Interactive Visualization Toolbox to Detect Sophisticated Android Malware*

Ganesh Ram Santhanam Benjamin Holland Suresh Kothari Jon Mathews
Department of Electrical and Computer Engineering, Iowa State University
Ames, IA 50011 {gsanthan,bholland,kothari}@iastate.edu
EnSoft Corp. Ames, IA 50010
jmathews@ensoftcorp.com

Abstract

Detecting zero-day sophisticated malware is like searching for a needle in the haystack, not knowing what the needle looks like. This paper describes Android Malicious Flow Visualization Toolbox that empowers a human analyst to detect such malware. Detecting sophisticated malware requires systematic exploration of the code to identify potentially malignant code, conceiving plausible malware hypotheses, and gathering evidence from the code to prove or refute each hypothesis. We describe interactive visualizations of program artifacts to understand and analyze complex Android semantics used by an app. The toolbox incorporates visualization capabilities that work together cohesively, and provides a mechanism to easily add new capabilities.

We present case studies of detecting Android malware with confidentiality and integrity breaches. We report the accuracy and efficiency achieved by our team of analysts by using the toolbox, while auditing 77 sophisticated Android apps provided by Defense Advanced Research Projects Agency (DARPA).

Toolbox URL: `https://kcsl.github.io/AMFVT/`

1 Introduction

Sophisticated zero-day malware do not conform to the signature of known malware. Sophisticated malware are often designed to enact catastrophic consequences, and are targeted at a nation-state or community, e.g., Flame [26], Stuxnet [8], and Android malware versions of the Kakao Talk messaging app [6, 7].

To the best of our knowledge, automated detection approaches using program analyses [30] and machine learning [11] cannot work for sophisticated zero-day Android malware, with no signature or samples. A purely manual approach is extremely laborious due to the complex semantics of Android APIs and their varying combinations. The need for a collaborative approach bringing together human intelligence and machine automation is the key for detecting sophisticated malware [5]. As Frederick Brooks articulates, complex problems require automation to amplify human intelligence [14].

This paper presents a novel human-on-the-loop approach [10] to detect sophisticated Android malware. The approach consists of three steps: (a) automatic pre-computation of data and control flows within an app and between the app and Android, (b) generate hypotheses as to how the app could manifest malicious behavior, using interactive visualizations to understand the relationships between app's program artifacts and Android, and (c) gather graphical evidence of the app's semantics that validates or refutes hypotheses. The steps (b) and (c) may be iteratively performed by an analyst until either a hypothesis is validated or all of them have been refuted.

*This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

A human analyst is faced with two specific challenges when detecting sophisticated Android malware. First is the open-ended search for plausible hypotheses of malicious behavior. The space of hypotheses can be arbitrarily large for a given app. With no prior knowledge of an app's malicious behavior, the analyst has to rely on a general model such as the CIA (confidentiality, integrity, availability) model [22] to systematically explore hypotheses related to malicious behaviors. In particular, an analyst may consider hypotheses related to confidentiality leaks (e.g. GPS location is leaked to an adversary), integrity breaches (e.g. incorrect GPS location is displayed in some geographic locations), or denial of service (e.g. malware runs loops to drain the device battery) attacks enabled by unscrupulous use of Android APIs. However, this can still be prohibitively expensive, as there may be numerous potential CIA hypotheses to explore for a given app. To precisely explore only the relevant CIA hypotheses, it is *critically important for the analyst to visualize and understand the data and control flow interactions between the app and the Android components it uses*. For example, knowing that an app interacts with the hardware and messaging APIs via data flow leads to the hypothesis that it may leak camera images by sending them as attachments with messages.

Once a hypothesis is generated, the second challenge for the analyst is to be able to gather evidence that validates or refutes the hypothesis. This is challenging because sophisticated malware typically contain several scattered and cleverly hidden components. The Android API stack consists of several layers, and its semantics is very complex. This is further complicated by several asynchronous call back mechanisms in the form of Intents, message handlers and broadcast receivers [18], and Java's exceptional flows. Piecing together the evidence required to identify the *modus operandi* of sophisticated malware requires analyst to be able to on-demand visualize a composition of data, control, and Android-specific flows of an app. Therefore, *tools that enable analyst to visualize and compose artifacts that can serve as evidence of the hypothesized malicious behavior are crucial* to ensure the efficiency of detection.

To address the above challenges, this paper presents the *Android Malicious Flow Visualization Toolbox* [3], a suite of interactive visualization capabilities that work together cohesively to enable the human-on-the-loop approach. The toolbox consists of interactive visualizations called *smart views* built on top of Atlas [17], which enable an analyst to generate and validate hypotheses about malicious behavior in an Android app. Specifically, smart views enable analysts to: (a) visualize interactions of the app with various subsystems of Android via data and control flow, (b) visualize how exceptions thrown and caught by the app may amount to malicious behavior by the agency of Java's exception flow semantics, and (c) visualize how sensitive data may be modified by the app leading to an integrity breach. Smart views generate human consumable evidence in the form of program graphs whose nodes and edges are the app's artifacts and their syntactic and semantic relationships. Smart views are interactive – an analyst can click on a specific artifact in the code, and the smart view is automatically recomputed and displayed corresponding to the new selection. The smart views are also composable, thus allowing an analyst to piece together a variety of evidence to validate hypotheses.

We illustrate using case studies how a human analyst uses the Android Malicious Flow Visualization Toolbox to detect sophisticated malware that violate confidentiality and integrity. While an extensive study of the impact of our approach and tooling on human analysts is beyond the scope of this paper, we report on the average accuracy and speed of malware detection on 77 apps provided by DARPA [4] based on the performance of 3 analysts who are knowledgeable about the Android system.

To summarize, our contributions include:

- Human-on-the-loop approach to detect sophisticated Android malware
- Android Malicious Flow Visualization Toolbox containing interactive and composable *smart views* to enable human analysts to hypothesize and gather evidence of malicious behavior in an Android app
- Case studies illustrating the use of our approach and the toolbox by human analysts to detect sophisticated malware involving confidentiality and integrity breaches

2 Human-on-the-loop Malware Detection Approach

This section describes a "human-on-the-loop" approach [24] to detect sophisticated Android malware. The analyst's role in this approach is to use knowledge about the Android system to review an app's interactions with Android libraries, hypothesize plausible malicious behaviors, and gather evidence to validate or refute them.

