Apposcopy: Semantics-Based Detection of Android

Malware through Static Analysis

ABSTRACT

We present Apposcopy, a new semantics-based approach for

identifying a prevalent class of Android malware that steals

private user information. Apposcopy incorporates (i) a high-

level language for specifying signatures that describe seman-

tic characteristics of malware families and (ii) a static anal-

ysis for deciding if a given application matches a malware

signature. The signature matching algorithm of Apposcopy

uses a combination of static taint analysis and a new form of

program representation called Inter-Component Call Graph

to efficiently detect Android applications that have certain

control- and data-flow properties. We have evaluated Ap-

poscopy on a corpus of real-world Android applications and

show that it can effectively and reliably pinpoint malicious

applications that belong to certain malware families.

INTRODUCTION

As the most popular mobile operating system, the An-

droid platform is a growing target for mobile malware [4].

Today, many of the malicious applications that afflict An-

droid users exploit the private and monetized information stored in a user’s smartphone. According to a recent re-

port [3], nearly half of Android malware are multi-functional

Trojans that steal personal data stored on the user’s phone.

In response to the rapid dissemination of Android mal-

ware, there is a real need for tools that can automatically

detect malicious applications that steal private user informa-

tion. Two prevalent approaches for detecting such Android

malware are taint analyzers and signature-based detectors:

Taint analyses, such as [17, 21], are capable of exposing

applications that leak private user information. Unfortu-

nately, since many benign apps also need access to sensitive

data to perform their advertised functionality, not every app

that leaks user information can be classified as malware. For

instance, an email client application will necessarily “leak”

email addresses of the user’s contacts in order to perform its

functionality. Thus, taint analyses cannot automatically dis-

tinguish benign apps from malware, and a security auditor

must invest significant effort in order to determine whether

a given information flow constitutes malicious behavior.

Signature-based malware detectors, including commer-

cial virus scanners, classify a program as malware if it con-

tains a sequence of instructions that is matched by a regu-

lar expression. As shown in a recent study, malware detec-

tors that are based on syntactic low-level signatures can be

easily circumvented using simple program obfuscations [34].

Hence, these malware signatures must be frequently updated

as new variants of the same malware family emerge.

In this paper, we present Apposcopy, a new semantics-

based approach for detecting Android malware that steal

private user information. Drawing insights from the re-

spective advantages of pattern-based malware detectors and

taint analyzers, Apposcopy incorporates (i) a high-level spec-

ification language for describing semantic characteristics of

Android malware families, and (ii) a powerful static analysis

for deciding if a given application matches the signature of a

malware family. The semantic, high-level nature of the sig-

nature specification language allows analysts to specify key

characteristics of malware families without relying on the

occurrence of specific instruction or byte sequences, making

Apposcopy more resistant to low-level code transformations.

The signature specification language provided by Apposcopy

allows specifying two types of semantic properties —control-

flow and data-flow—of Android applications. An example of

a control-flow property is that the malware contains a broad-

cast receiver which launches a service upon the completion

of some system event. An example of a data flow property

is that the malware reads some private data of the device and sends it over the Internet.

To match the signatures specified in this language, Ap-

poscopy’s static analysis relies on two key ingredients. First,

we construct a new high-level representation of Android

applications called the inter-component callgraph (ICCG),

which is used to decide whether an Android application

matches the control flow properties specified in the signa-

ture. Second, Apposcopy incorporates a static taint anal-

ysis which is used for deciding whether a given application

matches a specified data-flow property.

We have evaluated Apposcopy on a corpus of real-world

Android applications and show that it can effectively and re-

liably pinpoint malicious applications –including obfuscated

ones– that belong to certain malware families. Despite the

theoretical undecidability of the semantic signature match-

ing problem, Apposcopy yields both few false positives and

few false negatives on current Android applications.

MOTIVATING EXAMPLE

In this section, we illustrate our approach using a sim-

plified version of the GoldDream malware family. As de-

scribed in Jiang’s security report [27], the key characteristic

of members of this family is that they register a receiver

for certain system events such as SMS messages or outgoing

phone calls. When these events trigger the execution of code

in the receiver, the malware then starts a background ser-

vice for sending private user information, such as the phone’s

unique IMEI number and subscriber id, to a remote server.

MALWARE SPEC LANGUAGE

This section describes Apposcopy’s malware signature lan-

guage, which is a Datalog program augmented with built-

in predicates. For each malware family, the user defines a

unique predicate that serves as the signature for this mal-

ware family. The user may also define additional helper

predicates used in the signature. In what follows, we first

give some background on Datalog, and then describe the the

syntax and semantics of Apposcopy’s built-in predicates.

