Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding

Mu Zhang
Department of EECS
Syracuse University
Syracuse, USA
muzhang@syr.edu

Heng Yin
Department of EECS
Syracuse University
Syracuse, USA
heyin@syr.edu

ABSTRACT

As Android has become the most prevalent operating system in mobile devices, privacy concerns in the Android platform are increasing. A mechanism for efficient runtime enforcement of information-flow security policies in Android apps is desirable to confine privacy leakage. The prior works towards this problem require firmware modification (i.e., modding) and incur considerable runtime overhead. Besides, no effective mechanism is in place to distinguish malicious privacy leakage from those of legitimate uses. In this paper, we take a bytecode rewriting approach. Given an unknown Android app, we selectively insert instrumentation code into the app to keep track of private information and detect leakage at runtime. To distinguish legitimate and malicious leaks, we model the user's decisions with a context-aware policy enforcement mechanism. We have implemented a prototype called `Capper` and evaluated its efficacy on confining privacy-breaching apps. Our evaluation on 4723 real-world Android applications demonstrates that `Capper` can effectively track and mitigate privacy leaks. Moreover, after going through a series of optimizations, the instrumentation code only represents a small portion (4.48% on average) of the entire program. The runtime overhead introduced by `Capper` is also minimal, merely 1.5% for intensive data propagation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms

Security

Keywords

Android; Privacy leakage; Context-aware policy; Bytecode rewriting

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ASIA CCS'14, June 03 - 06 2014, Kyoto, Japan.
Copyright 2014 ACM 978-1-4503-2800-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2590296.2590312.

1. INTRODUCTION

Android continues to dominate in the battle to be the top smartphone system in the world, and ranked as the top smartphone platform with 52 percent market share (71.1 million subscribers) in Q1 2013. The success of Android is also reflected from the popularity of its application markets. Tens of thousands of Android apps become available in Google Play while popular apps (e.g., Adobe Flash Player 11) have been downloaded and installed over 100 million times.

Meanwhile, privacy concerns in the Android platform are increasing. Previous studies [12, 13, 19, 31, 36, 38] have exposed that both benign and malicious apps are stealthily leaking users' private information to remote servers. Efforts have also been made to detect and analyze privacy leakage either statically or dynamically [12, 13, 17, 21, 24, 25, 34]. Nevertheless, a good solution to defeat privacy leakage at runtime is still lacking. We argue that a practical solution needs to achieve the following goals:

- **Information-flow based security.** Privacy leakage is fundamentally an information flow security problem. A desirable solution to defeat privacy leakage would detect sensitive information flow and block it right at the sinks. However, most of prior efforts to this problem are "end-point" solutions. Some earlier solutions extended Android's install-time constraints and enriched Android permissions [14,29]. Some aimed at enforcing permissions in a finer-grained manner and in a more flexible way [6, 8, 27, 39]. Some attempted to improve isolation on various levels and each isolated component could be assigned with a different set of permissions [5, 22, 30]. In addition, efforts were made to introduce supplementary user consent acquisition mechanism, so that access to sensitive resource also requires user approval [23, 32]. All these "end-point" solutions only mediate the access to private information, without directly tackling the privacy leakage problem.
- **Low runtime overhead.** An information-flow based solution must have very low runtime overhead to be adopted on end users' devices. To directly address privacy leakage problem, Hornyack et al. proposed AppFence to enforce information flow policies at runtime [19]. With support of TaintDroid [12], AppFence keeps track of the propagation of private information. Once privacy leakage is detected, AppFence either blocks the leakage at the sink or shuffle the information from the source. Though effective in terms of blocking privacy leakage, its efficiency is not favorable. Due to the taint tracking on every single Dalvik bytecode instruction, AppFence incurs significant performance overhead.
- **No firmware modding.** For a practical solution to be widely adopted, it is also crucial to avoid firmware modding. Unfor-

tunately, the existing information-flow based solutions such as AppFence require modifications on the stock software stack, making it difficult to be deployed on millions of mobile devices.

- **Context-aware policy enforcement.** Many apps need to access user's privacy for legitimate functionalities and these information flows should not be stopped. Therefore, to defeat privacy leakage without compromising legitimate functionality, a good solution needs to be aware of the context where a sensitive information flow is observed and make appropriate security decisions. To the best of our knowledge, we are not aware that such a policy mechanism exists.

In this paper, we aim to achieve all these design goals by taking a bytecode rewriting approach. Given an unknown Android app, we selectively rewrite the program by inserting bytecode instructions for tracking sensitive information flows *only* in certain fractions of the program (which are called taint slices) that are potentially involved in information leakage. When an information leakage is actually observed at a sink node (e.g., an HTTP Post operation), this behavior along with the program context is sent to the policy management service installed on the device and the user will be notified to make an appropriate decision. For example, the rewritten app may detect the location information being sent out to a Google server while the user is navigating with Google Map, and notify the user. Since the user is actively interacting with the device and understands the context very well, he or she can make a proper decision. In this case, the user will allow this behavior. To ensure good user experiences, the number of such prompts must be minimized. To do so, our policy service needs to accurately model the context for the user's decisions. As a result, when an information leakage happens in the same context, the same decision can be made without raising a prompt. After exploring the design space of the context modeling and making a balance between sensitivity, performance overhead, and robustness, we choose to model the context using *parameterized source and sink pairs*.

Consequently, our approach fulfills all the requirements: 1) actual privacy leaks are captured accurately at runtime, with the support of inserted taint tracking code; 2) the performance overhead of our approach is minimal, due to the static dataflow analysis in advance and numerous optimizations that are applied to the instrumentation code; 3) the deployment of our approach is simple, as we only rewrite the original app to enforce certain information flow policies and no firmware modification is needed; 4) policy enforcement is context-aware, because the user's decisions are associated with abstract program contexts.

We implement a prototype, `Capper`[1], in 16 thousand lines of Java code, based on the Java bytecode optimization framework Soot [3]. We leverage Soot's capability to perform static dataflow analysis and bytecode instrumentation. We evaluate our tool on 4723 real-world privacy-breaching Android apps. Our experiments show that rewritten programs run correctly after instrumentation, while privacy leakage is effectively eliminated.

Contributions.

In summary, this paper makes the following contributions:

- We propose a bytecode rewriting approach to the problem of privacy leaks in Android applications. It requires no firmware changes and incurs minimal performance impact.

[1]`Capper` is short for Context-Aware Privacy Policy Enforcement with Re-writing.

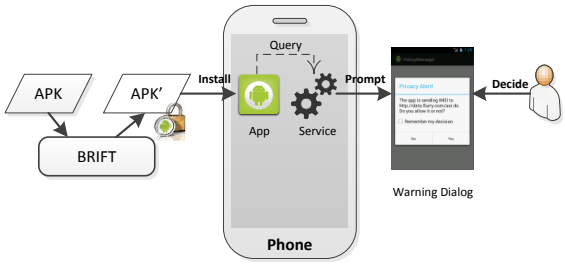


Figure 1: Architecture of `Capper`

- We design a novel mechanism to model program contexts with user knowledge. This helps differentiate between benign operations on sensitive data and actual privacy leakage.

- We evaluate `Capper` on 4723 real Android apps. The experiments show that our approach is both effective and efficient. We demonstrate that rewritten programs run correctly after instrumentation, while privacy leakage is effectively eliminated. We show that only a small portion (2.48% on average) of the program needs to be instrumented and after a series of optimizations the final program size increases only 4.48% on average. We also show that our runtime overhead is minimal, merely 1.5% for intensive data propagation.