2.1 Three-Phase Malware Detection

According to the widely accepted CIA model (industry standard for information security [30]), an app can be considered to be malware if it violates any of the three key principles: confidentiality, integrity and availability. To decide whether an app violates CIA principles, the analyst has to answer specific questions related to each of them. For instance, to determine whether an app has a confidentiality leak, the questions would be: (a) What could be the sensitive information? (b) How could the sensitive information be leaked (e.g., by writing to a SD card or sending it through internet)? (c) What triggers the leak? Similarly, the questions to detect integrity violations would be: What information should be considered critical and immutable to ensure integrity? How could the information get corrupted? What triggers the corruption? Similar questions can be asked to detect availability violations: What resources could be exhausted maliciously (e.g., battery drained by running an unnecessary loop)? How could the resource exhaustion take place? What triggers the resource exhaustion?

To help the analyst answer such questions, we describe a three-phase detection approach.

Phase I: ***Automated exploration***. The objective of the first phase is to precompute information that serves as the basis for the analyst to begin the investigation. The precomputed information includes the data, control and exceptional flows within the app, the permissions granted to the app and the APIs used to exercise them, and program artifacts that use the Android resource files. The precomputed information also includes the call and data flow interactions of the app with various Android *subsystems*. Subsystems are logical groupings of Android APIs according to the functionality they provide such as networking, storage device access, address book, etc. (see Section 3.1 for details). The interactions provide high level information about how the app interacts with Android components.

At the end of Phase I, this precomputed information is stored in the form of program graphs of syntactic and semantic (data, control or call) relationships between app's artifacts and Android. Parts of this precomputed program graph can be queried and visualized by the analyst using smart views (details in Section 3) in Phase II.

Phase II: ***Hypothesis formulation***. The objective of the analyst here is to develop hypotheses about how the app could violate confidentiality, integrity or availability properties. As examples, the hypotheses for confidentiality leak could be: "*the app leaks GPS location information through internet*" or "*the app leaks preview images from camera to the internet*."

To aid the formulation of relevant hypotheses, the analyst should know which Android subsystem the app interacts with, and whether the interaction is via data flow, control flow, or call graphs. The analyst uses the *Subsystem Interaction* smart view to query and visualize this information for each subsystem. The analyst may flag the app's use of certain Android subsystems as unwarranted based on knowledge of the app's intended functionality. For example, an analyst may consider the interaction of a calculator app with the Network subsystem or the Media subsystem to be suspicious. The analyst can then hypothesize that "*The calculator app writes sensitive information to the network*" or "*The calculator app deletes sensitive information from the SD card*".

By the end of Phase II, the analyst has malware hypotheses that need to be either validated or refuted.

Phase III: ***Gathering evidence to validate hypotheses***. The objective of the analyst here is: (1) to gather evidence that validates the hypotheses formulated in Phase II, and (2) to compose together pieces of evidence to identify the overall *modus operandi* of the malware, i.e., how CIA properties are violated.

To help the analyst gather evidence to validate the hypotheses, we provide two smart views: an *Exceptional Flow* smart view, and an *Integrity Checker* smart view. This *Exceptional Flow* smart view matches `throw` or `catch` blocks corresponding to a selected `catch` or a `throw` site in the app. This is crucial for an analyst because exceptional flows are control transfers implicitly performed by the JVM's exception handling mechanism, and are not visible in the control flow graph. This smart view enables the analyst to gather evidence needed to validate hypotheses that malicious flows use the exception flow mechanism. The *Integrity Checker* smart view allows the analyst to select a data flow artifact such as a variable in the app, and displays the set of program locations in the app that modify it along with the conditions governing each modification. This is useful to gather evidence needed to validate hypotheses related to integrity violations. For example, if the analyst hypothesizes that certain variables or types should not be modified according to the functionality of the app, this smart view shows whether and under what conditions the app modifies them.

In the final step of Phase III, the analyst manually composes the individually gathered evidence into an overall sequence of events that prove that the app violates confidentiality, integrity or availability. At the end of Phase III, if the analyst is unable to compose such a flow using the gathered evidence, the analyst rejects the hypothesis.

Iteration of phases. Given the complexity of Android semantics and that of the analyzed app, the analyst may come up with several malware hypotheses. Phases II and III may be iteratively performed for each hypothesis until one is validated; otherwise if all hypotheses are refuted the app is declared benign.

2.2 Illustration of Three-Phase Malware Detection

Consider an app that scans barcodes and looks up product information from the internet. Phase I computes the interactions of the app with various Android subsystems. In Phase II, the analyst may observe that the app invokes APIs from the Android messaging subsystem and Android network subsystem. After inspecting the relevant code causing the interactions, the analyst may identify that the app accesses the URL `http://www.malwareforsure.com`, which is suspicious. The analyst hypothesizes that the app may leak out sensitive information through the URL and thus violate confidentiality. In Phase III, the analyst gathers the following pieces of evidence: (i) the app consumes messages posted to the Android messaging subsystem in method m_1, (ii) m_1 throws an exception containing the message contents while consuming messages from Android, which is caught in another method m_2, and (iii) m_2 sends data from the exception object to the internet, and (iv) the app posts a message to the Android messaging subsystem containing the camera preview images. At the end of Phase III, the analyst composes the gathered evidence to show that the sequence of events (iv) $\rightarrow$ (i) $\rightarrow$ (ii) $\rightarrow$ (iii) is indeed malicious.

3 ANDROID MALICIOUS FLOW VISUALIZATION TOOLBOX

We present the Android Malicious Flow Visualization Toolbox for detecting sophisticated Android malware. We have developed this toolbox in addition to the previously reported research on the Android Security Toolbox [23]. The Android Malicious Flow Visualization Toolbox is an Eclipse plug-in built using the Atlas platform [17]. The toolbox can be used to analyze Android, Java source code and Java byte code applications.

Smart views. The Android Malicious Flow Visualization Toolbox includes three interactive visual models called *smart views*: (1) a *Android subsystem interaction* smart view showing the interaction between the app and different parts of the Android system, (2) an *exception flow* smart view showing the correspondence between matching throw and catch locations, and (3) an *integrity checker* smart view showing the conditions under which a predefined set of Android-specific or app-specific immutables are modified by the app.

