Collaborative verification of information flow for a high-assurance app store

Michael D. Ernst, René Just, Suzanne Millstein, Werner M. Dietl,
Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros,
Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu
UW Computer Science & Engineering
sparta@cs.washington.edu

Abstract

Current app stores distribute some malware to unsuspecting users, even though the app approval process may be costly and time-consuming. We propose the creation of high-integrity app stores that provide certain guarantees to their customers. Our approach has four key features. (1) Our analysis is based upon a flow-sensitive, context-sensitive information-flow type system. (2) We use finer-grained behavioral specifications of information flow than current app stores, along with automated analysis to prove correctness with respect to the specification. (3) Our approach works on source code rather than binaries and is based on formal verification rather than on bug-finding. (4) We use a collaborative verification methodology in which the software vendor and the app store auditor each do tasks that are easy for them, reducing overall cost.

We have implemented our system for Android apps written in Java. In an adversarial Red Team evaluation, we were given 72 apps (576,000 LOC) to analyze for malware. The 57 Trojans among these had been written specifically to defeat a malware analysis such as ours, and the Red Teams had access to our source code and documentation. Nonetheless, our information-flow type system was effective: it detected 96% of malware whose malicious behavior was related to information flow and 82% of all malware. In practice our toolset would be combined with other analyses to reduce the chance of approving a Trojan. The programmer annotation burden is low: one annotation per 16 lines of code. Every sound analysis requires a human to review potential false alarms, and in our experiments, this took 30 minutes per KLOC for an auditor unfamiliar with the app.

1 Introduction

App stores make it easy for users to download and run applications on their personal devices. App stores also provide a tempting vector for an attacker. An attacker can take advantage of bugdoors (software defects that permit undesired functionality) or can insert malicious Trojan behavior into an application and upload the application to the app store.

For current app stores, the software vendor typically uploads a compiled binary application. The app store then analyzes the binary to detect Trojan behavior or other violations of the app store's terms of service. Finally, the app store approves and publishes the app. Unfortunately, the process offers few guarantees, as evidenced by the Trojans that have been approved by every major app store [3, 15, 38, 40, 47].

We are exploring the practicality of a high-assurance app store that gives greater understanding of, and confidence in, its apps' behavior. Such a store would have different approval requirements to reduce the likelihood that a Trojan is approved and distributed to unsuspecting users. Corporations already provide lists of apps approved for use by employees (often vetted by ad hoc processes). The U.S. Department of Defense is also actively pursuing the creation of high-assurance app stores.

Four contributing factors in the approval of Trojans by existing app stores are: (1) Existing analysis tools are poorly automated and hard to use: much manual, error-prone human effort is required. (2) The vendor provides only a very coarse description of application behavior in the form of permissions it will access: system resources such as the camera, microphone, network, and address book. These properties provide few guarantees about the application's behavior. (3) The binary executable lacks much semantic information that is available in the source code but has been lost or obfuscated by the process of compilation. (4) The vendor has little incentive to make the application easy for the app store to analyze.

We have developed a new approach to verifying apps that addresses each of these factors. (1) We have created powerful, flow-sensitive, context-sensitive type system that verifies information flows. The system is easy to use and works with with Java and Android. (2) Our approach provides finer-grained specifications than current app stores, indicating not just which resources may be accessed but which information flows are legal — how the resources may be used by the program. Our initial analysis focuses on confidentiality and integrity security policies that can be expressed in terms of information flow. Our tools connect information flow security policies to lightweight specifications and connect specifications to code. (3) Our approach works on source code rather than binaries, and it aims to prove that an app satisfies information flow properties, rather than to detect some bugs/malware. An analyst approves or rejects the properties. Availability of source code fundamentally changes the verification process: it provides more information, en-

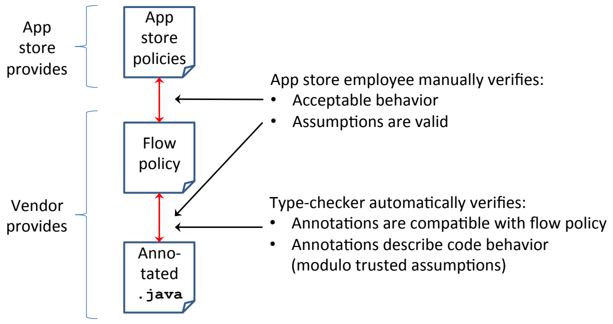
App store provides
App store policies
App store employee manually verifies:
• Acceptable behavior
• Assumptions are valid
Vendor provides
Flow policy
Type-checker automatically verifies:
• Annotations are compatible with flow policy
• Annotations describe code behavior (modulo trusted assumptions)
Anno-tated .java

Figure 1: The collaborative verification model. The flow policy is a high-level specification that expresses application behavior in terms of user-visible information flows.

ables more accurate and powerful analyses, and enables an auditor to evaluate warnings. (4) We propose a collaborative verification methodology in which the vendor participates in and contributes to the verification process, rather than casting the vendor and the app store in an antagonistic relationship. Each party provides information that is easy for them to provide, thus reducing the overall cost of verification. The developer is not trusted: all information provided by the developer is verified.

We report on initial experience with this system, including an adversarial Red Team exercise in which 5 corporate teams (funded externally, not by us) were given insider access to our source code and design documents then tasked with creating Trojans that would be difficult to detect. Our type system detected 82% of the Trojans, and 96% of the Trojans whose malicious behavior was related to information flow. (We have identified an enhancement to our system that would increase that number to 100%.) It is necessary for a human to investigate tool warnings to determine whether they are false positives. On average, it took an auditor unfamiliar with the programs 30 minutes per KLOC to analyze the information flow policy and the tool warnings. The annotation burden for programmers is also low.

Overall, our goal is to make it difficult to write Trojans and easy to determine when code is not a Trojan. Our tools cannot catch all malware, but they raise the bar for malware authors and thus improve security.

1.1 Verification of source code

An app store can be made more secure by requiring vendors to provide their applications in source code, and then performing strong verification on that source code. The app store would analyze the source code, compile it, and distribute it as a binary (signed by the app store's private key) to protect the vendor's intellectual property. Availability of source code fundamentally changes the approval process in favor of verification by providing more information to both the analysis and the analyst.

Source code verification is relevant for other domains than high-integrity application stores. One public example of inserting malicious behavior into an open source program is an attempt to insert a backdoor in the Linux kernel [24]. As another example, Liu et al. developed proof-of-concept malware as Chrome extensions [26], which are essentially distributed as source code. We believe that source code analysis for security will become increasingly important, so it is worthy of attention from security researchers.

1.2 Collaborative verification model

The app store's goal is twofold: to prevent approving malicious applications, and to approve non-malware with a minimum of cost and delay.

Most app store approval policies assume an adversarial, or at least non-cooperative, relationship between the developer and the app store. The developer delivers an app in binary form, and the app store uses an opaque process to make a decision about whether to offer the app on the app store.