2. APPROACH OVERVIEW

In this section, we present an overview of our technique. Further, we define the problem scope and provide a running example.

2.1 Key Techniques

Figure 1 depicts an overview of our techniques. When a user is about to install an app onto his Android device, this app will go through our bytecode rewriting engine (BRIFT) and be rewritten into a new app, in which sensitive information flows are monitored by the inserted bytecode instructions. Therefore, when this new app is actually running on the device and is observed to send out sensitive information, this behavior (along with the program context) will be reported to the policy management service for decision.

If this behavior under this program context is observed for the first time, the policy management service will prompt the user to make a proper decision: either allow or deny such a behavior. The user's decision will be recorded along with the program context, so the policy management service will make the recorded decision for the same behaviors under the same context.

Therefore, our solution to defeat privacy leakage consists of the following two enabling techniques.

(1) Bytecode Rewriting for Information Flow Control.

Given a bytecode program, the goal of our bytecode rewriting is to insert a minimum amount of bytecode instructions into the bytecode program to trace the propagation of certain sensitive information flows (or taint). To achieve this goal, we first conduct static dataflow analysis to compute a number of program slices that are involved in the taint propagation. Then we insert bytecode instructions along the program slices to keep track of taint propagation at runtime. Further, we perform a series of optimizations to reduce the amount of inserted instructions. More details are presented in Section 3.

(2) Context-aware Policy Enforcement.

The user allows or denies a certain information flow in a specific context. The key for a context-aware policy enforcement is to prop-

erly model the context. The context modeling must be sensitive enough to distinguish different program contexts, but not too sensitive. Otherwise, a slight difference in the program execution may be treated as a new context and may cause unnecessarily annoying prompts to the user. Further, the context modeling should also be robust enough to counter mimicry attacks. An attacker may be able to "mimic" a legitimate program context to bypass the context-aware policy enforcement. We present more details about the context modeling and policy enforcement in Section 4.

2.2 Problem Scope

Our proposed solution is designed to enforce user preferred privacy policy on innocent Android apps. Our solution can be used to block privacy leakage in most of current Android malware apps, but a dedicated malware author can still find ways to circumvent our confinement. For instance, malware can exfiltrate private information through side channels (such as implicit flows and timing channels). Most of previous solutions (including dynamic monitoring and static analysis approaches) share the same limitation. Moreover, the proposed technique (again shared by most of previous solutions) cannot handle native components and Java reflective calls in a general way. In rare occasions, our system will raise a warning to the user that the rewritten app may still be unsafe if we observe a JNI call or a reflective call appears on the information propagation path. According to our experimental evaluation in Section 5, only a small fraction (5%) of apps belong to this category. In this case, a security-conscious user may decide not to use this unsafe app or resort to other solutions.

2.3 Running Example

To elaborate the whole process, we use a synthetic running example to explain our approach. Figure 2 presents our synthetic running example in Java source code. More concretely, the example class extends an Android `Activity` and overrides several callbacks including `onStart()`, `onResume()` and `onDestroy()`. When the `Activity` starts, `onStart()` method will get the device identifier by calling `getIMEI()`, which returns the real device ID if it succeeds or empty string otherwise. On receiving the return value from `getIMEI()`, the program stores it to a static field `deviceId`. Further, both `onResume()` and `onDestroy()` read the device ID from this static field but use it for different purposes. While `onResume()` shows the device ID on screen via a `Toast` notification, `onDestroy()` encrypts it and sends it to a remote server through a raw socket. In Section 3 and 4 , we show how to perform bytecode rewriting and mitigate privacy leakage with this example.

3. BYTECODE REWRITING

We leverage our `BRIFT`[2] engine to insert the information flow control logic into a given Android app. BRIFT takes the following steps to rewrite a bytecode program: 1) it first translates the Dalvik executable (i.e., DEX file) to an intermediate representation (IR) to facilitate analysis and instrumentation; 2) it performs application-wide static dataflow analysis on IR to identify program slices for information leakage; 3) to keep track of data propagation and prevent the actual information leakage, it instruments the IR by inserting new IR statements along the program slices; 4) to further improve performance, it applies a series of optimization methods to remove redundant and unnecessary IR statements; and 5) in the end, it generates a new Dalvik executable from the IR and uses it

[2]It is short for Bytecode Rewriting for Information Flow Tracking

```
1  public class Leakage extends Activity{
2    private byte key = DEFAULT_KEY;
3    private String addr = DEFAULT_ADDR;
4    private static String deviceId;
5
6    public String getIMEI(){
7      TelephonyManager manager = (TelephonyManager)
         getSystemService("phone");
8      String imei = manager.getDeviceId();
9      if(imei==null){
10       imei = "";
11     }else{
12       imei = manager.getDeviceId();
13     }
14     return imei;
15   }
16
17   public byte crypt(byte plain){
18     return (byte)(plain ^ key);
19   }
20
21   public void post(String addr, byte[] bytes){
22     OutputStream output = conn.getOutputStream();
23     output.write(bytes, 0, bytes.length);
24     ...
25   }
26
27   public void toastIMEI(String imei){
28     Context app = getApplicationContext();
29     String text = "Your IMEI is " + imei;
30     int duration = Toast.LENGTH_SHORT;
31     Toast toast = Toast.makeText(app, text, duration);
32     toast.show();
33   }
34
35   public void onStart(){
36     Leakage.deviceId = getIMEI();
37   }
38
39   public void onResume(){
40     toastIMEI(Leakage.deviceId);
41   }
42
43   public void onDestroy(){
44     String imei = Leakage.deviceId;
45     byte[] bytes = imei.getBytes();
46     for(int i=0; i<bytes.length; i++){
47       bytes[i] = crypt(bytes[i]);
48     }
49     post(addr, bytes);
50   }
51 }
```

Figure 2: Java Code for the Running Sample

to repackage the new .apk file. We give a brief explanation for each step as below. For more details, please refer to our prior work [35].

3.1 Converting DEX into IR

An Android app generally consists of two parts, resources and a Dalvik executable file. Our bytecode rewriting is performed on the Dalvik executable file, so the resources remain the same, and will be repackaged into the new app in the last step. To convert DEX into IR, we first convert the DEX into a Java bytecode program using dex2jar [1][3] and then translate the Java bytecode into an intermediate representation with Soot [3]. More specifically, Soot produces Jimple IR to facilitate subsequent dataflow analysis and instrumentation.

3.2 Application-wide Dataflow Analysis

On Jimple IR, we perform flow-sensitive context-sensitive interprocedural analysis to track the propagation of certain *tainted* sensitive information and detect if the taint propagates into the sinks (e.g., Internet and file system). The result of this forward dataflow

[3]It is also possible to use alternative tools, such as Dare [28], ded [13], etc., for this conversion.

analysis is a taint propagation graph. If one or more sinks appear in this graph, it indicates a potential information leakage. Then for each sink in the graph, we perform backward dependency analysis to compute a "slice". Taint propagation outside this slice is irrelevant to the leakage and thus can be ignored. We call this slice *taint slice*, because it represents a program slice that contributes to a leakage from an information source to a sink. We present the analysis result for the running example in Appendix A.