Smart views are visual graph models, whose nodes and edges correspond to syntactic and semantic relationships of of a subset of app's artifacts and Android. Smart views accept a selected source code element or graph element from another visual model as input, and produce a visual model as output. Smart view are interactive, i.e., are automatically updated when the analyst changes the selection. They are composable – an analyst can select a graph element from the output of a smart view and feed it as the input to another smart view. Smart views provide two-way source correspondence, so the user can click on a node or an edge in the visual model to go to the corresponding source code artifact (control flow block, variable, etc.). The smart views in the Android Malicious Flow Visualization Toolbox seamlessly compose and integrate with built-in Atlas smart views that allow analyst to explore classical forward/reverse data flow for a selected data flow node (e.g., parameters, variables), control flow for a selected control flow block, and call graph for a selected method. To scale visualizations to large visual graph models, smart views provide interactive controls for incrementally unfolding, zooming and panning the graph models.

3.1 Android Subsystem Interaction View

Partitioning the app's interaction with Android. The Android operating system is large, consisting of the Linux kernel plus about 2 million lines of Java code spread across more than 200 packages and close to 5000 classes [2]. The analyst can get lost with hundreds of Android APIs an app may interact with. This visual model provides a solution by a logical partitioning of the Android API (at the package level) consisting of 19 *subsystems*. Each subsystem is mapped to a set of packages of libraries commonly used by Android apps. Table 1 shows our partitioning scheme. The result for each subsystem consists of two bipartite graphs: (1) a *call interaction view* showing all calls between the app and the Android subsystem, and (2) a *data flow interaction view* showing all data flows between the app and the Android subsystem. For the call interaction view, the nodes of the bipartite graph consist of methods in the app and the methods in the Android subsystem, and the edges represent calls between them. For the data flow interaction view, nodes consist of variables, parameters, return values, etc., in the app and the Android subsystem, and the edges represent data flows between them.

Subsystems vs. Permissions. Android's permission framework maps APIs to permissions. However, there are two reasons why permissions are inadequate and subsystems are needed to divide the Android APIs into logical groups with related functionality.

Subsystem	**Description**	**A**	**J+o**	**T**
Accounts	Device accounts	1	0	1
Administration	Device administration	1	0	1
App Resources	App manifest, R file, etc.	2	0	2
Core	Android core OS	10	0	10
Crypto	Cryptography libraries	0	13	13
Database	Database libraries	2	2	4
Hardware	Hardware libraries	4	0	4
Introspection	Reflection & runtime libraries	2	1	3
IO	I/O & serialization libraries	3	23	26
Java	Other Java language libraries	0	4	4
Location	Location based	2	0	2
Media	Media libraries	5	0	5
Network	Network IO libraries	14	32	46
Preference	Device preferences	1	0	1
Provider	Provider interfaces	1	0	1
Support	Android support libraries	24	0	24
Test	Mock interface & test libraries	3	2	5
UI	User interface libraries	19	3	22
Util	Android & Java utilities	2	10	12

Table 1: *Logical subsystem partitioning:* The number of packages within Android (A) and Java or other commonly used libraries (J + o) that are mapped to the subsystem category are given in columns 3 and 4 respectively; column 5 shows the total.

1. Permissions do not cover all the Android APIs [12] i.e., some Android APIs do not require a permission.
2. There are close to 200 permissions defined by Android (the number continues to grow with new versions of Android), and additionally an app can define further permissions. The large number of permissions provides low-level details, but also makes it difficult for an analyst to hypothesize malicious behavior in terms of higher level abstractions such as violation of the CIA principles.

In contrast to permissions, our subsystems listed in Table 1 cover all the Android APIs, i.e., every Android API can be mapped to one of the 19 subsystems. The subsystem views provide a higher level of abstraction than mapping to permissions.

Analyst interaction. From the *call interaction view*, the analyst can click on the methods in the app or Android to jump to the corresponding source code. The analyst can click on the edges to jump to the call site in the source code. From the *data flow interaction view*, the analyst can similarly click on variables, method parameters, return values etc. to jump to the corresponding source code location. The visual models do not display Android components that don't interact with the app. This keeps the analyst focused on the part of the Android subsystem that is relevant to the app's behavior. Figure 1 shows the call interaction of an app with Android's network subsystem. How this is useful in detecting malware is explained using a case study in Section 4.1.

3.2 Exception Flow View

Importance of reasoning about exceptional flows. Exceptional flow information is critical from the perspective of detecting malicious flows for two reasons. First, data carried in the exception object could be sensitive. Second, the exception flow may exercise malicious paths not visible through the control flow graph.

Analyst interaction. The exceptional flow smart view is a visual model that allows the analyst to view matching `throw` and `catch` blocks according to Java's exception flow semantics. Nodes correspond to `throw` and `catch` events in the code. Edges represent the transfer of control from the `throw` to the corresponding `catch` blocks according to Java's exceptional handling semantics. This visual model instantly updates in response to analyst's selections in the source code. If the analyst selects a catch block, the visual model shows potential throw blocks connected to the selected catch block. For convenience, it also allows the analyst to select an entire method to select all the throw and catch blocks in it and their respective connecting blocks. Figure 2 shows an example of this visual model when a method is selected. How it helps an analyst in malware detection will be described in a case study in Section 4.1.

3.3 Integrity Checker View

Detecting integrity breaches. An integrity breach occurs when an app modifies critical data (a.k.a. *immutable*) that should not be ordinarily modified unless an app's functionality warrants it. Integrity may be breached in two ways: (1) malware may directly modify an immutable (e.g., overwrite contact details), or (2) malware may modify the algorithm that computes the immutable (e.g., algorithm that computes checksum for a file).

We have categorized immutables in Android into data artifacts(e.g., variables) related to system location, device preferences, system settings, secure system settings, device sync state, user dictionary, and all Android widgets (`android.view.View` and its subclasses). In addition, variables computed by the app (e.g., checksums, hash functions) can also affect the app's performance, reliability and trustworthiness, and should be considered immutable.