We propose a collaborative model (Fig. 1) in which the application vendor provides more information to the auditor (an app store employee). This information is easy for the application vendor to provide, but it would be difficult for the auditor to infer. The auditor is able to make a decision more quickly and with greater confidence, which is advantageous to both parties.

As shown in Fig. 1, the auditor receives two artifacts from the vendor. The first vendor-provided artifact is the flow policy, a high-level specification of the intended information flows in the program from the user point of view. In our experiments, this averaged 6 lines long. For example, it might state that location information is permitted to flow to the network and that camera images may be written to the local disk. Any information flow not stated in the flow policy file is implicitly forbidden. The second vendor-provided artifact is the source code, annotated to show which parts of the program implement or participate in the information flows. The annotation burden is low: on average one annotation per 16 lines of code.

The annotations are untrusted. Our information-flow type-checker, Information Flow Checker (IFC), automatically ensures that the annotations are both permitted by the flow policy file and are an accurate description of the source code's behavior (modulo the trusted assumptions). If not, the app is rejected. Thus, the application vendor must provide accurate annotations and flow policy.

The auditor has two tasks, corresponding to the two vendor-provided artifacts. The first task is to evaluate the app's flow policy. This is a manual step, in which the auditor compares the flow policy file to the app's documentation and to any app store or enterprise policies. The app store analyst must approve that the requested flows are reasonable given the app's purpose; apps with unreasonable flow policies are rejected as potential Trojans. The second task is to verify each trusted assumption, using the verification methodology of his/her choice (e.g., [2]). Sect. 3.11 further describes the auditing process.

Not every app store will desire to differentiate itself through increased security, and not every vendor will desire to participate in high-assurance app stores. But market forces will enable such stores to exist where there are appropriate economic incentives — that is, whenever some organizations or individuals are willing to pay more for increased security. Large organizations already require their vendors to provide and/or escrow source code.

It makes economic sense for the vendor to annotate their code and possibly to be paid a premium: based on our experience, the effort is much less for the author of the code than for an auditor who would have to reverse-engineer the code before writing down the information about the information flows. The effort is small compared to overall development time and is comparable to writing types in a Java program. If the annotations are written as the code is first developed, they may even save time by preventing errors or directing the author to a better design.

The U.S. Department of Defense is also interested in high-assurance app stores, for example through DARPA's "Transformative Apps" and "Automated Program Analysis for Cybersecurity", along with related software verification programs such as "High-Assurance Cyber Military Systems" and "Crowd-Sourced Formal Verification" Our collaborative verification model is novel and differs from DARPA's existing programs.

1.3 Threat model

While there are many different types of malicious activities, we focus on Trojans whose undesired behavior involves information flow from sensitive sources to sensitive sinks. This approach is surprisingly general: we have found that our approach can be adapted to other threats, such as detecting when data is not properly encrypted, by treating encryption as another type of resource or permission.

More specifically, IFC uses the flow policy as a specification or formal model of behavior. If IFC issues no warnings, then the app does not permit information flows beyond those in the flow policy — that is, each output value is affected only by inputs specified in the flow policy. Manual checking is required for any trusted assumptions or IFC warnings. IFC does not perform labor-intensive full functional verification, only information-flow verification, which we show can be done at low cost.

Our threat model includes the exfiltration of personal or sensitive information and contacting premium services. However, it does not cover phishing, denial of service, or side channels such as battery drain or timing. It does not address arbitrary malware (such as Slammer, Code Red, etc.). We treat the operating system, our type checker, and annotations on unverified libraries as trusted components — that is, if they have vulnerabilities or errors, then an app could be compromised even if it passes our type system. App developers are not trusted.

Our approach is intended to be augmented by complementary research that focuses on other threats: it raises the bar for attackers rather than providing a silver bullet. Sect. 2.10 discusses limitations of our system in greater detail.

There have been previous studies of the kinds of malware present in the wild [13, 52]. Felt et al. [13] classify malware into 7 distinct categories based on behavior. Our system can catch malware from the 4 most prevalent and important ones: stealing user information (60%), premium calls or SMSs (53%), sending SMS advertising spam (18%), and exfiltrating user credentials (9%). The other 3 categories are: novelty and amusement (13%), search engine optimization (2%), ransom (2%).

1.4 Contributions

The idea of verifying information flow is not new, nor is using a type system. Rather, our contributions are a new design that makes this approach practical for the first time, and realistic experiments that show its effectiveness. In particular, the contributions are:

We have proposed a collaborative verification model that reduces cost and uncertainty, and increases security, when approving apps for inclusion in an app store. Our work explores a promising point in the tradeoff between human and machine effort.

We have extended information-flow verification to a real, unmodified language (Java) and platform (Android). Our design supports polymorphism, reflection, intents, defaulting, library annotations, and other mechanisms that increase expressiveness and reduce human effort.

We have designed a mechanism for expressing information flow policies, and we have refined the existing Android permission system to make it less porous.

We have implemented our design in a publicly-available system, and we have experimentally evaluated our work. Our system effectively detected realistic malware targeted against it, built by skilled Red Teams with insider knowledge of our system. The effort to use our system was low for both programmers and auditors: it is powerful, yet it requires less annotation overhead than previous systems and is simpler to use and understand.

2 Information Flow Checker

This section describes our system Information Flow Checker, IFC. IFC gives a guarantee that there are no information flows in a program beyond those expressed in a high-level specification called a flow policy. IFC is sound and conservative: if IFC approves a program, then the program has no undesired information flows, but if IFC issues a warning, then the program might or might not actually misbehave at run time. The guarantee is modulo human examination of a small number of trusted assumptions to suppress false positive warnings, including ones about indirect flow through conditionals.

As shown in Fig. 1, a programmer using IFC provides two kinds of information about the information flows in the program. First, the programmer provides a flow policy file, which describes the types of information flows that are permitted in the program (see Sect. 2.3). For example, a simple app for recording audio to the file system would have a flow policy containing only RECORD_AUDIO→ FILESYSTEM. It would be suspicious if this app's flow policy contained RECORD_AUDIO→INTERNET, because that flow allows audio to be leaked to an attacker's server.

Second, the programmer writes Java annotations to express the information flow properties of method signatures and fields. Each annotated type includes a set of sensitive sources from which the data may have originated and a set of sinks to which the data may be sent. For example, the programmer of the audio recording app would annotate the recorded data with `@Source(RECORD_AUDIO) @Sink(FILESYSTEM)`. IFC uses type-checking over an information flow type system to verify that the annotated code is consistent with the flow policy file.

2.1 Types: flow sources and sinks

The type qualifier `@Source` on a variable's type indicates what sensitive sources might affect the variable's value. The type qualifier `@Sink` indicates where (information computed from) the value might be output. These qualifiers can be used on any occurrence of a type, including in type parameters, object instantiation, and cast types.

As an example, consider the declaration

```
@Source(LOCATION) @Sink(INTERNET) double loc;
```

The type of variable `loc` is `@Source(LOCATION) @Sink(INTERNET) double`. The type qualifier `@Source(LOCATION)` indicates that the value of `loc` might have been derived from location information. Similarly, the type qualifier `@Sink(INTERNET)` indicates that `loc` might be output to the network. A programmer typically writes either `@Source` or `@Sink`, but not both; see Sect. 2.6.