3.3 Static Instrumentation

Static dataflow analysis is usually conservative and may lead to false positives. Therefore, we insert instrumentation code in these taint propagation slices. The inserted code serves as runtime checks to actually keep track of the taint propagation while the Android application is running. The sinks are also instrumented to examine the taint and confine the information leakage.

We create shadows, for each data entity defined or used within the slices, to track its runtime taint status. The data entities outside the slices do not need to be shadowed, because they are irrelevant to the taint propagation. We create shadows for different types of data entities individually. Static or instance fields are shadowed by adding boolean fields into the same class definition. A local variable is shadowed with a boolean local variable within the same method scope, so that compiler optimizations can be applied smoothly on related instrument code.

Shadowing method parameters and return value requires special considerations. Firstly, the method prototype needs modification. Extra "shadow parameters" are added to parameter list to pass the shadows of actual parameters and return value from caller to callee. Secondly, parameters are passed-by-value in Java, and therefore primitive-typed local shadow variables (boolean) need to be wrapped as objects before passing to a callee. Otherwise, the change of shadows in the callee cannot be reflected in the caller. To this end, we define a new class `BoolWrapper`. This class only contains one boolean instance field, which holds the taint status, and a default constructor, as shown below. Notice that the Java `Boolean` class cannot serve the same purpose because it is treated the same as primitive boolean type once passed as parameter.

```
public class BoolWrapper extends java.lang.Object{
  public boolean b;
  public void <init>(){
    BoolWrapper r0;
    r0 := @this: BoolWrapper;
    specialinvoke r0.<java.lang.Object: void <init>()>();
    return; }
}
```

With the created shadows, we instrument sources, data propagation code and sinks. At the source of information flow, we introduce taint by setting the corresponding shadow variable to "true". For data propagation code, we instrument an individual instruction depending on its type. 1) If the instruction is an assignment statement or unary operation, we insert a definition statement on correlative shadow variables. 2) If it is a binary operation, a binary OR statement is inserted to operate on shadow variables. If one of the operators is a constant, we replace its shadow with a constant "false". 3) Or, if it is a function call, we need to add code to bind shadows of actual parameters and return value to shadow parameters. 4) Further, if the instruction is a API call, we model and instrument the API with its taint propagation logic.

We generally put APIs into the following categories and handle each category with a different model. "get" APIs have straightforward taint propagation logic, always propagating taint from parameters to their return values. Therefore, we generate a default rule, which propagates taint from any of the input parameters to the return value. Similarly, simple "set" APIs are modeled as they propagate taint from one parameter to another parameter or "this" reference. APIs like `Vector.add(Object)` inserts new elements into an aggregate construct and thus can be modeled as a binary operation, such that the object is tainted if it is already tainted or the newly added element is tainted. APIs like `android.content.ContentValues.put(String key, Byte value)` that operate on (key, value) pairs can have more precise handling. In this case, an element is stored and accessed according to a "key". To track taint more precisely, we keep a taint status for each key, so the taint for each (key, value) pair is updated individually.

Then, at the sink, we insert code before the sink API to check the taint status of the sensitive parameter. If it turns out the critical parameter is tainted, the inserted code will query a separate policy service app for decision and a warning dialog is then displayed to the user.

We also devise taint cleaning mechanism. That is, if a variable is redefined to be an untainted variable or a constant outside taint propagation slices, we thus insert a statement after that definition to set its shadow variable to 0 (false).

3.4 Optimization

We further optimize the added instrumentation code. This is to remove the redundant bytecode instructions that are inserted from the previous step. As Soot's built-in optimizations do not apply well on this instrumentation code, we devise three custom optimization techniques to safely manipulate the instrumentation code and remove redundant ones. Thereafter, the optimized code is now amenable to the built-in optimizations. Consequently, after going through both custom and built-in optimizations, the added instrumentation code can be reduced to a minimum, ensuring the best performance of the rewritten bytecode program.

To be more specific, we have devised four steps of optimizations, described as follows.

- **O1**: In instrumentation, we add a shadow parameter for every single actual parameter and return value. However, some of them are redundant because they don't contribute to taint propagations. Therefore, we can remove the inserted code which uses solely these unnecessary shadow parameters.
- **O2**: Next, we remove redundant shadow parameters from parameter list and adjust method prototype. Consequently, instrumentation code, that is used to initialize or update the taint status of these shadow parameters, can also be eliminated.
- **O3**: Further, if inserted taint tracking code is independent from the control-flow logic of a method, we can lift the tainting code from the method to its callers. Thus, the taint propagation logic is inlined.
- **O4**: After custom optimizations, instrumentation code is amenable to Soot's built-in optimization, such as constant propagation, dead code elimination, etc.

3.5 Code Generation

At last, we convert the modified Jimple IR into a new package (.apk file). More concretely, we translate the Jimple IR to Java package using Soot, and then re-target Java bytecode to Dalvik bytecode with Android SDK. In the end, we repackage the new DEX file with old resources and create the new .apk file.

3.6 Rewritten Running Example

Figure 3 presents the rewritten program after the instrumentation and optimizations. For the sake of readability, we present the

rewritten code in Java as a "diff" to the original program, even though the rewritten program is actually generated on the Jimple IR level. The statements with numeric line numbers are from the original program, whereas those with special line number "**I**" are inserted statements. The underlines mark either newly introduced code or modified parts from old statements. We hereby use this code to exemplify our design and implementation choices.

We can see that a boolean variable `deviceId_s0_t` is created to shadow the static field `deviceId`. "`s0`" denotes the source label which distinguishes different taint sources, while "`_t`" is the suffix for shadow in our implementation. Then in `onStart()`, the shadow variable `deviceId_s0_t` is set to be the "`status`" field in an object `ret_s0_wrapper`.

`ret_s0_wrapper` is an object of "`BoolWrapper`" structure which is introduced for the purpose of wrapping and passing primitive-typed shadow variables. In `Start()`, such an object is created and passed to `getIMEI(BoolWrapper)`, and further will be updated in the latter with the shadow of return value.

The shadow return value of `getIMEI(BoolWrapper)` is obtained from the local shadow variable `imei_s0_t`, which is initialized to be "true" if privacy access branch is taken (Ln.12), or cleaned to be "false" if `imei` is set as a constant (Ln.10). Without a cleaning mechanism, the value of corresponding shadow variable will remain "true" even if the taint status has already changed.

In the method `onDestroy()`, `deviceId` is used and sensitive data can thus be propagated to "`bytes`" array. In the meantime, `bytes_s0_t`, the shadow of "`bytes`" array is also assigned with `deviceId_s0_t`. We here choose to keep one single shadow for an entire array for efficiency. Further, the array is passed to the `crypt()` method in a loop and we update its taint status in every iteration. It is notable that taint propagation was devised in the callee (i.e., `crypt()`) at the beginning. However, due to optimizations, the taint logic in `crypt()` has been lifted up to the body of `onDestroy()` and further optimized there.