Analyst interaction. We have developed an *integrity checker* smart view that shows all program locations that modify a selected *Android-specific* immutable or *app-specific immutable*, along with the branch conditions under which they are modified. On selection of an immutable, the view internally invokes analyses that uses the data flow graph to reach the assignments to the immutable, and then identifies the branch conditions under which those assignments occur using the control flow graph. The view then updates with the paths involving these branches and assignments. These branch conditions are important because the path in which the variable is modified can be crucial information for the analyst to decide whether the modification is legitimate given the app's functionality. The analyst can invoke the visual model with respect to a specific category of Android-specific immutable, such as location, system settings, etc. or with respect to an app-specific immutable. Figure 6 and Figure 7 show the visual model produced on the selection of an Android-specific and app-specific immutable respectively. Section 4.2 illustrates case studies in which these serve as critical pieces of evidence to validate malware hypotheses.

4 Case Studies for Evaluation

The Android Malicious Flow Visualization Toolbox cannot be evaluated independent of the analyst's capability to detect sophisticated malware. Therefore, we illustrate how the Android Malicious Flow Visualization Toolbox helped human analysts during malware detection using case studies of confidentiality leak and integrity breaches.

The case studies presented here are drawn from a repository of 77 Android apps provided by DARPA, which were designed to be representative of real world apps. Of the 77 apps, 62 contained novel and sophisticated malware designed to evade automatic, signature-based detection techniques. Their average size was 7000 lines of code, with a maximum of 70,000 lines of code.

An extensive human-subject experimental study of the impact of our approach and toolbox on analysts is beyond the scope of this paper. Nevertheless, to illustrate the practical feasibility of our human-on-the-loop approach and the utility of the toolbox, we include here the average accuracy and speed of malware detection (classifying an app as malware or benign) achieved by 3 different analysts, and the scalability of our toolbox for the 77 apps before proceeding to the case studies. The analysts for this study included a graduate student, a research scientist, and an experienced software engineer from the industry. The target groups of our system are security analysts with knowledge about the Android system.

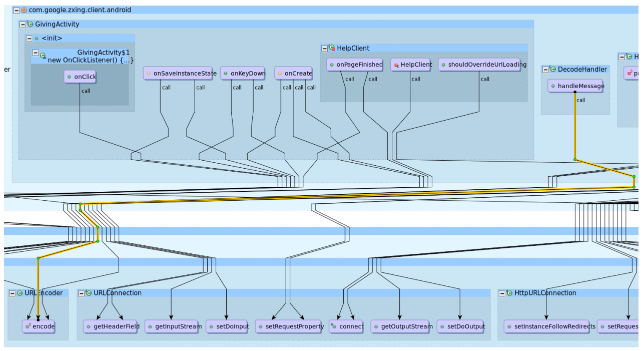

Figure 1: Network subsystem interaction visual model with calls: Top layer shows app methods that call Android APIs. Bottom layer shows Android and Java networking APIs.

1. **Time:** Each analyst required 1.13 hours per app on average using our approach and the toolbox. The maximum analyst time for an app was about 12 hours.
2. **Accuracy:** The analysts using the toolbox as part of the human-on-the-loop approach correctly classified 66 (85.7%) apps as malicious or benign, found unintended malicious behaviors in 6 (7.8%) apps, and missed malicious behaviors in 5 (6.5%) of the apps.
3. **Scalability:** Our toolbox was able to dynamically recompute information for the smart views within few seconds even for the largest app with 70,000 lines of code.

4.1 Confidentiality Leak

We illustrate how a confidentiality leak in the *BarcodeScanner* app is detected using Android Malicious Flow Visualization Toolbox .

App functionality: The *BarcodeScanner* app (1) Scans barcodes images on products using the camera, (2) Looks up the product information for the scanned barcode from the internet, and (3) Displays the looked up information to the user. The app consists of 68 Java files with 6307 lines of code.

4.1.1 Phase I, Automated Exploration

The Android permissions and interactions between the BarcodeScanner app and the Android subsystems are precomputed by the Android Malicious Flow Visualization Toolbox (see Section 3.1).

4.1.2 Phase II, Hypothesis Generation

The analyst surveys the interactions between the app and the Android subsystems using the Subsystem Interaction smart view to get a high level understanding of the app's activities. The analyst observes the interactions of the app with the Camera subsystem and Network subsystem. The analyst then wants to check if the access to internet is used for the legitimate purpose only. By looking at the Subsystem Interaction smart view with respect to the network subsystem (see Figure 1), the analyst finds a call from the method `DecodeHandler.handleMessage` in the app to `URLEncoder.encode(String)` within the `java.net` package. The API converts strings to a URL format to send it to an internet server. The analyst wants to know:

Q1. *What information does `DecodeHandler.handleMessage` encode in the URL using `URLEncoder.encode`?*

To probe deeper, the analyst clicks on the `handleMessage` method in the smart view. This brings up the source code for the method (Listing 1). The analyst observes

a. The call site to `URLEncoder.encode` occurs within a catch block for `DecodeFormatException`.
b. The URL uses data from the exception object.

These observations lead the analyst to the following hypothesis.

Hypothesis 1 *BarcodeScanner app leaks sensitive data into the internet in* `DecodeHandler.handleMessage` *via an exception flow involving* `DecodeFormatException`*.*

Listing 1: `DecodeHandler.handleMessage`

```
@Override
public void handleMessage(Message message) {
  ...
  switch (message.what) {
  case decode:
    try {
        validateAndDecode((byte[]) message.obj,
            message.arg1, message.arg2);
    } catch (DecodeFormatException e) {
      try {
        HttpHelper.downloadViaHttp(
          "http://www.google.com/search?q="
            + URLEncoder.encode(
                e.getLocalizedMessage(), "utf-8"),
                  ...);
      } catch (java.lang.Exception e1) {
        ...
      }
    }
    break;
  case quit: ...
  }
}
```

The app could load sensitive payload into exception objects `DecodeFormatException`, throw the exception object at one program location in the app, catch the exception in `DecodeHandler.handleMessage`, then leak the payload to the internet.

Messages are posted and consumed asynchronously using the `Message.sendToTarget` and `MessageHandler.handleMessage` APIs respectively. The control and the data flow through the messages is not visible in the app. The analyst becomes suspicious because of the invisibility of flows to carry the payload from its point of production to the launch site to send it to an Internet server. The indirections through an exception and a message handler make the program difficult to understand; if they have a legitimate purpose, it is not clear to the analyst. So, the analyst probes further.