The arguments to `@Source` and `@Sink` are permissions drawn from our enriched permission system (Sect. 2.2). The argument may also be a set of permissions to indicate that a value might combine information from multiple sources or flow to multiple locations.

2.2 Comparison to Android permissions

IFC's permissions are finer-grained than standard Android manifest permissions in two ways. First, Android permits any flow between any pair of permissions in the manifest — that is, an Android program may use any resource mentioned in the manifest in an arbitrary way. Second, IFC refines Android's permission, as we now discuss.

IFC's permissions are not enforced at run time as Android permissions are (potentially resulting in an exception during execution). Rather, they are statically guaranteed at compile time. Even if an app inherited a permission from another app with the same sharedUserId, IFC will require that permission be listed in the flow policy.

2.2.1 Restricting existing permissions

Android's INTERNET permission represents all reachable hosts on the Internet, which is too coarse-grained to express the developer's intention. IFC allows this permission to be parameterized with a domain name, as in INTERNET("*.google.com"). Other permissions can be parameterized in a similar style in which the meaning of the optional parameter varies based on the permission it qualifies. For example, a parameter to FILESYSTEM represents a file or directory name or wildcard, whereas the parameter to SEND_SMS represents the number that receives the SMS. Other permissions that need to be parameterized include CONTACTS, *_EXTERNAL_FILESYSTEM, NFC, *_SMS, and USE_SIP, plus several of those described in Sect. 2.2.2, such as USER_INPUT to distinguish sensitive from non-sensitive user input.

2.2.2 Sinks and sources for additional resources

IFC adds additional sources and sinks to the Android permissions. For example, IFC requires a permission to retrieve data from the accelerometer, which can indicate the user's physical activity, and to write to the logs, which a colluding app could potentially read. Table 1 lists the additional sources and sinks. We selected and refined these by examining the Android API and Android programs, and it is easy to add additional ones. Our system does not add much complexity: it only adds 26 permissions to Android's standard 145, or 18% more permissions.

Some researchers feel that the Android permission model is already too complicated for users to understand [12]. But our perspective is that of a full-time auditor who is trained to analyze applications. The flow policy is examined once per application by that skilled engineer, not on every download by a user, so the total human burden is less. (See Sect. 3.11 for empirical measurements.)

Table 1: Additional sources and sinks used by IFC, beyond the built-in 145 Android permissions.

Sources	Sinks	Both source and sink
ACCELEROMETER	CONDITIONAL	CAMERA_SETTINGS
BUNDLE	DISPLAY	CONTENT_PROVIDER
LITERAL	SPEAKER	DATABASE
MEDIA	WRITE_CLIPBOARD	FILESYSTEM
PHONE_NUMBER	WRITE_EMAIL	PARCEL
RANDOM	WRITE_LOGS	PROCESS_BUILDER
READ_CLIPBOARD		SECURE_HASH
READ_EMAIL		SHARED_PREFERENCES
READ_TIME		SQLITE_DATABASE
USER_INPUT		SYSTEM_PROPERTIES

The more detailed flow policy file yields more insight than simple Android permissions, because the flow policy (Sect. 2.3) makes clear how each resource is used, not just that it is used.

We now discuss two permissions, LITERAL and CONDITIONAL, whose meaning may not be obvious.

Literals The LITERAL source is used for programmer-written manifest constants, such as `"Hello world!"`. This enables IFC to distinguish information derived from the program source code from other inputs. Manifest literals are used benignly for many purposes, such as configuring default settings. The flow policy shows how they are used in the program, and they can be examined by the analyst.

Conditionals Indirect information flow through conditionals can leak private information. For example, consider the following code and a flow policy containing LITERAL→INTERNET and USER_INPUT→FILESYSTEM:

```
@Source(USER_INPUT) @Sink(FILESYSTEM)
long creditCard = getCCNumber();
final long MAX_CC_NUM = 9999999999999999;
for (long i = 0 ; i < MAX_CC_NUM ; i++) {
    if (i == creditCard)
        sendToInternet(i);
}
```

We investigated two mechanisms for bringing indirect flows to the attention of an auditor who can determine whether they are malicious. (As noted earlier, IFC's guarantees are modulo human examination of places its type system is conservative.) The first mechanism is the classic approach of Volpano [46]: taint all computations in the dynamic scope of a conditional with all the sources/sinks from the conditional's predicate. This includes all statements in the body of the conditional and all statements in any method called by the conditional. One downside of this mechanism is over-tainting of computations within the scope of conditionals, which leads to false positive alarms and extra effort by auditors to review them. The need to analyze (and possibly over-taint) all methods called within conditionals is also a disadvantage.

The second mechanism is to introduce a new CONDITIONAL sink. The type of a conditional expression must include the sink CONDITIONAL. The type-checker permits only uses of sensitive sinks that are permitted by the flow policy file. A disadvantage is that every one of those uses must be treated as a warning that is a possible false alarm and must be reviewed by the auditor. In our experiments (Sect. 3.11), 27% of conditionals used sensitive sources, or fewer than 23 per KLOC; the others used only LITERAL.

In each case, similar auditor effort is required: warnings lead the auditor to examine the same conditional expressions and bodies to ensure that no sensitive information leaks indirectly. The first approach focuses auditor effort on the statements in the conditional body (or in bodies of methods transitively called by the conditional body) where over-tainting leads to false positive type-checker warnings, but the auditor must also consider the conditional expression. The second approach focuses auditor effort on the conditional expression, but the auditor must also consider all effects of the conditional body.

The auditors in our experiments (Sect. 3.11) felt that the second approach, with the new CONDITIONAL sink, was easier for them. They preferred to think about an entire conditional expression at once rather than statement-by-statement. Over-tainting within method bodies could be difficult to trace back to specific, seemingly-unrelated, conditional expressions. Oftentimes, examining a conditional expression enabled the auditors to rule out bad behavior without needing to examine all the assignments in its dynamic scope; this was particularly true for simple conditionals such as tests against `null`.

As future work, we plan to improve each approach, and combine them, to further reduce auditor effort.

2.3 Flow policy

A flow policy is a list of all the information flows that are permitted to occur in an application. A flow policy file expresses a flow policy, as a list of *flowsource* → *flowsink* pairs. Just as the Android manifest lists all the permissions that an app uses, the flow policy file lists the flows among permissions and other sensitive locations. The flow policy file is best written by the original application vendor, just as the Android manifest is.

Consider the "Block SMS" application of Table 5, which blocks SMS messages from a blacklist of blocked numbers and saves them to a file for the user to review later. Its policy file must contain READ_SMS→FILESYSTEM to indicate that information obtained using the READ_SMS permission is permitted to flow to the file system.