3.2

Apposcopy’s Built-in Predicates

We now describe the syntax and semantics of the four

classes of built-in predicates provided by Apposcopy.

3.2.1

Component Type Predicates

Component type predicates in Apposcopy represent the

different kinds of components provided by the Android framework. An Android application consists of four kinds of com-

ponents, Activity, Service, BroadcastReceiver, and Content-

Provider. Activity components form the basis of the user

interface, and each window of the application is typically

controlled by an activity. Service components run in the

background and remain active even if windows are switched.

BroadcastReceiver components react asynchronously to mes-

sages from other applications. Finally, ContentProvider com-

ponents store data relevant to the application, usually in a

database, and allow sharing data across applications.

Corresponding to each of these Android components, Ap-

poscopy provides pre-defined predicates called service(C),

activity(C), receiver(C), and contentprovider(C) which

represent the type of component C. For example, activity(C)

is true if C is an Activity. Each of these four predicates cor-

respond to relations of type (comp : C) where domain C is

the set of all components in the application.

3.2.2

Predicate icc

A key idea behind inter-component communication (ICC)

in Android is that of Intents, which are messages passed

between components. Intents are used to start Activities;

start, stop, and bind Services; and broadcast information to

Broadcast Receivers. Table 1 shows a list of Android APIs

that are used in ICC. We refer to those methods, all of which

take Intent objects as arguments, as ICC method s. We use

the term ICC site to represent a statement that invokes one

of the ICC methods listed in Table 1. When component a

initiates ICC with component b, the Android system will

eventually call life-cycle methods associated with compo-

nent b. The life-cycle methods are shown in Table 2.

While an Intent object passed to ICC methods can contain

many different types of information, Apposcopy’s signature

language takes into account only two kinds: action and data.

Action is a string that represents the type of action that the

receiving component needs to perform, and data specifies the

type of data that the component needs to operate on.

4.

STATIC ANALYSES

This section describes Apposcopy’s static analyses for de-

ciding whether an application matches a malware signature.

The main idea is to compute an over-approximation of the

icc, calls, and flow relations.

4.1

Pointer Analysis & Callgraph Contruction

In order to build the inter-component callgraph and track

information flow, Apposcopy starts by performing a pointer

analysis, which computes the set of abstract heap objects

that each variable may point to. In the remainder of the

paper, we use the notation v ,→ o to denote that variable

v may point to an abstract heap object represented by o in

some execution of the program.

Since the precision of the underlying pointer analysis is

critical for detecting malware with few false alarms, we use

a field- and context-sensitive Andersen-style pointer analy-

sis [7]. For context-sensitivity, we use a hybrid approach that

combines call-site sensitivity [30] and object-sensitivity [31].

In particular, our approach is similar to the technique de-

scribed in [28] and uses call-site sensitivity for static method

calls and object-sensitivity for virtual method calls.

Another key ingredient of our malware detection algo-

rithm is callgraph construction, which is used for resolving

the targets of virtual method calls. Since callgraph precision

has significant impact on the precision of the ICCG, Ap-

poscopy computes the callgraph on-the-fly, simultaneously

refining the targets of virtual method calls and points-to sets

until a fixed point is reached.

IMPLEMENTATION & EVALUATION

Our implementation consists of about 30,000 lines of Java

and uses several publicly-available software such as Soot [35]

and bddbddb [36]. Soot is used to convert Android’s .apk

files to Jimple, a higher-level intermediate representation.

A pre-processing step processes Jimple instructions to ex-

tract various types of program facts, and our static analyses

are specified as Datalog programs. The bddbddb system

takes as input the Datalog specification of a static analysis

and extracted program facts and outputs the results of the

analysis. Apposcopy’s static analyses use manually-written

models of the Android framework; currently, we have models

for about 1,100 classes that are relevant for our analyses.

To evaluate the effectiveness and accuracy of Apposcopy,

we performed four sets of experiments, including evaluation

on (i) known malware, (ii) Google Play apps, (iii) obfuscated

malware. In addition, (iv) we also compare Apposcopy with

another research tool called Kirin for Android malware de-

tection. The remainder of this section describes the details

and results of our evaluations.

Table 7 presents the results of this experiment. The first

column indicates the malware family, and the second col-

umn shows the number of analyzed instances of that mal-

ware family. The next two columns show the number of false

negatives (FN) and false positives (FP) respectively. In this

context, a false negative arises if an application A belongs

to a certain malware family F but Apposcopy cannot detect

that A is an instance of F . Conversely, a false positive arises

if an application A does not belong to malware family F but

Apposcopy erroneously reports that it does. The final col-

umn of Table 7 reports Apposcopy’s overall accuracy, which

is calculated as the number of correctly classified instances

divided by the total number of samples.

As shown in the last row of Table 7, the overall accuracy

of Apposcopy over all malware instances that we analyzed

is 90.0%. That is, it can correctly classify approximately 9

out of 10 malware instances accurately.

However, looking at the results more closely, we see that

Apposcopy performs quite poorly on the BaseBridge mal-

ware family. Specifically, among the 121 samples that we

analyzed, Apposcopy only classifies 46 of these applications

as instances of BaseBridge. Upon further inspection of this

family, we found that many of the BaseBridge instances dy-

namically load the code that performs malicious function-

ality. Such behavior can inherently not be detected using

static analysis and causes Apposcopy to yield many false

negatives. Observe that, if we exclude BaseBridge from our

samples, the overall accuracy of Apposcopy rises to 96.9%.

For the other malware families for which Apposcopy yields

false negatives, there are several contributing factors. First,

since we have written the specifications by inspecting only a

small number of samples, our signatures may not adequately

capture the essential characteristics of all instances of that

family. Second, the malware family may have some key fea-

ture that is not expressible in our malware specification lan-

guage. For example, if a given malware performs malicious

functionality without leaking sensitive data, Apposcopy will

be unable to detect it. A third contributing factor for false

negatives is due to missing models. Specifically, while our

static analysis is sound, we do not analyze the underlying

code of the Android system, but instead rely on method stubs

that capture the relevant behavior of the Android SDK. If

these applications call SDK methods for which we have not

provided stubs, Apposcopy may yield false negatives.

Based on the data from Table 7, we observe that Ap-

poscopy reports very few false positives. Among the 1027

malware samples, Apposcopy reports two instances of the

Geinimi family as instances of both Geinimi as well as the

DroidKungFu family. This corresponds to an overall false

positive ratio of less than 0.2%, indicating that Apposcopy’s

static analysis is precise enough to accurately answer control-

and data-flow queries about Android applications.

Evaluation on Google Play Apps

In a second experiment, we evaluate Apposcopy on thou-

sands of apps from Google Play. Since these applications are

available through the official Android market rather than

less reliable third-party app markets, we would expect a

large majority of these applications to be benign. Hence,

by running Apposcopy on Google Play apps, we can assess

whether our high-level signatures adequately differentiate

benign applications from real malware.

In our experiment, among the 11,215 apps analyzed by

Apposcopy, only 16 of them were reported as malware. Specif-

ically, Apposcopy reported two applications to be instances

of DroidDreamLight, one to be an instance of DroidDream

and another one to be an instance of Pjapps. The remaining

12 applications were categorized as DroidKungFu. To de-

cide whether these 16 apps are indeed malware, we uploaded

them to VirusTotal [5] for analyzing suspicious applications.

VirusTotal is a service that runs multiple anti-virus tools on

the uploaded application and shows their aggregate results.

Based on the results provided by VirusTotal, the majority

of anti-virus tools agree with Apposcopy’s classification for

13 of the 16 reported malware. For the remaining three

applications, the majority of the tools classify them as ma-

licious adware while Apposcopy classifies them as instances

of DroidKungFu. This experiment confirms our claim that

Apposcopy does not generate a lot of false alarms and that

our malware signatures can distinguish benign applications

from real malware.

Similar to the experiments from Section 5.1, Apposcopy

takes an average of 346 seconds to analyze a Google Play

application with 26,786 lines of Dalvik bytecode on average.

Evaluation on Obfuscated Apps

To substantiate our claim that Apposcopy is resilient to

code transformations, we compare the detection rate of Ap-

poscopy with other anti-virus tools on obfuscated versions of

known malware. For this experiment, we obfuscated exist-

ing malware using the ProGuard tool [2], which is commonly

used by malware writers to evade detection. In addition,

since the databases of some of the anti-virus tools include

signatures of malware samples obfuscated by ProGuard, we

also applied three additional obfuscations that are not per-

formed by ProGuard: First, our obfuscator changes the

names of components, classes, methods, and fields. Second,

all invocations to methods of android.\* classes are redi-

rected through proxy methods. Third, our obfuscator also

performs string encryption, including encryption of compo-

nent names as well as action and data type values of intents.

Table 8 shows the results of our third experiment on ob-

fuscated malware. Each row in this table corresponds to an

application that is an instance of a known malware family

(and whose unobfuscated version can be identified as mal-

ware by all tools considered in our evaluation). Each column

in the table corresponds to a leading anti-virus tool, namely,

AVG, Symantec, ESET, Dr. Web, Kaspersky, Trend Micro,

and McAfee. A check (3) indicates that the tool is able to

detect the obfuscated version of the program as malware,

and a cross (7) means that the tool is unable to classify

the obfuscated version as malicious. As Table 8 shows, Ap-

poscopy is resistant to these obfuscations for all malware

that we considered. In contrast, none of the other tools can

successfully classify all of the obfuscated apps as malware.

5.4

Comparison with Kirin

In addition to comparing Apposcopy with commercial anti-

virus tools, we also compared Apposcopy against Kirin [19],

which is the only publicly available research tool for An-

droid malware detection. As explained in Section 6, Kirin is

a signature-based malware detector that classifies an app as

malware if it uses a dangerous combination of permissions

specified by the malware signature. On the set of malicious

apps considered in Section 5.1, Kirin reports only 532 apps

out of 1,027 malicious apps to be malware. This corresponds

to a false negative rate of 48% , which is quite high compared

to the 10% false negative rate of Apposcopy. On the other

hand, for the set of applications taken from Google Play and

considered in Section 5.2, Kirin reports 8% of these apps to

be malware, while Apposcopy classifies only 0.14% of these

apps as malicious. We manually inspected 20 out of the

886 apps classified as malware by the Kirin tool and also

compared with the results of VirusTotal. Our evaluation re-

vealed that the overwhelming majority of the apps classified

as malware by Kirin are false positives. Hence, our experi-

ments demonstrate that Apposcopy outperforms Kirin both

in terms of false positives as well as false negatives.

6.

RELATED WORK

Taint analysis. Both dynamic and static taint anal-

yses have been proposed for tracking information-flow in

mobile applications. For example, TaintDroid [17] and Vet-

Droid [37] are dynamic taint analyses that track information

flow by instrumenting the Dalvik VM, and examples of static

taint analyses include [20, 21, 23, 16]. While Apposcopy

employs static taint analysis as one of its components, we

observe that not every application that leaks sensitive data

is malicious — in fact, many benign apps transmit sensi-

tive data for performing their required functionality. Thus,

taint analyses on their own are not sufficient for automat-

ically differentiating malicious apps from benign apps, and

we propose to combine taint analysis with high-level mal-

ware signatures to effectively identify malicious code.

Signature-based malware detection. Many techniques

for identifying existing malware are signature-based, mean-

ing that they look for patterns identifying a certain malware

family. In its simplest form, these patterns are sequences

of bytes or instructions [25]. Since such syntactic patterns

can be defeated by semantics-preserving transformations,

previous work has considered semantics-aware malware de-

tection [13]. Similar to [13], Apposcopy detects malware

based on their semantic rather than syntactic characteris-

tics. However, our signatures are much higher-level com-

pared to the templatized instruction sequences used in [13]

and allow directly specifying control- and data-flow proper-

ties of Android applications. Furthermore, the underlying

signature matching techniques are also very different.

A popular signature-based malware detection technique

for Android is the Kirin tool [19]. The malware signatures

in Kirin specify dangerous combinations of Android permis-

sions, and Kirin decides if an application matches a signa-

ture by analyzing its manifest file. As demonstrated in our

experimental evaluation, Kirin yields many more false posi-

tives and false negatives compared to Apposcopy.

Another related approach is the behavioral detection tech-

nique described in [9]. In that approach, one specifies com-

mon malware behavior using temporal logic formulas. How-

ever, a key difference between Apposcopy and behavioral

detection is that our techniques are purely static, while [9]

requires monitoring behavior of the application at run-time.

The DroidRanger tool [40] uses permission-based behav-

ioral footprint to detect instances of known malware fam-

ilies. Behavioral footprints include characteristic malware

features, such as listening to certain system events, calling

suspicious APIs, and containing hard-coded strings. While

these behavioral footprints can be viewed as high-level mal-

ware signatures, they differ from those of Apposcopy in sev-

eral ways: Behavioral footprints neither refer to information

flow properties of an application nor do they express control

flow dependencies between different components. Further-

more, behavioral footprints include hard-coded string val-

ues, which are easy to obfuscate by malware writers.

Zero-day malware detection. A known limitation of

signature-based approaches, including Apposcopy, is that

they can only detect instances of known malware families.

In contrast, zero-day malware detectors try to uncover previ-

ously unknown malware families. For example, RiskRanker [24]

performs several risk analyses to rank Android applications

as high-, medium-, or low-risk. These risk analyses include

techniques to identify suspicious code that exploits platform-

level vulnerabilities or sends private data without being trig-

gered by user events. In addition to identifying instances

of known malware, DroidRanger [40] also tries to uncover

zero-day malware by performing heuristic-based filtering to

identify certain “inherently suspicious” behaviors.

Many recent malware detectors, such as [8, 22, 10, 33, 6],

use machine learning to detect zero-day malware. For ex-

ample, Drebin [8] performs light-weight static analysis to ex-

tract features, such as permissions and API calls, and trains an SVM to find a hyperplane separating benign apps from

malware. The DroidAPIMiner tool [6] also considers API

features and uses machine learning to automatically classify

an application as malicious or benign. While learning-based

approaches are powerful for detecting unknown malware,

their precision relies on representative training sets.

We believe all of these zero-day malware detection tech-

niques are complementary to Apposcopy: While Apposcopy

can identify instances of known malware families with few

false alarms, zero-day malware detectors can help uncover

new malware families, albeit at the cost of more false posi-

tives or more involvement from a security auditor.

Static analysis for malware detection. Static anal-

ysis and model checking have been used to detect security

vulnerabilities for a long time. In the context of mobile

malware, the SAAF tool [26] uses program slicing to iden-

tify suspicious method arguments, such as certain URLs or

phone numbers. The work described in [18] performs vari-

ous static analyses, including taint analysis, to better under-

stand smartphone application security in over 1,000 popular

Android apps. One of the interesting conclusions from this

study is that, while many apps misuse privacy-sensitive in-

formation, few of these apps can be classified as malware.

The Pegasus system [11] focuses on malware that can be

identified by the order in which certain permissions and APIs

are used. The user writes specifications of expected app be-

havior using temporal logic formulas, and Pegasus model

checks these specifications against an abstraction called the

permission event graph (PEG). Pegasus differs from our ap-

proach in that (i) it targets a different class of malware, and

(ii) PEG abstracts the relationship between the Android sys-

tem and the application, while ICCG abstracts the relation-

ship between different components in the application.

Some recent papers address the detection of re-packaged

apps which often inject adware or malicious features into

legitimate apps [39, 38, 15, 14]. While some repackaged

apps may contain malware, these techniques mainly focus

on clone rather than malware detection.

Analysis of ICC. ComDroid [12] analyzes ICC of An-

droid apps to expose security vulnerabilities, such as intent

spoofing or unauthorized intent receipt. CHEX [29] per-

forms static analysis to identify app entry points and uses

this information to detect component hijacking vulnerabil-

ities. In contrast to Apposcopy, ComDroid and CHEX are

meant to be used by developers to identify security vulner-

abilities in their own applications.

LIMITATIONS

Like any signature-based solution, Apposcopy is not invin-

cible; it is very hard to design any signature-based scheme

that cannot be defeated by a suitably designed automatic

obfuscator. In particular, similar to any static analysis based

system, Apposcopy may be defeated by obfuscation tech-

niques such as dynamic code loading and use of reflection

in combination with obfuscation of method or class names.

However, such attempts to escape detection are likely to be

deemed suspicious and may invite further scrutiny.

Second, since Apposcopy performs deep static analysis

to uncover semantic properties of an app, it may be un-

fit for scenarios that require instant detection of malware.

However, smartphone apps are generally distributed through

centralized app stores, which enables deployment of Ap-

poscopy to scan apps as they are submitted to the app store.

8.

CONCLUSION AND FUTURE WORK

We presented Apposcopy, a static analysis approach for

detecting malware in the mobile apps ecosystem. Malware

that belong to one family share a common set of charac-

teristic behaviors, which an auditor can encode through Ap-

poscopy’s Datalog-based malware specification language. Ap-

poscopy performs deep static analysis to extract data-flow

and control-flow properties of Android applications and uses

these results to identify whether a given application belongs

to a known malware family. Our experiments indicate that

Apposcopy can detect malware with high accuracy and that

its signatures are resilient to various program obfuscations.

There are several opportunities for future work. We will

develop techniques to improve the efficiency and precision

of Apposcopy’s static analyses. We also plan to develop

techniques to automatically de-obfuscate apps to enhance

Apposcopy’s resilience to some types of obfuscations (see

Section 7). Finally, we plan to develop techniques to auto-

matically learn malware signatures from a set of apps labeled

with their corresponding malware family (or as benign).