In the end, the shadow of the byte array is wrapped and passed to `post()`, where a sink API `OutputStream.write()` is encountered. We check the taint status of its first parameter (i.e., data packet) and query the policy service for decision if it is tainted.

```
1  public class Leakage extends Activity{
     ...
4    private static String deviceId;
I    public static boolean deviceId_s0_t;
     ...
6    public String getIMEI(BoolWrapper ret_s0_wrapper){
       ...
8      String imei = manager.getDeviceId();
9      if(imei==null){
10       imei = "";
I        imei_s0_t = false;
11     }else{
12       imei = manager.getDeviceId();
I        imei_s0_t = true;
13     }
I      ret_s0_wrapper.status = imei_s0_t;
14     return imei;
15   }
     ...
21   public void post(String addr, byte[] bytes,
        BoolWrapper bytes_s0_wrapper){
I      boolean bytes_s0_t = bytes_s0_wrapper.status;
22     OutputStream output = conn.getOutputStream();
I      boolean isAllow = false;
I      if(bytes_s0_t == true)
I        isAllow = queryPolicyService(0, 0, addr);
I      if(isAllow){
23       output.write(bytes, 0, bytes.length);}
I      else{ ... }
24     ...
25   }
     ...
35   public void onStart(){
I      BoolWrapper ret_s0_wrapper = new BoolWrapper();
I      ret_s0_wrapper.status = false;
36     Leakage.deviceId = getIMEI(ret_s0_wrapper);
I      Leakage.deviceId_s0_t = ret_s0_wrapper.status;
37   }
     ...
43   public void onDestroy(){
44     String imei = Leakage.deviceId;
45     byte[] bytes = imei.getBytes();
I      boolean bytes_s0_t = Leakage.devicdId_s0_t;
46     for(int i=0; i<bytes.length; i++){
47       bytes[i] = crypt(bytes[i]);
I        bytes_s0_t = bytes_s0_t || false;
48     }
I      BoolWrapper bytes_s0_wrapper = new BoolWrapper();
I      bytes_s0_wrapper.status = bytes_s0_t;
49     post(addr, bytes, bytes_s0_wrapper);
50   }
51 }
```

Figure 3: Java Code for the Rewritten Program

4. CONTEXT-AWARE POLICY

Once our inserted monitoring code detects an actual privacy leakage, policy service will enforce privacy policy based on user preferences. To be specific, the service app inquires user's decision upon detection and offers the user options to either "one-time" or "always" allow or deny the specific privacy breaching flow. The user can then make her decision according to her user experience and the policy manager will remember users preference for future decision making situations if the "always" option is chosen.

There exist two advantages to enforce a privacy policy with user preference history. Firstly, it associates user decisions with certain program contexts and can thus selectively restrict privacy-related information flow under different circumstances. Privacy-related outbound traffic occurs in both benign and malicious semantics. However, from dataflow analysis perspective, it is fairly hard to distinguish between, for example, a coordinates-based query towards a benign map service and a location leakage via some covert malicious process within, say, a wallpaper app. On the contrary, it is fairly straight-forward for a user to tell the difference because she knows the application semantics. With human knowledge, it is possible to avert overly strict restriction and preserve usability to the largest extent.

Secondly, it avoids repetitive warning dialogs and improves user experience. Once an "always" decision is made, this decision will be remembered and used for the same scenario next time. Thus, the user doesn't need to face the annoying dialog message over and over again for the exactly identical situation.

However, it is non-trivial to appropriately model the program context specific to a user decision and the challenge lies in the way semantics is extracted from a dataflow point of view. We hereby discuss some possible options and our solution.

4.1 Taint Propagation Trace

To achieve high accuracy, we first consider using the exact execution trace as pattern to represent a specific information flow. An execution trace can be obtained at either instruction or method level. It consists of all the instructions or methods propagating sensitive data from a source to a sink, and therefore can uniquely describe a dataflow path. When a user decision is made for a certain information flow, its execution trace is computed and saved as a pattern along with user preference. Next time when a new leakage instance is detected, the trace computation will be done on the new flow and compared with saved ones. If there exists a match, action taken on the saved one will be applied correspondingly.

Nevertheless, there exist two major drawbacks with this approach. Firstly, dynamic tracing is considerably heavy-weight. Comparison of two traces is also fairly expensive. This may affect the responsiveness of interactive mobile apps. Secondly, each dataflow instance is modeled overly precisely. Since any execution divergence will lead to a different trace pattern, even if two leakage flows occur within the same semantics, it is still difficult to match their traces. This results in repeated warning messages for semantically equivalent privacy-related dataflows.

4.2 Source and Sink Call-sites

Trace-based approach is too expensive because the control granularity is extremely fine. We therefore attempt to relax the strictness and achieve balance in the accuracy-efficiency trade-off. We propose a call-site approach which combines source-sink call-sites to model privacy flow. That is to say, information flows of same source and sink call-sites are put into one category. Once an action is taken on one leakage flow, the same action will be taken on future sensitive information flow in the same category. To this end, we introduce labels for source and sink call-sites. Information flows starting from or arriving at these call-sites are associated with corresponding labels, so that they can be differentiated based on these labels.

With a significant improvement of efficiency, this approach is not as sensitive to program contexts as the traced-based one - different execution paths can start from the same origin and end at the same sink. However, we rarely observed this inaccuracy in practice because the app execution with same source and sink call-sites usually represents constant semantics.

4.3 Parameterized Source and Sink Pairs

In addition to source/sink call-sites, parameters fed into these call-sites APIs are also crucial to the semantic contexts. For example, the user may allow an app to send data to certain trustworthy URLs but may not be willing to allow access to the others. Therefore, it is important to compare critical parameters to determine if a new observed flow matches the ones in history.

Notice that checking parameters can minimize the impact of mimicry attack. Prior research shows that vulnerable Android apps are subject to various attacks [9, 16, 18, 24, 37]. For instance, an exposed vulnerable app component can be exploited to leak private information to an attacker-specified URL. Without considering the URL parameter, it is difficult, if not possible, to distinguish internal use of critical call-sites from hijacking the same call-sites to target a malicious URL. Once a flow through some call-sites is allowed and user preference is saved, mimicking attack using same call-sites will also get approved. On the contrary, a parameter-aware approach can differentiate outgoing dataflows according to the destination URL, and thus, exploitation of a previously allowed call-sites will still raise a warning.

API Description	Source or Sink	Critical Parameter
Send data to Internet	Sink	destination URL
Send SMS message	Sink	target phone number
Query contacts database	Source	source URI

Table 1: APIs and Critical Parameters

Besides the URL of a Internet API, we also consider some other critical combinations of an API and its parameter. Table 1 summarizes our list. The target of a sink API is sensitive to security. Similar to Internet APIs, target phone numbers are crucial to `sendTextMessage()` APIs and thus need watching. On the other hand, some source call-sites also need to be distinguished according to the parameters. For instance, the API that queries contacts list may obtain different data depending on the input URI (e.g., `ContactsContract.CommonDataKinds.Phone` for phone number, `ContactsContract.CommonDataKinds.Email` for email).

4.4 Implementation

We implement the policy service as a separate app. This isolation guarantees the security of the service app and its saved policies. In other words, even if the client is exploited, the service is not affected or compromised, and can still correctly enforce privacy policies.

The service app communicates with a rewritten client app solely through Android IPC. Once a client app wants to query the service, it encapsulates the labels of source and sink call-sites as well as the specific critical parameter into an `Intent` as extra data payload. The client app is then blocked and waiting for a response. Since this transaction is usually fast, the blocking won't affect the responsiveness of the app mostly. On the service side, it decodes the data and searches for a match in its database. If there exists a match, it returns immediately with the saved action to the client. Otherwise, the service app will display a dialog message within a created `Activity`. The user decision is saved if the user prefers, or not saved otherwise. Either way, user's option is sent back to the client. On receiving the response from service, the rewritten app will either continue its execution or skip the sink call-site with respect to the reply.