The flow policy restricts what types are legal Every flow in a program is explicit in the types of the program's expressions. For example, if there is no expression whose type has the type qualifiers `@Source(CAMERA) @Sink(INTERNET)`, then the program never sends data from the camera to the network (modulo conditionals and transitive flows, which are discussed elsewhere). The expression's type might be written by a programmer or might be automatically inferred by IFC.

IFC guarantees that there is no information flow except what is explicitly permitted by the policy file. If the type of a variable or expression indicates a flow that is not permitted by the policy file, then IFC issues a warning even if the program otherwise would type-check. For example, the following declaration type-checks, but IFC would still produce an error unless the policy file permits the CAMERA→INTERNET flow:

```
@Source(CAMERA) @Sink(INTERNET) Video video = getVideo();
```

Transitive flows IFC forbids implied transitive flows. If a flow policy permits CAMERA→FILESYSTEM and FILESYSTEM→INTERNET, then it must also include the transitive flow CAMERA→INTERNET, because the application may record from the camera into a file and then send the contents of the file over the network. The developer must justify the purpose of each flow or convince the app store that the flow is not used. Parameterized permissions (Sect. 2.2.1) reduce the number of transitive flows; for example, the FILESYSTEM permissions in our example would probably refer to different files, so no transitive flow would be possible nor required in the flow policy file.

2.4 Subtyping

A type qualifier hierarchy indicates which assignments, method calls, and overridings are legal, according to standard object-oriented typing rules.

`@Source(`*B*`)` is a subtype of `@Source(`*A*`)` iff *B* is a subset of *A* [7]. For example, `@Source(INTERNET)` is a subtype of `@Source({INTERNET, LOCATION})`. This rule reflects the fact that the `@Source` annotation places an upper bound on the set of sensitive sources that were actually used to compute the value. If the type of `x` contains `@Source({INTERNET, LOCATION})`, then the value in `x` might have been derived from both `INTERNET` and `LOCATION` data, or only from `INTERNET`, or only from `LOCATION`, or from no sensitive source at all.

The opposite rule applies for sinks: `@Sink(`*B*`)` is a subtype of `@Sink(`*A*`)` iff *A* is a subset of *B*. The type `@Sink({INTERNET, LOCATION})` indicates that the value is permitted to flow to both `INTERNET` and `FILESYSTEM`. This is a subtype of `@Sink(INTERNET)`, as the latter type provides fewer routes through which the information may be leaked.

2.5 Polymorphism

Information flow type qualifiers interact seamlessly with parametric polymorphism (Java generics). For example, a programmer can declare

```
List<@Source(CONTACTS) @Sink(SMS) String> myList;
```

Here, the elements of `myList` are strings that are obtained from `CONTACTS` and that may flow to `SMS`.

IFC also supports qualifier polymorphism, in which the type qualifiers can change independently of the underlying type. For example, this allows a programmer to write a generic method that can operate on values of any information flow type and return a result of a different Java type with the same sources/sinks as the input.

Parametric polymorphism, qualifier polymorphism, and regular Java types can be used together. The type system combines the qualifier variables and the Java types into a complete qualified type. Although extensions to the type system are always possible, we have found our system effective in practice thus far.

2.6 Inference and defaults

A complete type consists of a `@Source` qualifier, a `@Sink` qualifier, and a Java type. To reduce programmer effort and code clutter, most of the qualifiers are inferred or defaulted. A programmer need not write qualifiers within method bodies, because such types are inferred by IFC. Even for method signatures and fields, a programmer generally writes either `@Source` or `@Sink`, but not both. We now explain these features. For experimental measurements, see Sect. 3.10.

2.6.1 Type inference and flow-sensitivity

A programmer does not write information flow types within method bodies. Rather, local variable types are inferred.

IFC implements this inference via flow-sensitive type refinement. Each local variable declaration (also casts, `instanceof`, and resource variables) defaults to the top type, `@Source(ANY) @Sink({})`. At every properly-typed assignment statement, the type of the left-hand side expression is flow-sensitively refined to that of the right-hand side, which must be a subtype of the left-hand side's declared type. The refined type applies until the next side effect that might invalidate it.

IFC limits type inference to method bodies to ensure that each method can be type-checked in isolation, with a guarantee that the entire program is type-safe if each method has been type-checked. It would be possible to perform a whole-program type inference, but such an approach would be heavier-weight, would need to be cognizant of cooperating or communicating applications, and would provide fewer documentation benefits.

2.6.2 Determining sources from sinks and vice versa

If a type contains only a flow source or only a flow sink, the other qualifier is filled in with the most general value that is consistent with the policy file. If the programmer writes `@Source(α)`, IFC defaults this to `@Source(α) @Sink(ω)` where ω is the set of flow sinks that all sources in α can flow to. Similarly, `@Sink(ω)` is defaulted to `@Source(α) @Sink(ω)` where α is the set of flow sources allowed to flow to all sinks in ω. Defaults are not applied if the programmer writes both a source and a sink qualifier.

Suppose the flow policy contains the following:

```
A -> X,Y
B -> Y
C -> Y
```

Then these pairs are equivalent:

```
@Source(B,C)   =   @Source(B,C) @Sink(Y)

     @Sink(Y)   =   @Source(A,B,C) @Sink(Y)
```

This mechanism is useful because oftentimes a programmer thinks about a computation in terms of only its sources or only its sinks. The programmer should not have to consider the rest of the program that provides context indicating the other end of the flow.

This defaulting mechanism is essential for annotating libraries. IFC ships with manual annotations for 10,470 methods of the Android standard library. Only .0007% of methods use both a `@Source` and a `@Sink` annotation. An example of a method that uses only a `@Source` annotation is the `File` constructor: a newly-created readable file should be annotated with `@Source(FILESYSTEM)`, but there is no possible `@Sink` annotation that would be correct for all programs. Instead, the `@Sink` annotation is omitted, and our defaulting mechanism provides the correct value based on the application's flow policy.

This mechanism can be viewed as another application of type polymorphism.

2.6.3 Defaults for unannotated types

Table 2 shows the default qualifiers for completely unannotated types. When the default is only a source or only a sink, the other qualifier is inferred from the policy file as explained in Sect. 2.6.2.

Most unannotated types (including field types, return types, generic type arguments, and non-`null` literals) are given the qualifier `@Source(LITERAL)`. This is so that simple computation involving manifest literals, but not depending on Android permissions, does not require annotations.

As is standard, the `null` literal is given the bottom type qualifier, which allows it to be assigned to any variable. For IFC, the bottom type qualifier is `@Source({}) @Sink(ANY)`.

The bytecode indicates whether a library method was given no `@Source` annotation and no `@Sink` annotation (in which case it is defaulted exactly as above) or has not yet been examined by a human to write a summary. Unexamined methods are conservatively given a special type that guarantees a type-checking error, thus signaling to the developer the need to annotate that library method.

Table 2: Default flow qualifiers for unannotated types.