Hypothesis 2 *A program location within BarcodeScanner encapsulates sensitive information in a message and posts it using* `Message.sendToTarget`*. The information is subsequently asynchronously consumed by the message handler* `DecodeHandler.handleMessage`*.*

The above two hypotheses, if validated, could be composed by the analyst to prove that the app has a confidentiality leak. Hence, the analyst moves to Phase III to validate the hypotheses.

4.1.3 Phase III, Validating the Hypotheses

Gathering Evidence to Validate Hypothesis 1. To validate Hypothesis 1, the analyst needs to answer the following questions:

Q2. *Which program locations throw an exception of type* `DecodeFormatException`*?*

Q3. *Does the exception object contain sensitive information?*

The analyst conducts an investigation to get the answer to Q2. To comprehend the exception flows propagated through `DecodeHandler.handleMessage`, the analyst invokes the *Exception flow smart view* as described in Section 3.2. The resulting visual model shown in Figure 2 reveals that `DecodeFormatException` is thrown in exactly one location, namely the method `DecodeHandler.decode`. The analyst clicks on the throw location to go to the source code of the method `DecodeHandler.decode` shown in Listing 2. The analyst finds that the `DecodeFormatException` object thrown here is constructed using the contents of the variable `data`.

Next, the analyst addresses the question Q3. The analyst selects the `data` and opens the reverse data flow smart view[1] (see Figure 3). This smart view shows that the `data` flowing into the thrown `DecodeFormatException` object actually comes from the parameter passed to the `DecodeHandler.decode` method, which is called from the method `DecodeHandler.validateAndDecode` (Listing 3), which in turn receives the contents of an `android.os.Message` cast as `byte[]`. Thus, the analyst validates Hypothesis 1 by gathering evidence to show that the exception object encapsulates a payload that came through a message.

[1]Smart views for visualizing forward and reverse data and control flow graphs with respect to a selected data or control flow node are part of the Atlas platform [17].

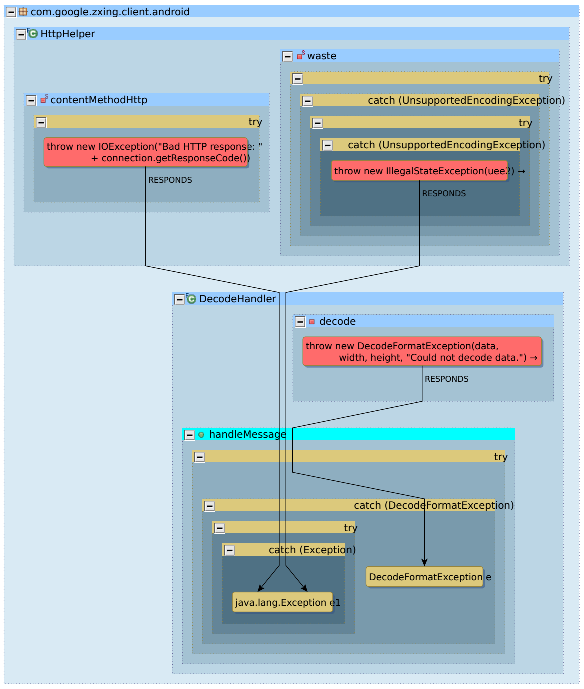
com.google.zxing.client.android
HttpHelper
waste
try
catch (UnsupportedEncodingException)
try
catch (UnsupportedEncodingException)
throw new IllegalStateException(uee2) →
RESPONDS
contentMethodHttp
try
throw new IOException("Bad HTTP response: " + connection.getResponseCode())
RESPONDS
DecodeHandler
decode
throw new DecodeFormatException(data, width, height, "Could not decode data.") →
RESPONDS
handleMessage
try
catch (DecodeFormatException)
try
catch (Exception)
java.lang.Exception e1
DecodeFormatException e

Figure 2: Exception visual model generated by clicking `DecodeHandler.handleMessage`. The model shows the correspondence between throw and catch blocks using connecting edges labeled "RESPONDS."

Listing 2: `DecodeHandler.decode`

```
private void decode(byte[] data, int width, int height)
      throws DecodeFormatException {
  ...
  Handler handler = activity.productionHandler();
    if (...) {
       // scanner produces valid barcode image
       Message message = Message.obtain(handler,...);
       Bundle bundle = ... // get barcode image data
       message.setData(bundle);
       message.sendToTarget(); // send barcode
    } else {
        // scanned image is not valid barcode
       if (handler != null) {
        // send: captured image isn't valid barcode
        Message msg = Message.obtain(handler, ...);
        msg.sendToTarget();
        throw new DecodeFormatException(data, width,
            height, "Could not decode data.");
       }
    }
}
```

Listing 3: `Decode.validateAndDecode`

```
private void validateAndDecode(byte[] data, int width, int
     height)
    throws DecodeFormatException {
    if(...) {// data is formatted and decodable
      decode(data, width, height);
    }
}
```

Gathering Evidence to Validate Hypothesis 2. To validate Hypothesis 2, the analyst needs to know the callers of `Message.sendToTarget`.

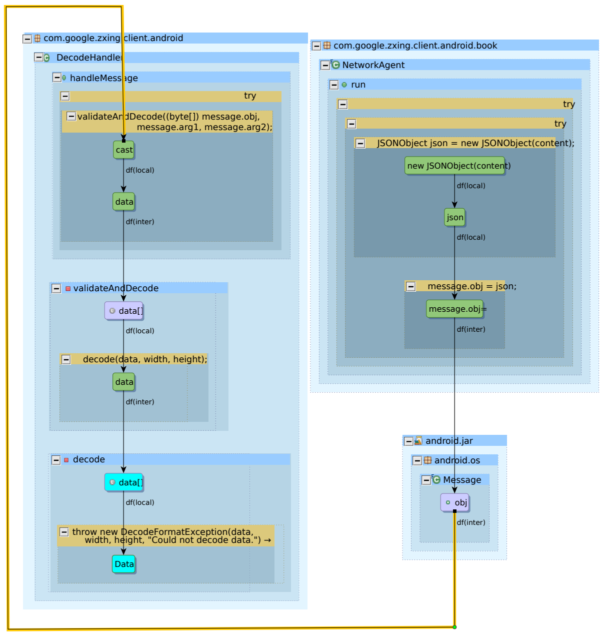
com.google.zxing.client.android
DecodeHandler
handleMessage
try
validateAndDecode((byte[]) message.obj, message.arg1, message.arg2);
cast
df(local)
data
df(inter)
validateAndDecode
data[]
df(local)
decode(data, width, height);
data
df(inter)
decode
data[]
df(local)
throw new DecodeFormatException(data, width, height, "Could not decode data.") →
Data
com.google.zxing.client.android.book
NetworkAgent
run
try
try
JSONObject json = new JSONObject(content);
new JSONObject(content)
df(local)
json
df(local)
message.obj = json;
message.obj=
df(inter)
android.jar
android.os
Message
obj
df(inter)