It is noteworthy that we have to defend against spoofing attack and prevent forged messages from being sent to a client app. To address that, we instrument the client app to listen for the service reply with a dynamically registered `BroadcastReceiver`. When a broadcast message is received, the receiver is immediately unregistered. Thus, attack window is reduced due to this on-demand receiver registration. Further, to restrict who can send the broadcast, we protect the receiver with a custom `permission`. Broadcaster without this `permission` is therefore unable to send messages to the client app. To defeat replay attack, we also embed a session token in the initial query message, and the client app can therefore authenticate the sender of a response message.

Similarly, we need to protect a policy service from spoofing, too. The service app has to check the caller identity from a bound communication via `getCallingUid()`, so that a malicious application cannot pretend to be another app and trick the service to configure the policies for the latter.

4.5 Policy Enforcement in Running Example

We distinguish an information flow based on call-sites and their critical input. Therefore, here in this example, `queryPolicyService()` takes three parameters. The first "0" and second "0" represent the labels of source and sink call-sites, respectively. The third one is an object taking the critical input and is specific to a call-site API.

With the reply from policy service (i.e., `isAllow`), the enforcement logic devised in the app will be exercised. If the user decision is to block the dataflow, we need to properly skip the `output.write()` and disable the leakage.

5. EXPERIMENTAL EVALUATION

To evaluate the efficacy, correctness and efficiency of `Capper`, we conducted experiments on real-world Android applications. In the policy setting, we consider IMEI, owner's phone number, location, contacts to be the sensitive information sources, and network output APIs (e.g., `OutputStream.write()`, `HttpClient.execute()`, `WebView.loadUrl()`, `URLConnection.openConnection()` and `SmsManager.sendTextMessage()`) as the sinks. The action on the sink can be "block" or "allow". User

can also check an "always" option to have the same rule applied to future cases of the same semantic context. Note that this policy is mainly for demonstrating the usability of Capper. More work is needed to define a more complete policy for privacy leakage confinement.

We first present our experiment setup and in Section 5.1. We then discuss summarized analysis results in Section 5.2, and describe a detailed analysis in Section 5.3. Finally we measure the runtime performance in Section 5.4.

5.1 Experiment Setup

We collect 11,939 real-world Android applications from Google Play, and perform a preparatory package-level investigation into these apps.

Use of Native Components.

Out of the 11,939 apps, 2631 of them enclose at least one native library in the package. This shows that a small portion of Android apps (22%) need auxiliary native code to function. In addition, we discover that a large amount of these native libraries are commonly used across these apps. These native libraries are often used for video/audio processing, encoding/decoding, etc. For example, libandroidgl20.so, an OpenGL graphical library for Android system, is used by 265 apps.

Framework Compatibility.

Fragmentation is a well-known issue for Android platform. Apps are developed with different versions of framework SDK, as well as supporting libraries including Google Maps, Google Analytics, Admob, and therefore require different sets of Google APIs for analysis and instrumentation. In our experiment, we prepare a set of specific libraries which consist of Android SDK of API level 16 (Android 4.1, Jelly Bean), Google Maps of API level 16, Google Analytics SDK v2 and Google Admob SDK, version 6.2.1. We then inspect the Manifest files and exclude those apps that don't fit into our setting. In the end, we obtain 4915 applications and use them as our experiment sample set. Of course, to use Capper in the field, multiple sets of APIs are needed to accommodate diverse SDK requirements in the apps.

We then perform bytecode rewriting on these apps to enable runtime privacy protection. We conduct the experiment on our test machine, which is equipped with Intel(R) Xeon(R) CPU E5-2690 (20M Cache, 2.90GHz) and 200GB of physical memory. The operating system is CentOS 6.3 (64bit).

To verify the effectiveness and evaluate runtime performance of the rewritten apps, we further run them on a real device. Experiments are carried out on Google Nexus S, with Android OS version 4.0.4.

5.2 Summarized Analysis Results

Figure 4 illustrates the partition of 4915 realworld apps. Amongst these apps, Capper did not finish analyzing 314 of them within 30 minutes. These apps are fairly large (many over 10 MB). Application-wide dataflow analysis is known to be expensive for these large programs. We further extended the analysis timeout to 3 hours, and 122 more apps were successfully analyzed and rewritten. Given sufficient analysis time, we believe that the success rate can further increase from currently 96% to nearly 100%.

Out of the 4723 apps that were completely processed by Capper, 1414 apps may leak private information, according to our static analysis, and Capper successfully performed bytecode rewriting on them. We observed that most of them leak IMEI and location information. These types of information are frequently sent

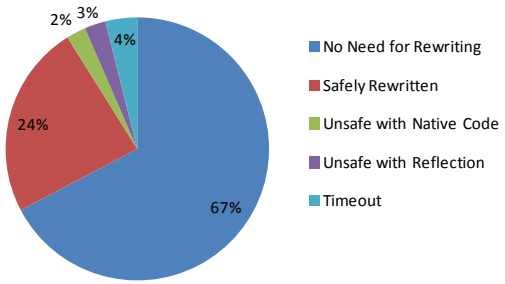


Figure 4: Bytecode Rewriting Results on 4915 Realworld Android Apps

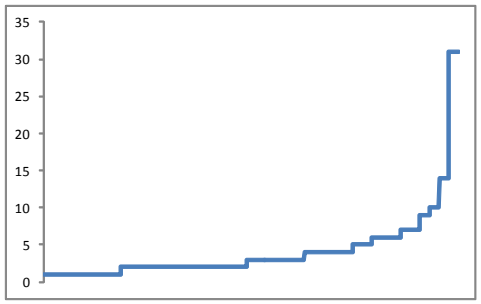
out by apps due to analytic or advertisement reasons. Apps may also sometimes leak owner's phone number or phone numbers from contacts. For the rest of them (67%), static analysis couldn't find a viable path from the sensitive information sources to the network output sinks. It means that these apps do not leak private information, so no rewriting is needed for these apps.

For these 1414 apps that were rewritten, we further investigate their use of native code and reflection. Our study shows that unknown native code is invoked within the taint slices for 118 (2%) apps. As a bytecode-level solution, our system cannot keep track of information flow processing in the native code. So the information flow may be broken on these unknown native calls. The rest 3% contain reflective calls within the slices. If the class name and method name cannot be resolved statically, we do not know how information is propagated through this function. Therefore, totally 5% apps may be unsafe and may not be fully enforced with the specified policies. The best suggestion for the end user is not to use these unsafe rewritten apps due to the potential incompleteness in policy enforcement. More discussion on this issue is presented in Section 6.

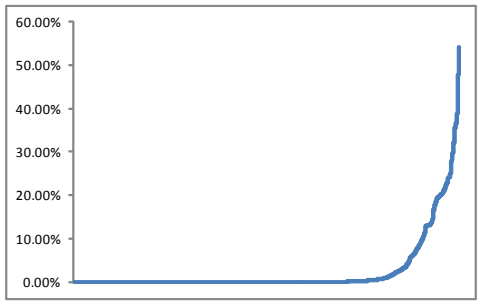
We compute the taint propagation slice for each single leakage instance, and conduct a quantitative study on them. Figure 5a gives the number of generated taint propagation slices for every app. While most of the apps retrieve privacy-related information moderately, some apps leak user's privacy through up to 31 taint slices. Such apps usually enclose various Ads libraries, each of which acquires private data separately.

Figure 5b shows the proportional size of the slices, compared to the total size of the application. We can see that most of the program slices represent a small portion of the entire application, with the average proportion being 2.48%. However, we do observe that for a few applications, the slices take up to 54% of the total size. Some samples (e.g., com.tarsin.android.dilbert) are fairly small. Although the total slice size is only up to several thousands Jimple statements, the relative percentage becomes high. Apps like de.joergjahnke.mario.android.free and de.schildbach.oeffi operate on privacy information in an excessive way, and due to the conservative nature of static analysis, many static and instance fields are involved in the slices.

We measure the program size on different stages. We observe that the increase of the program size is roughly proportional to the slice size, which is expected. After instrumentation, the increased size is, on average, 10.45% compared to the original program. Figure 6 further visualizes the impact of the four optimizations to demonstrate their performance. The six curves on the figure represent the relative size of the program, compared to the original application, on different processing stages, respectively. The baseline stands for the size of the original program, while the top curve is that of instrumentation stage. We can see that 1) for most of these apps, the increase of program size due to instrumentation is



(a) **Number of Taint Slices.** X-axis is app ID; Y-axis is the number of taint slices.



(b) **Proportional Size of Slices.** X-axis is app ID; Y-axis is the proportional size of slices.

Figure 5: Results for Taint Propagation Slices

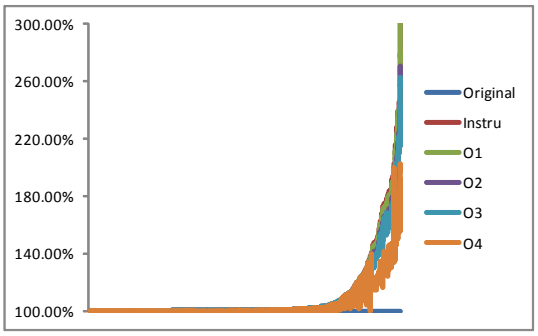


Figure 6: **Optimization Impact on LOC.** X-axis is app ID, while Y-axis is the proportional app size. The result is sorted based on Y-axis. Six curves quantify these sizes at different stages.

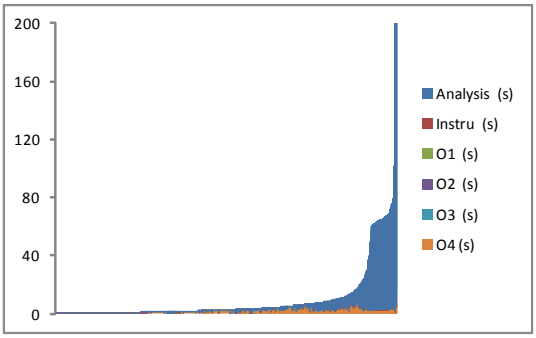


Figure 7: **Inspection & Rewriting Runtime.** X-axis is app ID, and Y-axis is the cumulative time for analysis, instrumentation and optimizations. The result is sorted based on Y-axis.

fairly small; and 2) these optimizations are effective, because they significantly reduce the program sizes. In the end, the average size of inserted code drops to 4.48%.

5.3 Detailed Analysis

Here we present the detailed analysis results of ten applications, and demonstrate the effectiveness of context-aware privacy policy enforcement. To this end, we rewrite these apps with `Capper`, and run them in a physical device along with our policy service app. We further manually trigger the privacy leakage components in the app, so that inserted code would block the program and query policy service for decision. The service will then search its own database to see if there exists a rule for the specific dataflow context of the requesting app. If a corresponding rule exists, service replies to requester immediately. Otherwise, a dialog is displayed to the user asking for decision. The user can also check the "always" option so that current decision will be saved for further reference (Figure 8). Notice that, in order to test the context-awareness of our approach, we always check this option during the experiment. Therefore, from the logcat information on the service side, we may observe and compare the number of queries an app makes with the amount of warning dialogs prompted to the user. We also compute the number of information flow contexts with trace-based model for comparison.

Table 2 lists the summarized results including number of queries, prompts and trace-based contexts. For these apps, the prompt number is often equal to or sometimes slightly smaller than the amount of trace-based dataflow contexts, while the number of queries is usually much larger than that of prompts. This means leakage contexts are modeled correctly: disparate contexts are usually treated differently in our callsite-based approach; and equivalent contexts are enforced with the same rule. The fundamental reason is that Android apps are often componentized while a separate component exercises a dedicated function.

Firstly, different types of private information are accessed through separate execution paths. Some apps (ID 2, 3) retrieve both device identifier and location information, and send them at separate sinks. Similarly, mabilo.ringtones.app leaks both geolocation data and user's phone number.

Secondly, the same type of privacy-related data can be retrieved from isolated packages but serves as different purposes. These apps (ID 4, 7, 10) consist of a main program and advertisement components, both of which produce outgoing traffic taking IMEI or location data. Take net.flixster.android as an example. In this movie app, location data is both sent to flixster server for querying theaters and to Ads server for analytical purpose.

Further, the same private data can be accessed in one package but contributes to different use cases. For instance, com.rs.autorun obtains device identifier via `getDeviceId()` API call in the first place. Then the app sends IMEI at multiple sinks to load advertisement `View`, retrieve configuration or upload analytical information including demographics, ratings, etc. Each semantic context is captured from the perspective of taint propagation trace. However, due to the use of same sink call-site, all the analytical traffics are considered to be with the same semantic context in the call-site model. Though call-site-based model is not as sensitive to context as the trace-based approach, it is still able to differentiate the contexts of Ads `View`, configuration file loading and analytical dataflow, according to disparate sink call-sites.

We also observe inconsistency between traced-based contexts and real program semantics. That lies in apps (ID 1, 6, 7, 8, 9,

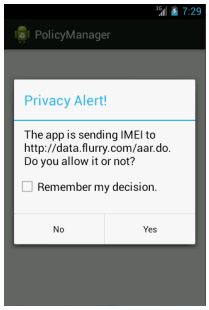


Figure 8: Warning Dialog

ID	App-Version	Queries	Prompts	Trace-based Contexts
1	artfulbits.aiMinesweeper-3.2.3	5	2	2
2	com.avantar.wny-4.1.5.1	3	2	2
3	com.avantar.yp-4.1.5	4	2	2
4	com.bfs.papertoss-1.09	3	2	2
5	com.rs.autorun-1.3.1	10	4	6
6	com.skyfire.browser-2.0.1	4	2	2
7	com.startapp.wallpaper.browser-1.4.15	5	3	3
8	mabilo.ringtones.app-3.6	5	3	3
9	mabilo.wallpapers-1.8.4	4	2	2
10	net.flixster.android-2.9.5	6	3	3

Table 2: Effectiveness of Context-aware Policy Enforcement

10) which acquire location information, where the number of trace-based contexts exceeds that of actual contexts. Geographic location is obtained either approximately from `getLastKnownLocation()`, or from a real-time update by registering a listener through `requestLocationUpdates()` and receiving updates via callback `onLocationChanged(Location)`. Some apps adopt both ways so as to achieve higher accuracy. For instance, artfulbits.aiMinesweeper reads location data by calling `getLastKnownLocation()` at the very beginning of the program, stores it into an instance field, and then periodically updates it with the aforementioned callback. Consequently, two separate paths achieve one sole purpose and thus should be considered as of equivalent context. However, from either trace or call-site point of view, there exist two separate contexts. Despite the disparity, we believe that this would at most introduce one extra prompt. Further, it is also reasonable to split this context into two, because one conducts a one-time access while the other obtains data repeatedly.