Location	**Default flow qualifier**
Method parameters	`@Sink(CONDITIONAL)`
Method receivers	`@Sink(CONDITIONAL)`
Return types	`@Source(LITERAL)`
Fields	`@Source(LITERAL)`
`null`	`@Source({}) @Sink(ANY)`
Other literals	`@Source(LITERAL)`
Type arguments	`@Source(LITERAL)`
Local variables	`@Source(ANY) @Sink({})`
Upper bounds	`@Source(ANY) @Sink({})`
Resource variables	`@Source(ANY) @Sink({})`

IFC allows a developer to choose different default qualifiers for a particular method, class, or package, and for specific locations as in Table 2.

2.7 Trusted assumptions to suppress false positive warnings

Every sound static analysis is conservative: that is, there exists source code that never misbehaves at run time, but the static analysis cannot prove that fact and issues a warning about possible misbehavior. Every cast in a Java program is an example of such conservatism in the Java type system. For example, application invariants might guarantee a specific property about some datum that is stored in a heterogeneous container. IFC, being conservative, assumes that information is implicitly leaked in this case and issues a warning, which might be a false positive. In 11 Android apps (9437 LOC), IFC suffered 26 false positives, or fewer than 3 per KLOC (see Sect. 3.10).

A programmer who determines that one of IFC's warnings is a false positive can disable the warning by writing a trusted assumption using Java's standard `@SuppressWarnings` mechanism. The vendor is expected to write a justification for each trusted assumption.

The app store auditor manually uses other (non-IFC) techniques to verify each trusted assumption. The auditor validates the vendor's claim that the code is well-behaved for some reason that is beyond the precision of the type checker. Such a step is required for any static analysis, not just IFC.

2.8 Indirect control flow: reflection, intents

Indirect control flow, for example in reflection and intents, is challenging for a static analysis. IFC soundly handles these constructs.

2.8.1 Reflection

IFC analyses Java reflection to determine the target method of a reflective call. This enables a downstream analysis, such as IFC's information-flow type-checking, to treat the reflective code as a direct method call, which has a much more precise annotated signature than does `Method.invoke`. The library's conservative annotations ensure that any unresolved reflective call is treated soundly.

The reflection analysis first performs constant folding and propagation for string, integer, and array types, and also for classes such as `Class` and `Method`. The constant folding handles not only basic computations like addition and string concatenation, but also method calls, even into program code, whose results depend only on constant arguments.

The analysis resolves the reflective call to a single concrete method in 96% of cases in our experiments, including malicious examples where reflection is used intentionally as a form of code obfuscation. Additionally, the constant analysis automatically determined the value of an obfuscated phone number in an app that used Base64 encoding to hide the value from other forms of static analysis.

In our experiments, 17 out of 72 apps accessed a sensitive API using reflection.

2.8.2 Intents

Intents are an Android mechanism for interprocess communication, and they can also create processes (Android activities). An intent carries a payload of data to some other process. An activity can register to receive arbitrary intents.

To handle intents, we extended IFC with map types (similar to record types) that represent the mappings of data in an intent payload. Type inference makes annotations unnecessary at most intent-sending operations.

In order to type-check communication, an interface specification is necessary. The overriding implementation of intent-receiving methods acts as this interface and permits modular checking. Even if new apps are added to the app store later, previous apps need not be re-checked. We leverage previous work to determine the possible targets for an intent-sending method [34].

In our experiments, 3 apps exploited the `ACTION_VIEW` intent to access a URL without the INTERNET permission.

2.9 Implementation

IFC is implemented as a pluggable type system built on top of the Checker Framework [8]. The implementation of IFC consists of 3731 lines of Java, plus annotations for 10,470 library methods. IFC's source code is available at `http://types.cs.washington.edu/sparta/release/`.

2.10 Limitations

IFC is focused on Trojans that cause an undesired information flow, as indicated by the threat model of Sect. 1.3. This section discusses further limitations. IFC should be used in conjunction with complementary techniques that address other security properties.

As with any static analysis, IFC's soundness guarantee only extends to code that is analyzed at compile time. Use of native code and un-analyzed Android activities requires a different analysis or trusted annotations that describe the information flows enabled by those components. IFC forbids dynamic code loading.

Our cooperative verification model means that the vendor knows one of the techniques that the app store will use to verify an app. This knowledge might permit a malicious developer to design Trojans that are beyond the capabilities of IFC or that exploit IFC's limitations.

As with many security mechanisms, human judgment can be a weak link. A malicious developer could write a misleading explanation for an information flow in the flow policy or for a trusted assumption in the code, in an effort to convince the auditor to approve malware. Our work also does not address how to decide upon an app store's policies (Fig. 1) nor whether an app is worthy of inclusion in the app store.

Despite these limitations, use of IFC increases the difficulty of hiding Trojans in source code, and the requirement that code be accepted by IFC may also make the Trojan more likely to be detected using other tools or manual verification.

3 Empirical study

3.1 Methodology

The sponsor of our research wished to evaluate our tools and approach. To this end, they hired five development companies (in the following referred to as Red Teams) to create Android applications with and without Trojans. We had neither control over the Red Teams nor any knowledge of the malware they were creating. While they were creating the malware, the Red Teams had access to a current version of IFC, including source code, documentation, and our own analysis of IFC's vulnerabilities. A total of 20 people worked on the Red Teams. On average they had more than 2 years of Android experience. Other than two interns, they hold BS or MS degrees and work full-time as computer security analysts. Most have been exposed to information flow theory, with the maximum experience being 6 years working with information flow.

The Red Teams created both malware and non-malware apps. The malware had to be written in Java. The Red Teams started out by surveying real-world mobile malware. They tried to produce diverse malware, including malware that is representative of that found in the wild, novel malware that they devised, and malware specifically targeting the limitations of IFC. They had two goals: to evaluate how well IFC might work in practice, and to see how IFC could be defeated.

Overall, the Red Teams created 72 Java applications. Our sponsor provided us with the apps in five batches over an eight-month period from December 2012 to July 2013. For each batch, we were given a limited amount of time to analyze the applications with our tools. We wrote a report stating whether each application was malware or not, and describing the malware (including its trigger and effect) or justifying our conclusion that there was no malware. The Red Teams were given our results for the first three batches, and they used this information to create malware that was harder for our tools to find.

We received the applications in source code form. IFC does not run the applications. The applications were not obfuscated, but they were also not well-documented, and the Red Teams had no motivation to make them understandable. The user documentation was only a few sentences stating the general purpose of the app, but usually omitting significant details about the functionality — considerably less than a typical app has in an app store. The code documentation was just as bad: the source code often used poor style; code comments and design documentation were absent; and the apps contained neither flow policies nor the information flow annotations used by our tools (Sect. 2). Thus, we spent most of our time reverse-engineering the apps to understand their purpose, operation, and implementation, and relatively less time searching for malware.

3.2 Summary of results

IFC detected 96% of the 47 apps that contain malicious information flow. The 72 total apps produced by the Red Teams were as follows.

- 15 are non-malicious.
- 18 use a source or sink that is at odds with the application's description; see Sect. 3.3.
- 11 use an information flow between Android permissions that is at odds with the application's description; see Sect. 3.4.
- 8 use an information flow involving our new sources or sinks (Sect. 2.2.2) that is at odds with the application's description; see Sect. 3.5.
- 10 use an information flow involving parameterized sources or sinks (Sect. 2.2.1) that is at odds with the application's description; see Sect. 3.6.
- 10 are not detected by IFC because the malware is not related to information flow; see Sect. 3.7.