Figure 3: Reverse data flow graph smart view

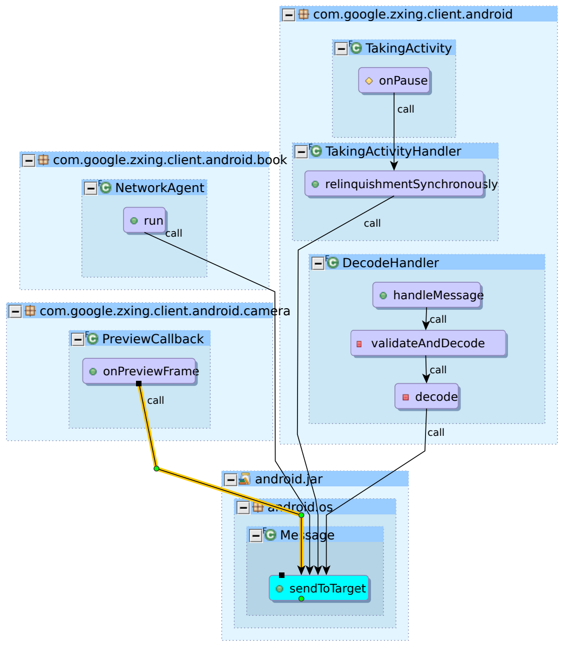

Figure 4: Reverse call graph smart view

The analyst opens the reverse call graph smart view (provided by Atlas [17]). From the reverse call graph, shown in Figure 4, the analyst observes a call to `Message.sendToTarget` from `PreviewCallback.onPreviewFrame` (Listing 4), which is an Android API `Camera.PreviewCallback.onPreviewFrame` overridden by the app. The `PreviewCallback` interface is used to deliver copies of preview frames as they are displayed [1]. In the smart view, the analyst clicks on `PreviewCallback.onPreviewFrame` to bring up its source code.

Listing 4: `PreviewCallback.onPreviewFrame`

```
@Override
public void onPreviewFrame(byte[] data, Camera camera) {
  Point resolution = ... // obtain camera resolution
  Handler previewHandler = .. // obtain preview handler
  if (resolution != null && previewHandler != null) {
    // send a message with preview image from camera
    Message message = previewHandler.obtainMessage(
        previewAdvice, resolution.x, resolution.y, data);
    message.sendToTarget();
    previewHandler = null;
  } else {
    // log message: No handler available for preview callback
  }
}
```

Inspecting `PreviewCallback.onPreviewFrame` shows that each preview image from the camera is indeed encapsulated in a `Message` object and posted to the Android messaging system via `Message.sendToTarget`. Evidence from Figure 3, Figure 4, and the corresponding code segments together validate Hypothesis 2.

Composing evidence to detect sophisticated malware. As the final step in Phase III, the analyst composes the evidence gathered for Hypothesis 1 and Hypothesis 2 to document the full operation of the malware. BarcodeScanner leaks confidential information from the camera as shown in Figure 5. Its full operation is as follows. The app overrides `Camera.PreviewCallback.onPreviewFrame` to asynchronously post a message containing camera preview images. This message is asynchronously consumed by `DecodeHandler.handleMessage`, which eventually (via method `DecodeHandler.validateAndDecode`) calls `DecodeHandler.decode`, which in turn throws the `DecodeFormatException` object containing sensitive camera data. This exception is caught again in `DecodeHandler.handleMessage`, packaged using `URLEncoder.encode` and leaked to the internet. This concludes the analyst's audit.

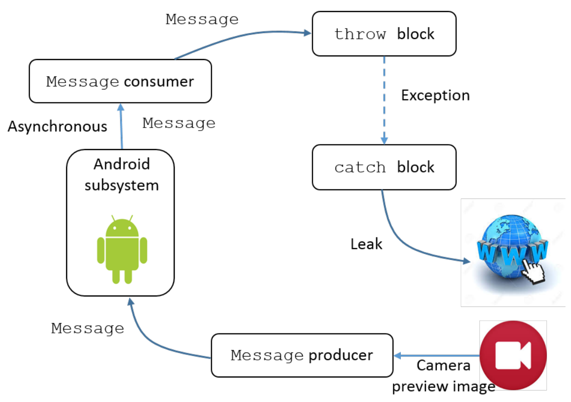

Figure 5: Confidentiality leak from camera preview image to the internet is hidden by BarcodeScanner via asynchronous flows through Android messaging passing and exception flows.

4.2 Integrity Breach

We illustrate the use of Android Malicious Flow Visualization Toolbox for integrity breaches in two apps: the *TRAQ* app to show a compromise of an Android-specific immutable and the *CalcC* app to show a compromise of an app-specific immutable. The toolbox incorporates Android-specific immutables irrespective of any app. The analyst determines the app-specific immutables for a particular app taking into account its functionality. We show how the integrity checker smart view (see Section 3.3) helps the analyst detect integrity breaches in these apps. We do not describe the steps an analyst takes to formulate hypotheses.

4.2.1 Detecting Modifications to Android Immutables

App description. **TRAQ** is a data gathering and relaying app. It allows for planning of strategic missions as well as audio and video recording and taking geo-tagged snapshots through the camera based on the GPS location. It contains 422 Java files (63083 lines of code).

From the description, the analyst infers that the GPS location reported by the app is critical to its integrity, as all functionalities rely on its correctness. The analyst then invokes the *integrity checker*

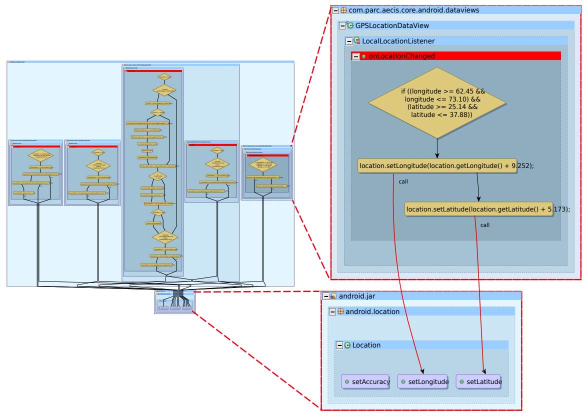
com.parc.aecis.core.android.dataviews
GPSLocationDataView
LocalLocationListener
onLocationChanged
if ((longitude >= 62.45 && longitude <= 73.10) && (latitude >= 25.14 && latitude <= 37.88))
location.setLongitude(location.getLongitude() + 9.252);
call
location.setLatitude(location.getLatitude() + 5.173);
call
android.jar
android.location
Location
setAccuracy
setLongitude
setLatitude

Figure 6: Integrity check smart view with Android-specific immutable: GPS location integrity is critical for the TRAQ app. Under certain conditions (magnified, top), the app corrupts the Android GPS location (magnified, bottom).

smart view (see Section 3.3) in the Android Malicious Flow Visualization Toolbox specifically for the predefined Android immutables for GPS location. Figure 6 shows the latitude and longitude variables in `android.location.Location`. The figure reveals five different program locations that write to these predefined immutable variables.

The analyst inspects the source code for each condition by clicking on the condition. One of the conditions under which these variables are modified (magnified in the figure) is clearly suspicious, as it modifies these immutables conditionally for only certain latitudes and longitudes (Listing 5). This condition corresponds to a geographical region at the border of Afghanistan and Pakistan. This serves as evidence to confirm the hypothesis that the integrity of Android GPS location information is compromised.

Listing 5: `LocalLocationListener.onLocationChanged`

```java
private class LocalLocationListener implements
     LocationListener {
  @Override
  public void onLocationChanged(Location location) {
    double latitude = location.getLatitude();
    double longitude = location.getLongitude();
    if((longitude >= 62.45 && longitude <= 73.10)
      && (latitude >= 25.14 && latitude <= 37.88)) {
    // Malware corrupts GPS location when
    // device is in Afghanistan or Pakistan
      location.setLongitude(location.getLongitude()
       + 9.252);
      location.setLatitude(location.getLatitude()
       + 5.173);
    }
    ...
  }
  ...
}
```

4.2.2 Detecting Modifications to App Immutables

App description. **CalcC** is a simple calculator app consisting of 2 Java files and 482 lines of code. From the description, the analyst decides that correct calculation is critical to the app's integrity.

The analyst wants to check if the integrity of calculator is compromised by providing incorrect results under certain conditions. After inspecting the code, the analyst determines that the variable `runningResult` should be treated as an app-specific immutable.

The analyst invokes the *integrity checker smart view* (Section 3.3) and selects the `runningResult` variable. The smart view in Figure 7 reveals that the app adds a random number to `runningResult` under certain conditions to corrupt the result (Listing 6).

This case study showed that to identify integrity breach, analysts may have to select app-specific immutables for integrity checks.

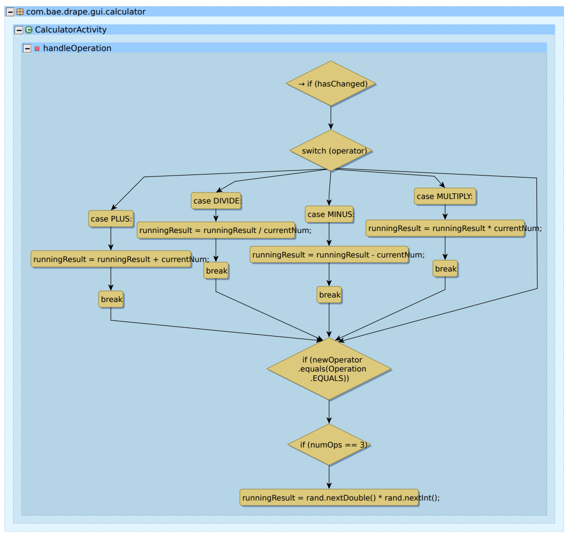
com.bae.drape.gui.calculator
CalculatorActivity
handleOperation
→ if (hasChanged)
switch (operator)
case PLUS:
case DIVIDE:
case MINUS:
case MULTIPLY:
runningResult = runningResult + currentNum;
runningResult = runningResult / currentNum;
runningResult = runningResult - currentNum;
runningResult = runningResult * currentNum;
break
break
break
break
if (newOperator .equals(Operation .EQUALS))
if (numOps == 3)
runningResult = rand.nextDouble() * rand.nextInt();

Figure 7: Integrity check visual model with App-specific immutable: `runningResult` is integrity critical for the CalcC app. Under certain conditions, the app corrupts `runningResult`.

Listing 6: `Calculator.handleOperation`

```java
private void handleOperation(Operation newOperator){
  if (hasChanged) {
    switch (operator) {
      case PLUS: ...
      case MINUS: ...
      case MULTIPLY: ...
      case DIVIDE: ...
    }
  if (...) {
    if (numOps == 3) {
      // Malware corrupts calculator result
    runningResult = rand.nextDouble() * rand.nextInt();
        }
    ...
    }
  }
}
```

5 Related Work

Automated analysis tools. Detecting Android specific malicious behavior is hard because it is often hidden amidst several other legitimate functionalities provided by Android. Hence, sophisticated one-of-a-kind Android malware are beyond the radar of detection approaches that rely on only traditional static and/or dynamic analysis [4, 20, 30]. Moreover, their practical applicability is limited because some of them require annotations [19], which are generally unavailable.

General comprehension and visualization tools. Existing tools for source code visualization [13] and program understanding such as Source Insight [9] come with a limited set of features that are inadequate for identifying sophisticated Android malware. The distinguishing aspects of our visual models is their specific relevance to Android, and their composability.

Analyst-friendly Android malware detection tools. The research papers that come closest to our approach and toolbox in helping analysts are: (1) MAMA [29], which analyzes the permissions from the app's Manifest file and the features, then uses machine learning to classify apps as malware, and (2) Anadroid [25], which performs static analysis to provide the analyst a list of semantic predicates that characterize an app's behavior. Neither MAMA nor Android offer visual models to assist the human analyst to visualize and spot malicious behavior.

Colluding Apps. Our toolbox also can also help detect colluding apps [15,28]. Colluding apps orchestrate malicious behavior through Android services such as Intents. As an example, an app can send sensitive information to another app using an Intent, and the app receiving the Intent can leak the information to the internet. Flows due to Intents are invisible because Intents are sent and received using asynchronous Android APIs. They follow a producer-consumer pattern similar to exception flows. We have also developed a smart view to visualize flows between apps via Intents, which is not described in this paper.

Other related work attempts to verify that an app's behavior adheres to its stated functionality using crowdsourcing [16], natural language processing [27], or clustering [21].

To the best of our knowledge, ours is the only toolbox that provides interactive visual graph models to understand an app's interactions with the Android subsystems, match corresponding throw and catch blocks via exceptional flows, and identify modifications of Android-specific and app-specific sensitive data.

6 Conclusion

Understanding an app's complex interactions with Android and the open-ended search for malicious functionality are the key challenges for detecting sophisticated Android malware. This paper presented a human-on-the-loop approach and associated Android Malicious Flow Visualization Toolbox to help human analysts detect sophisticated malware. In this approach, a human analyst formulates hypotheses about how malware could violate confidentiality, integrity or availability (CIA) properties, and gathers evidence to validate the hypotheses. Specifically we presented interactive visual models called smart views for visualizing the interaction of an app with Android subsystems, visualizing throw and catch events in the code are connected via exceptional flow semantics, and visualizing paths involving conditions under which an app modifies integrity critical variables in the app or Android.

We illustrated three case studies consisting of confidentiality and integrity breaches. These show how the toolbox helps an analyst to visualize malicious behavior in an app on-demand, and to compose them as evidence of violation of CIA properties. An important direction for future work is to perform user studies to systematically study the impact of our approach and toolbox on analysts of varying expertise and experience, using real world malware databases.

Acknowledgments

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

References

[1] Android api documentation for displaying camera previews. `http://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html`. Accessed: Jan. 2016.

[2] Android developer reference. `http://developer.android.com/reference/`. Accessed: Jan. 2016.

[3] Android malicious flow visualization toolbox. `https://kcsl.github.io/AMFVT/`. Accessed: Jul 2017.

[4] Darpa automated program analysis for cybersecurity. `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-63/listing.html`. Accessed: Jan. 2016.

[5] Darpa space/time analysis for cybersecurity. `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html`. Accessed: Jan. 2016.

[6] First-known targeted malware attack on android phones steals contacts and text messages. `http://www.forbes.com/sites/parmyolson/2013/03/26/first-known-targeted-malware-attack-on-android-phones-steals-contacts-and-text-messages/`. Accessed: Jan. 2016.

[7] Permission to spy: An analysis of android malware targeting tibetans. `https://citizenlab.org/2013/04/permission-to-spy-an-analysis-of-android-malware-targeting-tibetans/`. Accessed: Jan. 2016.

[8] The real story of stuxnet. `http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet`. Accessed: Jan. 2016.

[9] Source insight. `http://www.sourceinsight.com/`. Accessed: Jan. 2016.

[10] Supervisory automation – humans on the loop. `http://web.mit.edu/aeroastro/news/magazine/aeroastro5/cummings.html`. Accessed: Jul. 2017.

[11] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 21(1):183–211, Feb 2016.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 217–228. ACM, 2012.

[13] S. Bassil and R. Keller. Software visualization tools: survey and analysis. In *International Workshop on Program Comprehension*, pp. 7–17, 2001.

[14] F. P. Brooks, Jr. The computer scientist as toolsmith ii. *Communications of the ACM*, 39(3):61–68, Mar. 1996.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network and Distributed System Security Symposium*, 2012.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of Security & privacy in smartphones and mobile devices*, pp. 15–26. ACM, 2011.

[17] T. Deering, S. Kothari, J. Sauceda, and J. Mathews. Atlas: A new way to explore software, build analysis tools. In *International Conference on Software Engineering*, ICSE Companion, pp. 588–591. ACM, 2014.

[18] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security Privacy*, 7(1):50–57, 2009.

[19] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Conference on Computer and Communications Security*, pp. 1092–1104. ACM, 2014.

[20] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *Communications Surveys Tutorials, IEEE*, 17(2):998–1022, 2015.

[21] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 1025–1035. ACM, 2014.

[22] S. Greene. *Security Program and Policies: Principles and Practices*. Pearson Education, 2014.

[23] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade. Security toolbox for detecting novel and sophisticated android malware. In *Proceedings of the International Conference on Software Engineering - Volume 2*, pp. 733–736. IEEE Press, 2015.

[24] S. Kothari, A. Deepak, A. Tamrawi, B. Holland, and S. Krishnan. A "human-in-the-loop" approach for resolving complex software anomalies. In *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1971–1978, 2014.

[25] S. Liang, M. Might, and D. Van Horn. Anadroid: Malware analysis of android with user-supplied predicates. *Electronic Notes in Theoretical Computer Science*, 311:3–14, 2015.

[26] K. Munro. Deconstructing flame: the limitations of traditional defences. *Computer Fraud and Security*, 2012(10):8 – 11, 2012.

[27] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: towards automating risk assessment of mobile applications. In *USENIX conference on Security*, pp. 527–542. USENIX Association, 2013.

[28] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using covert. In *International Conference on Software Engineering-Volume 2*, pp. 725–728. IEEE Press, 2015.

[29] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Álvarez Marañón. Mama: manifest analysis for malware detection in android. *Cybernetics and Systems*, 44(6-7):469–488, 2013.

[30] S. Schmeelk, J. Yang, and A. Aho. Android malware static analysis techniques. In *Cyber and Information Security Research Conference*, pp. 5:1–5:8. ACM, 2015.