5.4 Performance Evaluation

We evaluate both offline rewriting time cost and runtime performance.

Rewriting Performance.

Figure 7 illustrates the time consumption of analysis and rewriting for 4723 Android apps in our experiment. To be specific, it depicts the execution time of analysis, instrumentation, and four optimization phases. We do not show the bytecode conversion time here, because code conversion by dex2jar is usually within several seconds and thus not significant as compared to the other steps. We find that more than 83.6% of the apps are processed within 10 seconds and the majority (92.8%) can be rewritten within 1 minute. However, still a few of them cost excessive time to finish. Global data-flow analysis dominates the overall processing time, especially for those privacy-breaching apps. In comparison, the time consumption of instrumentation and optimizations is fairly small.

Runtime Performance.

We compare the runtime overhead of bytecode rewriting with that of dynamic taint analysis in TaintDroid (on Android gingerbread version 2.3.4). In principle, if an app leaks private information only occasionally, the rewritten version would have much better performance than the original version on TaintDroid. This is because in the rewritten app nearly no instrumentation code is added on non-leaking execution paths whereas TaintDroid has to monitor taint propagation all the time.

Rather, we would like to compare the performance when the taint is actively propagated during the execution. This would be the worst-case scenario for `Capper`. Specifically, we build two customized applications for the measurement. Both leak IEMI string via `getDeviceId()` API, decode the string to a byte array, encrypt the array by doing XOR on every byte with a constant key, reassemble a string from the encrypted byte array, and in the end, send the string out to a specific URL through a raw socket interface. The only difference is that one merely sends the IMEI string, while the other also appends extra information of totally 10KB to the IMEI string before encryption. In other words, the former conducts a short-period data transfer while the latter manipulates the outgoing message within a much longer period. We expect that the execution of the first one to be mainly under `mterp` interpretation mode and the execution of the second to be boosted by JIT.

	Orig.	Orig. on TaintDroid	Rewritten
Short	30ms	130ms	34ms
Long	10583ms	15571ms	10742ms

Table 3: Runtime Performance Evaluation

We measured the execution time from `getDeviceId()` to the network interface. We observed that the rewritten application runs significantly faster than the original on TaintDroid, and only yields fairly small overhead compared to the original one running on Android, for both short-period and long-period data propagation. Table 3 illustrates the result of runtime measurement. While our approach causes 13% and 1.5% overhead for short and long data propagation respectively, TaintDroid incurs 330% and 47% overhead. The results also show that the presence of JIT significantly reduces runtime overhead, in both approaches. However, though newer version of TaintDroid (2.3.4 and later) benefits from JIT support, the overhead caused by dynamic instrumentation is still apparently high.

To further confirm the runtime overhead of the rewritten programs, we conduct an experiment on Google Nexus S, with Android OS version 4.0.4. It is worth mentioning that such verification on real device requires considerable repetitive manual efforts and thus is fairly time consuming. We therefore randomly pick 10 apps from the privacy-breaching ones, rewrite them, run both the original app and secured one on physical device, and compare the runtime performance before and after rewriting. We rely on the timestamps of Android framework debugging information (logcat logs) to compute the app load time as benchmark. The app load time is measured from when Android `ActivityManager` starts an `Activity` component to the time the `Activity` thread is displayed. This includes application resolution by `ActivityManager`, IPC and graphical display. Our observation complies with prior experiment result: rewritten apps usually have insignificant slowdown, with an average of 2.1%, while the maximum runtime overhead is less than 9.4%.

6. DISCUSSION

In this section, we discuss the limitations of our system and possible solutions. We also shed light on future directions.

Soundness of Our Bytecode Rewriting.

Our static analysis, code instrumentation, and optimizations follow the standard program analysis and compiler techniques, which have been proven to be correct in the single threading context. In the multi-threading context, our shadow variables for local variables and function parameters are still safe because they are local to each individual thread, while the soundness of shadow fields depends on whether race condition vulnerability is present in original bytecode programs. In other words, if the accesses to static or instance fields are properly guarded to avoid race condition in the original app, the corresponding operations on shadow fields are also guarded because they are placed in the same code block. However, if the original app does have race condition on certain static or instance fields, the information flow tracking on these fields may be out of sync.

We modeled Android APIs for both analysis and instrumentation. We manually generate dedicated taint propagation rules for frequently used APIs and those of significant importance (e.g., security sensitive APIs). Besides, we have general default rules for the rest. It is well-recognized that it is a non-trivial task to build a fairly complete API model, and it is also true that higher coverage of API model may improve the soundness of our rewriting. However, previous study [7] shows that a model of approximately 1000 APIs can already cover 90% of calls in over 90,000 Android applications. In addition, it is also possible to automatically create a better API model by analyzing and understanding Android framework, and we leave it as our future work.

Tracking Implicit Flow.

It is well known that sensitive information can propagate in other channels than direct data flow, such as control flow and timing channels. It is extremely challenging to detect and keep track of all these channels. In this work, we do not consider keeping track of implicit flow. This means that a dedicated malicious Android developer is able to evade `Capper`. This limitation is also shared by other solutions based on taint analysis, such as TaintDroid [12] and AppFence [19]. Serious research in this problem is needed and is complementary to our work.

Java Reflection.

A study [15] shows that many Android applications make use of Java reflection to call undocumented methods. While in 88.3% cases, the class names and method names of these reflective calls can be statically resolved, the rest can still cause problems. In our experiment, we seldom encounter this situation, because even though some apps indeed use reflective calls, they are rarely located within the taint propagation slices. That is, these reflective calls in general are not involved in privacy leakage. We could use a conservative function summary, such that all output parameters and the return value are tainted if any of input parameter is tainted, but it might be too conservative. A more elegant solution might be to capture the class name and the method name at runtime and redirect to the corresponding function summary, which enforces more precise propagation logic. We leave this as our future work.

Native Components.

Android applications sometimes need auxiliary native components to function, while, unfortunately, static bytecode-level analysis is not capable of keeping track of information flow within JNI calls. However, many apps in fact use common native components, which originate from reliable resources and are of widely recognized dataflow behavior. Thus, it is possible to model these known components with offline knowledge. In other words, we could build a database for well-known native libraries and create proper function summaries for JNI calls, and therefore exercise static data propagation through native calls with the help of such summaries.

7. RELATED WORK

In this section, we discuss previous work that is related to code rewriting and optimization and privacy leakage detection and mitigation on mobile platforms.

Bytecode Rewriting.

Many efforts are made to rewrite apps for security purposes. The Privacy Blocker application [2] performs static analysis of application binaries to identify and selectively replace requests for sensitive data with hard-coded shadow data. I-ARM-Droid [10] rewrites Dalvik bytecode, where it interposes on all the invocations of these API methods to implement the desired security policies. Aurasium [32] repackages Android apps to sandbox important native APIs and watch the applications' behaviors for security and privacy violations. Livshits and Jung [23] implemented a graph-theoretic algorithm to place mediation prompts into bytecode program and thus protect resource access. In comparison, we proposed a new bytecode rewriting technique to track information flow and enforce information-flow based policies.

Instrumentation Code Optimization.

We reduce code instrumentation overhead by performing various static analysis and optimizations. Several other works share the same insight. To find error patterns in Java source code, Martin et al. optimized dynamic instrumentation by performing static pointer alias analysis [26]. To detect numerous software attacks, Xu et al. inserted runtime checks to enforce various security policies in C source code, and remove redundant checks via compiler optimizations [33]. While we share the same insight, our work deals with Dalvik bytecode programs and specific instrumentation methods are completely different.

Privacy Leakage Detection.

Egele et al. [11] studied the privacy threats in iOS applications. They proposed PiOS, a static analysis tool to detect privacy leaks in Mach-O binaries. TaintDroid is a dynamic analysis tool for detecting and analyzing privacy leaks in Android applications [12]. It modifies Dalvik virtual machine and dynamically instruments Dalvik bytecode instructions to perform dynamic taint analysis. Enck et al. [13] proposed a static analysis approach to study privacy leakage as well. They convert a Dalvik executable to Java source code and leverage a commercial Java source code analysis tool Fortify360 [20] to detect surreptitious data flows. CHEX [24] is designed to vet Android apps for component hijacking vulnerabilities and is essentially capable of detecting privacy leakage. It converted Dalvik bytecode to WALA [4] SSA IR, and conducted static dataflow analysis with WALA framework. AndroidLeaks [17] is a static analysis framework, which also leverages WALA, and identifies potential leaks of personal information in Android applications on a large scale. Mann et al. [25] analyzed the Android API for possible sources and sinks of private data and thus identified exemplary privacy policies.

Privacy Leak Mitigation.

Based on TaintDroid, Hornyack et al. [19] proposed AppFence to further mitigate privacy leaks. When TaintDroid discovers the data dependency between source and sink, AppFence enforces privacy policies, either at source or sink, to protect sensitive information. At source, it may provide the app with fake information instead of the real one; at sink, it can block sending APIs. AppFence requires

modifications in the Dalvik virtual machine to track information flow and incurs considerable performance overhead (14% on average according to TaintDroid [12]). In contrast, `Capper` takes a bytecode rewriting approach. There is no hurdle for deployment, and due to static dataflow analysis and a series of optimizations, the new application rewritten by `Capper` can achieve higher efficiency.

8. CONCLUSION

We developed a bytecode rewriting approach to defeat privacy leakage in Android applications. Given an unknown Android app, we firstly selectively rewrite the program by inserting bytecode instructions along taint propagation slices for potential information leakage. Then to differentiate true leakage with benign sensitive dataflow, we leverage user's semantic knowledge and devise a context-aware policy enforcement mechanism. Our evaluation on 4723 real-world Android applications demonstrates that `Capper` can effectively track and mitigate privacy leaks. Moreover, after going through a series of optimizations, the instrumentation code only represents a small portion (4.48% on average) of the entire program. In addition, the runtime overhead introduced by `Capper` is also minimal, merely 1.5% for intensive data propagation.

9. ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF Grant #1018217, NSF Grant #1054605 and McAfee Inc. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] dex2jar. `http://code.google.com/p/dex2jar/`.

[2] Privacy blocker. `http://privacytools.xeudoxus.com/`.

[3] Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[4] T.j. watson libraries for analysis. `http://wala.sourceforge.net`.

[5] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, October 2011.

[6] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*, March 2011.

[7] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[8] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of the 13th International Conference on Information Security (ISC'10)*, October 2010.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security (ISC'10)*, October 2011.

[10] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proceedings of the Mobile Security Technologies Workshop (MoST'12)*, May 2012.

[11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, February 2011.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[14] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, October 2011.

[16] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[17] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12)*, June 2012.

[18] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't The Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, October 2011.

[20] C. Inc. Hp fortify source code analyzer. `http://www.cigital.com/training/elearning/fortify-source-code-analyzer`.

[21] J. Kim, Y. Yoon, K. Yi, and J. Shin. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Proceedings of the Mobile Security Technologies Workshop (MoST'12)*, May 2012.

[22] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, October 2011.

[23] B. Livshits and J. Jung. Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications. In *Proceedings of the 22th Usenix Security Symposium*, August 2013.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference*

on Computer and Communications Security (CCS'12), October 2012.

[25] C. Mann and A. Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, March 2012.

[26] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, October 2005.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, April 2010.

[28] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, November 2012.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC'09)*, December 2009.

[30] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 20th Usenix Security Symposium*, August 2012.

[31] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[32] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, August 2012.

[33] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium*, July 2006.

[34] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, November 2013.

[35] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[36] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)*, May 2012.

[37] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[39] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*, June 2011.

APPENDIX

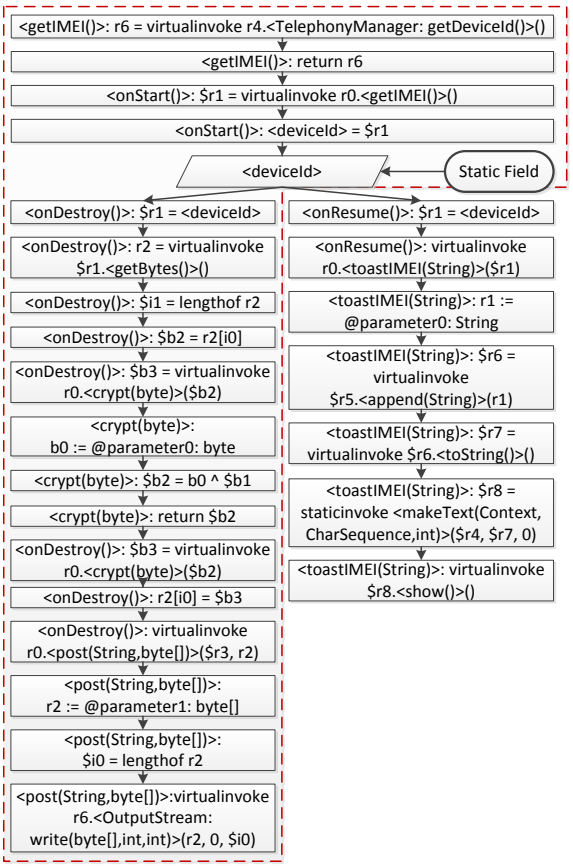


Figure 9: Taint Propagation Tree for the Running Example.

A. TAINT PROPAGATION TREE FOR RUNNING EXAMPLE

Figure 9 illustrates the result for applying the dataflow analysis on the running example. While the entire graph shows the taint propagation tree, the subgraph surrounded by dotted lines is the taint slice for the critical information flow (i.e. IMEI leakage). Each node represents a Jimple statement and its function call context. This tree starts at the invocation of `getDeviceId()` in `getIMEI()`. The source data is thus obtained and further propagated back to `onStart()` via return value. Then, in `onStart()`, IMEI is stored into a static field. Taking the static field as the source of the next iteration, the graph further expands into two branches. In the right branch, the static field is read from `onResume()` and then passed to `toastIMEI()` for display; In the left branch, the static field is accessed from `onDestroy()` which further propagates the data to `crypt()` and `post()` sequentially, and thus sends it out to the network. From this graph, we can see that the left branch is actually leading to privacy leakage and thus is our target to instrument. The right branch is irrelevant to leakage and thus removed from the taint slice.