Table 5 describes the 57 apps that contain malware.

3.3 Unjustified permissions

Of the 57 malicious applications, 18 applications use a permission that cannot be justified for the application, based on its description. These unjustified permissions are grounds for rejection from a high-assurance app store. For example, the SMS Backup app lists READ_BROWSER_HISTORY in the Android manifest file, but the description of the app does not explain why.

Not all of this malware is apparent from the Android manifest. For example, the SMS Notification app's unjustified permission is WRITE_LOGS, one of the new sinks that IFC adds.

3.4 Unjustified information flows

For 11 apps, all the Android permissions requested are justified based on the description of the app. Malicious information flow becomes apparent only after a flow policy is written and verified using IFC.

For example, 2D Game has a malicious flow, READ_EXTERNAL_STORAGE→INTERNET. The app is allowed to access the external storage to load photos in the game, so READ_EXTERNAL_STORAGE is justified. The app description states that the app sends high scores to a leaderboard on a server, so INTERNET is justified. The description says nothing about uploading the photos directly to the server, nor would user expect a simple game to do so. Therefore, READ_EXTERNAL_STORAGE→INTERNET is a malicious flow.

The writer of Calculator 1 tried to hide use of the INTERNET by stating that the application automatically checks for updates. IFC still caught the malware due to an unjustified information flow USER_INPUT→FILESYSTEM.

3.5 Information flows using new sources/sinks

For 8 apps, the malicious information flow is apparent only via use of the additional permissions listed in Table 1. For example, RSS Reader has a malicious flow of RANDOM→VIBRATE. The description of the app gives no valid reason to use a random number, but because RANDOM is not an Android permission, the manifest file does not list it. The app is supposed to vibrate the phone when one of the user's feeds is updated, so VIBRATE is listed in the manifest file as expected. However, the app's user would not expect the app to cause random vibrations, so RANDOM→VIBRATE is malicious.

3.6 Flows using parameterized permissions

For 10 apps, the malicious information flow is apparent only via use of parameterized permissions (Sect. 2.2.1). For example, in GPS 3, the location data should only flow to `maps.google.com`, but it also flows to `maps.google-cc.com`. To express this, the flow policy lists LOCATION→INTERNET("maps.google.com") but not LOCATION →INTERNET("maps.google-cc.com"). Another app, Geocaching, should only send data from specific geocaching NFC tags to the server, but it collects all NFC tags in range and sends them to the server, NFC("*")→INTERNET.

For two of these apps, PGP Encryption 2 and Password Saver, the leaked information is allowed to flow to the sensitive sink, but only if it is encrypted first. IFC cannot yet express this property, but Sect. 3.15 describes how to extend it to catch this sort of vulnerability.

3.7 Malware not related to information flow

The malware in 10 out of the 57 malicious applications is not related to information flow. These applications neither exhibit unjustified permissions nor reveal an unjustified or exploited information flow. These apps implement types of attacks that are out of the scope of IFC. For example, Backup transposes digits in a phone number during backup. This is a functional correctness error, which IFC does not address. In a high-assurance app store, IFC would be used with complementary tools designed to find malware besides exploited information flow.

3.8 Bugdoors

In 8 apps, our tools found a bugdoor (undesired, exploitable functionality) that the Red Team was unaware of. Even though the Red Team had written and/or modified the app before presenting it to us for analysis, they had not noticed these.

GPS 1 passes the device ID as a waypoint to the remote server. This allows the remote server to correlate location to specific devices and to other information collected using the device ID.

Password Saver saves unencrypted passwords in shared preferences, where they are accessible to other applications on the device.

Furthermore, 6 apps exfiltrated sensitive data to the log, which Android does not require a permission to write. It does, however, require a permission in our finer-grained permission system (see Sect. 2.2). Consequently, IFC reported an information flow violation.

Table 3: Results from the annotation burden experiment.

		Time		**Ass-**	**Annotations**		
App	**LOC**	**(min.)**		**ump.**	**src.+sink=total**		**ratio**
CameraTest	92	20	**.22**	1	6 + 5 = 11	**.12**	6%
Shares Pictures[†]	141	10	**.07**	0	12 + 0 = 12	**.09**	4%
BusinessCard	183	10	**.05**	1	9 + 0 = 9	**.05**	3%
Calculator 3	520	40	**.08**	0	7 + 0 = 7	**.01**	1%
Dynalogin	625	300	**.48**	0	66 + 0 = 66	**.11**	6%
TeaTimer	1098	295	**.27**	7	51 + 3 = 54	**.05**	3%
FourTrack	1108	120	**.11**	0	27 + 18 = 45	**.04**	3%
RingyDingy	1322	180	**.14**	2	41 + 26 = 67	**.05**	4%
VoiceNotify	1360	185	**.14**	11	68 + 44 = 112	**.08**	4%
Sky	1441	240	**.17**	5	33 + 35 = 68	**.05**	3%
Pedometer	1547	165	**.11**	0	71 + 58 = 129	**.08**	5%
Total	9437	1565	**.17**	26	391 + 189 = 580	**.06**	4%

Boldfaced numbers (time, annotations) are per line of code. "Assump." is trusted assumptions. Annotation ratio compares the number of annotations written to how many could have been written — the number of uses of types in the app's source code. Throughout, lines of code (generated using David A. Wheeler's "SLOCCount") omit whitespace and comment lines.
[†]Malicious applications

3.9 Control team results

For the third and fifth batches of applications, our sponsor hired a control team to evaluate the applications and search for malware. The control team consisted of 4 PhD students with, on average, 11 years of Java experience and .5 years of Android experience — very similar to our team. The control team used both static and dynamic analysis tools, including Androguard, Eclipse, and emulators.

For the third batch of apps, the control team took an average of 9.5 hours per application to declare it malicious or benign, and their answer was right 60% of the time. Our team, using IFC plus simple tools that search for suspicious code, spent 42 minutes on average per application and correctly classified 80% of the applications.

For the fifth batch of apps, the control team took an average of 2.3 hours per application and correctly classified 63% of the apps. Our team spent 1.25 hours per app on average and correctly classified 75% of the apps.

3.10 Annotation burden

The IFC verification methodology and toolset provide guarantees, but at a cost: the developer must write source code annotations that express the relevant information flows.

In order to estimate this cost, five of the authors of this paper annotated 11 applications. 1 app was a malicious app written by the Red Teams and 10 apps were benign apps written by third-party developers or the Red Teams. Each person was given an unannotated application and a flow policy. The person fully annotated the application even if they found malware, in which case they suppressed a warning and continued the task. The annotator had never seen the application before, so the vast majority of their time was spent reverse-engineering the application — work that would not be necessary for the application vendor.

Table 3 shows the results. On average, the annotators annotated 6 lines of code per minute, which was primarily the effort to understand the code. This compares favorably with industry-standard averages of about 20 lines of delivered code per day [4, 29, 42, 22].

The annotated code contained on average one annotation per 16 lines of code. This compares favorably with the annotation burden for Jif, another information-flow system for Java. In three studies over a 6-year period, Jif required one annotation per 4–9 lines of code [50], one annotation per 3 lines [51], and one annotation per 4 lines [6]. In our case studies, the annotator wrote an annotation in 4% of the places an annotation could have been written; the other locations were defaulted or inferred.

The number of annotations per application is not correlated with the number of lines of code nor the number of possible annotations. Rather, the number of annotations is dependent on how, and how much, information flows through the code. When information flow is contained within procedures, type inference reduces the number of annotations required (Sect. 2.6.1).

3.11 Auditing burden

Another cost in the use of a static tool is the need to examine warnings to determine which ones are false positives. This cost falls on the developer, then again on the auditor. We wished to determine the cost to the app store of approving an app, which requires auditing the flow policy and each trusted assumption.

Two of the authors of this paper acted as app store auditors. They reviewed the applications developed in the Annotation Burden experiment from the previous section. The auditors had never before seen the applications that they reviewed, and they did not know whether they were malware. The review was split into two phases: a review of the app description and policy, then a review of the trusted assumptions and conditionals in the source code. This is exactly the same workflow as an app store auditor. Table 4 summarizes the results.

The first part of the review ensures that the description of the app matches the flow policy. An auditor begins by reading the app description and writing a flow policy; then the auditor compares that to the submitted flow policy. If there is any difference, the developer must modify the

Table 4: Results from the collaborative app store experiment.

App Name	Review time (min.)		Reviewed Assump.	Reviewed Cond.		Accepted
CameraTest	26	**.28**	1	0	0%	Accept
Shares Pictures[†]	5	**.04**	0	0	0%	Reject
BusinessCard	11	**.06**	1	1	14%	Accept
Calculator 3	11	**.02**	0	3	5%	Accept
Dynalogin	10	**.02**	0	10	37%	Accept
TeaTimer	50	**.05**	7	20	22%	Accept
FourTrack	61	**.06**	0	11	14%	Accept
RingyDingy	20	**.02**	2	11	9%	Accept
VoiceNotify	35	**.03**	11	73	47%	Accept
Sky	25	**.02**	5	19	15%	Accept
Pedometer	15	**.01**	0	65	57%	Accept
Total	269	**.03**	27	213	27%	

Boldfaced times are per line of code. All trusted assumptions were reviewed. The Reviewed Conditions column gives the number of reviewed conditions and the percentage of all conditional sinks that needed to be reviewed. [†]Malicious applications

description or flow policy. The policy review took 35% of total auditing time.

The second part of the review ensures that all trusted assumptions and indirect information flows are valid. The auditor first reviewed each suppressed warning its developer-written justification. Only CameraTest had one rejected justification, which the developer rectified in a resubmission. The other justifications were accepted by the auditors. Then, the auditors investigated the information flow into conditional sinks, ensuring that any dependency is benevolent.

After the experiment, auditors mentioned that there were many unexpected flows, which ended up being necessary. Also, they wanted clear guidelines to accept or reject flow policies. We believe that both concerns will be resolved as auditors and app stores get more experience; this was their first time to audit apps.

We have not evaluated the effort of analyzing an update to an existing app, but this should also be low. An update can re-use the explicit flow policy specification, annotations, and justifications for trusted assumptions) of previous versions.

3.12 Learnability

IFC integrates smoothly with Java and re-uses type system concepts familiar to programmers. Nonetheless, learning about information flow, or learning our toolset, may prove a barrier to some programmers. The programmers in the study of Sect. 3.10 were already familiar with Android and IFC. We wished to determine how difficult it is to come up to speed on IFC.

We conducted a study involving 32 third-year undergraduate students enrolled in an introductory compilers class. Most of the students had no previous experience with Android. They received a two-hour presentation, then worked in pairs to annotate an app of 1000–1500 lines. The apps came from the `f-droid.org` catalog; we do not have access to the source code of most apps in the Google Play Store.

The students' task was to learn Android, information flow theory, and the IFC toolset, then to reverse-engineer the app and to annotate it so that IFC issues no warnings. On average the task required 15 hours. The students reported that the first annotations were the most time-consuming because they were still learning to understand IFC; after that the task was easier.

3.13 Lessons learned

This section states a few lessons we learned during our experiments.

Program annotations foil some of the ways to hide malware. IFC hampers data exfiltration. Hiding data-flow based malware in an application that is annotated with flow sources and flow sinks turned out to be difficult for the Red Teams, even though they had access to our source code, documentation, and our own evaluation of our system's limitations.

Generality of our analysis. Our information-flow based approach turned out to be surprisingly general. Our toolset revealed malicious data flow of the payload as well as the injected triggers. We found, for instance, malware in applications that give wrong results based on a certain time of day or a random value. Perhaps more importantly, we were able to easily extend our system as we discovered new properties that we wished IFC to handle — we did so over the course of our own usage and also between batches of malware analysis in the experiments.

3.14 Threats to validity

Our success in the experiments shows promise for our approach. Nonetheless, we wish to highlight a few of the most important threats to validity.

Characteristics of malware. The malware we analyzed was created by five different teams, each consisting of multiple engineers working full-time on the task of creating stealthy malware. The teams had previously surveyed real malware, and they created malware representative both of commercial malware that makes a profit and advanced persistent threats who aim to steal information. Nonetheless, we have no assurance that this malware was representative of malware in the wild, either in terms of types of malware or its quality. It is also possible that our tools became tuned to the sort of malware created by those five Red Teams.

Skill of the analysts. The same instrument may be more or less effective depending on who is using it. It is possible that our team was particularly skilled or lucky in effectively classifying all the apps that it analyzed — or that another team would have done a better job. An analyst needs time to come up to speed on IFC; we have found that a few weeks is sufficient for an undergraduate working part time, as confirmed by experiments (Sect. 3.12). Training only needs to occur once, and our team's unfamiliarity with the apps was a bigger impediment.

Collaborative app verification model. Our model assumes that application vendors are willing to annotate their source code. We believe this is true for high-assurance app stores, but our approach may not be applicable to ordinary app stores.

3.15 Future work

We plan to enrich flow policies in three ways, while retaining the simple and high-level flavor of these specifications. (1) We will refine permissions, such as splitting the WRITE_CONTACTS permission so that separate policies can be specified for email addresses, phone numbers, and notes fields. (2) The flow policy will indicate not just the endpoints of the information flow, but an entire path. For example, it might be valid to send personal information to the Internet only if it has passed through an encryption module first. (3) The flow policy will indicate conditional information flows, such as permitting information flow from the microphone to the network only when the user presses the "transmit" button.

We plan to implement a variant of record types, so that (for example) different parts of a data structure or file can be given different information-flow types. We have already successfully implemented this for Android's intents, improving IFC's analysis of inter-process communication.

4 Related work

This section discusses the research most closely related to our approach.

4.1 Information flow

Information flow tracking has been investigated for several languages and paradigms [14, 37, 25, 19]. These approaches are largely complementary to our work as they are theoretical or do not employ type systems to achieve static guarantees of information flow properties. Besides statically verifying properties, several approaches for enforcing information flow properties have been proposed, such as refactoring [41], dynamic analysis [28], or encoding as safety properties [44, 33]. Milanova and Huang [30] recently presented a system that combines information flow with reference immutability to improve precision. Yet, the system has not been applied in a security context. Engelhardt et al.[11] discuss handling intransitive information-flow policies; IFC requires making transitive flows explicit. Sun et al. [43] discusses modular inference for information flow; IFC provides flow-sensitive type refinement within method bodies.

In the domain of information flow tracking for Java programs, the closest related work is Jif (Java information flow) [32, 31, 39]. Jif uses an incompatible extension of the Java programming language and its own compiler to express and check information flow properties of a program. In contrast, IFC uses standard Java annotations and the standard Java compiler. Furthermore, IFC achieves its effects with a simpler, easier-to-use type system. While Jif focuses on the expressiveness and flexibility of the type system and trust model, IFC aims at practicality and scalability to be applicable on large real-world Android applications. Jif has not been evaluated in an adversarial challenge exercise comparable to our experiments using IFC.

WebSSARI (Web application Security by Static Analysis and Runtime Inspection) [21] is another related approach but targets a different domain. WebSSARI focuses on web applications written in PHP and aims at preventing vulnerabilities such as Cross-Site Scripting or SQL Injection. In this context, static analysis is applied to reveal existing weaknesses and to insert runtime checks. In contrast, IFC statically verifies information flow properties for Android applications.

4.2 Android studies

Many recent research studies have focused on understanding the weaknesses of the Android platform, as well as characterizing Android malware in the wild. This section discusses IFC in the context of those prior studies since it also targets the Android platform.

Recent studies (e.g., [1, 12, 45]) investigated the Android permission system and revealed that many Android applications are overprivileged, meaning that they are granted more permissions than they use. These studies also provided a mapping of API calls to required permissions. IFC utilizes those existing mappings and enhances the Android permission system by adding finer-grained sources and sinks for sensitive APIs.

Chin et al. [5] described a weakness caused by the Android Intent mechanism: implicitly sent intents can be intercepted by malicious applications. IFC analyzes communication through intents to mitigate such attacks.

4.3 Malware detection and prevention

Ongtang et al. [35] suggest an application-centric security model to strengthen Android's security. The Google Play Store runs Bouncer to detect and reject malicious applications. Unfortunately, Bouncer can be circumvented [36, 23], which motivates our work.

Tools for detecting or preventing malicious behavior on smartphones employ static analysis for detection or dynamic analysis for both detection and prevention. Woodpecker [17] uses static analysis to detect capability leaks and ComDroid [5] to locate Intent-related vulnerabilities. In addition, several systems have been proposed to detect the leakage of personal data (e.g., [16, 27]). In this context, PiOS [9] is a system for the detection of privacy leaks in iOS Applications, which constructs a control flow graph from compiled code and performs data flow analysis. Unlike those existing approaches, IFC uses a finer-grained model for sources and sinks, operates on the source code, and is not limited to explicit information flow. RiskRanker [18] and DroidRanger [53] combine multiple analyses in an attempt to detect likely malware.

Besides the detection tools, dynamic enforcement tools have been proposed in the literature that monitor the execution of an application at runtime and intervene, if necessary, to ensure safe behavior. TaintDroid [10] and DroidScope [49] use taint-tracking to monitor the flow of sensitive data that is sent from the phone, whereas AppFence [20] automatically replaces the sensitive data with innocuous data. Both of these techniques require modification to the Android runtime framework, making the approach non-portable. As an alternative Aurasium [48] uses inlined dynamic enforcement, which rewrites the target application to embed runtime checks. Although inlined dynamic enforcement improves portability, the performance and code size overhead may affect its suitability for resource-constrained mobile platforms.

5 Conclusion

We have described IFC, a flow-sensitive, context-sensitive type system that enables collaborative verification of information flow properties in Android applications. Its design focuses on usability and practicality, and it supports a rich programming model.

We evaluated IFC by analyzing 72 new applications (57 of them malicious), which were written by 5 different corporate Red Teams who were not under our control. IFC detected 96% of the information-flow-related malware (we explain how to increase this number to 100%), and 82% of all malware. Other experiments show that IFC is easy to use for both programmers and auditors.

Our system is freely available, including source code, library API annotations, user manual, and example annotated applications.

6 Acknowledgments

We thank Yoshi Kohno, David Wetherall, Dylan McNamee, David Naumann, John Singleton, David Brumley, Zach Tatlock, and Marcelo d'Amorim for helpful discussions.

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. PScout: Analyzing the Android permission specification. In *CCS* (Oct. 2012), pp. 217–228.

[2] Banerjee, A., Naumann, D. A., and Rosenberg, S. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy* (2008), pp. 339–353.

[3] Bonnington, C. First instance of iOS app store malware detected, removed, 2012. `http://www.wired.com/gadgetlab/2012/07/first-ios-malware-found/`.

[4] Brooks, Jr., F. P. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Boston, MA, USA, 1975.

[5] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In *MobiSys* (June 2011), pp. 239–252.

[6] Chong, S., Vikram, K., and Myers, A. C. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX Security* (Aug. 2007).

[7] Denning, D. E. A lattice model of secure information flow. *CACM 19*, 5 (May 1976), 236–243.

[8] Dietl, W., Dietzel, S., Ernst, M. D., Muşlu, K., and Schiller, T. Building and using pluggable type-checkers. In *ICSE* (May 2011), pp. 681–690.

[9] Egele, M., Kruegel, C., Kirdaz, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *NDSS* (Feb. 2011).

[10] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (Oct. 2010).

[11] Engelhardt, K., van der Meyden, R., and Zhang, C. Intransitive noninterference in nondeterministic systems. In *ACM Conference on Computer and Communications Security* (2012), pp. 869–880.

[12] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *CCS* (Oct. 2011), pp. 627–638.

[13] Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A survey of mobile malware in the wild. In *SPSM* (Oct. 2011), pp. 3–14.

[14] Ferrari, E., Samarati, P., Bertino, E., and Jajodia, S. Providing flexibility in information flow control for object-oriented systems. In *IEEE Security and Privacy* (May 1997), pp. 130–140.

[15] Foresman, C. Proof-of-concept app exploiting iOS security flaw gets researcher in trouble with Apple, 2012. `http://arstechnica.com/apple/2011/11/safari-charlie-discovers-security-flaw-in-ios-gets-booted-from-dev-program/`.

[16] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST* (June 2012), pp. 291–307.

[17] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock Android smartphones. In *NDSS* (Feb. 2012).

[18] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. RiskRanker: Scalable and accurate zero-day Android malware detection. In *MobiSys* (June 2012), pp. 281–294.

[19] Hammer, C., