation* framework with these

Framework	High Code Coverage	Field Type Inference	Fully Automated	w/ GUI Enumeration	w/ Static Analysis	Adaptive Input	Tested w/ Large #Apps	State Aware	Stock OS
Monkey	✗	✗	✔	✗	✗	✗	✗	✗	✔
Smartdroid	✔	✗	✗	✗	✗	✗	✗	✔	✗
SMV-HUNTER	✔	✔	✔	✔	✔	✔	✔	✔	✔
AppsPlayground	✔	✔	✔	✔	✗	✔	✔	✔	✗

TABLE V: Comparison with recent UI automation frameworks

approaches.

VIII. Discussion and Future Work

In this section, we discuss the limitations of our approach and propose future work to address them.

Static Analysis Our static analysis component considers any app that overrides the X509TrustManager or HostNameVerifier interface potentially vulnerable (see §IV). Such conservative analysis will not introduce any false negative results. However, it may introduce false positives in cases where an app's developers override these interfaces but do not introduce vulnerabilities.

Dynamic Analysis Any false positives introduced by the static analysis component are eliminated by the dynamic analysis component, as we will only declare an app to be vulnerable if we actually exploit the vulnerability. However, the current implementation of the *UI automation* component will introduce false negatives due to the following limitations:

- *Multi-Window Input:* In some cases, an app prompts the user for several pages of input before establishing a secure connection. Because we only traverse the immediate child windows of each entry point, our system will fail to trigger the connection in these cases. A solution to this would be to continue providing input in a depth-first-search style, until the app establishes an SSL connection. Such an approach may be prohibitively slow or lead to infinite loops, so it would require some sort of heuristic to determine when to move on to a new entry point. For example, it may be possible to use static analysis to construct a code path from entry point to vulnerability and detect when execution strays from this path. We leave this non-trivial effor to future work.
- *Advanced UI Operations:* Some apps require more complex UI interaction such as swipe or long touch events. Detecting when such forms of input are required, and generating them is left as future work. Additionally, the system is limited to simple UI elements that have one clickable region and do not change shape or position based on input, but it could be extended to support complex UI elements such as Spinners [16], Pickers [13] or custom UI elements which require more complex input. Extending SMV-HUNTER to support these is another venue of future work.
- *WebViews:* Some apps use WebView UI elements, which are essentially embedded browser components, which the ViewServer cannot inspect. We leave to future work the task of inspecting these elements to generate appropriate UI events to test them. Such a solution could leverage existing work on in-browser automation, such as [24].

IX. Conclusion

In this paper, we have developed SMV-HUNTER, a system which combines static and dynamic analysis techniques to perform automated, large-scale SMV detection for Android apps. Our system first uses a static analysis component to detect probable vulnerabilities, identify UI targets to trigger these vulnerabilities, and generate smart input to guide the dynamic analysis component, which performs automatic UI exploration while attempting MITM attacks. Our empirical evaluation results show that our system is practical and effective, achieving detection rates comparable to previous manual analysis.

Acknowledgements

This material is based upon work supported by The Air Force Office of Scientific Research under Award No. FA-9550-12-1-0077. We thank our anonymous reviewers for their helpful comments.

References

[1] Activities. http://developer.android.com/guide/components/activities.html.

[2] android-apktool. http://code.google.com/p/android-apktool/.

[3] Android debug bridge. https://developer.android.com/tools/help/adb.html.

[4] Android emulator. https://developer.android.com/tools/help/emulator.html.

[5] Android open source project. http://source.android.com/.

[6] Glossary. http://developer.android.com/guide/appendix/glossary.html.

[7] Hadoop. https://hadoop.apache.org/.

[8] Issue 10255: Adb hangs intermittently. http://code.google.com/p/android/issues/detail?id=10255.

[9] Issue 1946: javax.net.ssl.sslexception: Not trusted server certificate. http://code.google.com/p/android/issues/detail?id=1946.

[10] Issue 38315: Devices are going in offline state in "adb devices" after random time. http://code.google.com/p/android/issues/detail?id=38315.

[11] Monkeydevice. http://developer.android.com/tools/help/MonkeyDevice.html.

[12] Monkeyrunner. https://developer.android.com/tools/help/MonkeyRunner.html.

[13] Pickers. https://developer.android.com/design/building-blocks/pickers.html.

[14] Robotium. https://code.google.com/p/robotium/.

[15] Services. https://developer.android.com/guide/components/services.html.

[16] Spinners. https://developer.android.com/design/building-blocks/spinners.html.

[17] Ui/application exerciser monkey. https://developer.android.com/tools/help/monkey.html.

[18] Httpclient - httpclient ssl guide. http://hc.apache.org/httpclient-3.x/sslguide.html, 2008.

[19] Mallory: Transparent tcp and udp proxy. http://intrepidusgroup.com/insight/mallory/, 2010.

[20] android: Single activity, multiple views. http://stackoverflow.com/questions/10862052/android-single-activity-multiple-views, 2012.

[21] Burp suite. http://www.portswigger.net/burp/, 2012.

[22] Class allowallhostnameverifier. http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/conn/ssl/AllowAllHostnameVerifier.html, 2012.

[23] smali. http://code.google.com/p/smali/, 2012.

[24] ARTZI, S., DOLBY, J., JENSEN, S. H., MØLLER, A., AND TIP, F. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering* (2011).

[25] AYUSO, P. N. netfilter/iptables project homepage - the netfilter.org "iptables" project. http://www.netfilter.org/projects/iptables/.

[26] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (2003).

[27] CLARK, J., AND VAN OORSCHOT, P. C. Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements. *2013 IEEE Symposium on Security and Privacy 0* (2013), 511–525.

[28] DESNOS, A. androguard. https://code.google.com/p/androguard/.

[29] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010).

[30] FAHL, S., HARBACH, M., MUDERS, T., SMITH, M., BAUMGÄRTNER, L., AND FREISLEBEN, B. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *19th ACM Conference on Computer and Communications Security* (2012).

[31] FELT, A. P., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11* (2011).

[32] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *19th ACM Conference on Computer and Communications Security* (2012).

[33] HU, C., AND NEAMTIU, I. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011).

[34] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res. 7* (2006).

[35] OCTEAU, D., ENCK, W., AND MCDANIEL, P. The ded Decompiler. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, 2010.

[36] PORTOKALIDIS, G., HOMBURG, P., ANAGNOSTAKIS, K., AND BOS, H. Paranoid Android: Versatile Protection For Smartphones. In *Annual Computer Security Applications Conference* (2010).

[37] RASTOGI, V., CHEN, Y., AND ENCK, W. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Third ACM Conference on Data and Application Security and Privacy* (2013).

[38] TAM, D. Google forecasts 70 million android tablet activations by year's end. http://news.cnet.com/8301-1023_3-57595262-93/google-forecasts-70-million-android-tablet-activations-by-years-end/, 2013.

[39] TEITELBAUM, D. Posts tagged 'viewserver'. http://blog.apkudo.com/tag/viewserver/, 2012.

[40] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium* (2012).

[41] ZHENG, C., ZHU, S., DAI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (2012).

[42] ZHOU, W., ZHOU, Y., JIANG, X., AND NING, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012).

[43] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *19th Network and Distributed System Security Symposium* (2012).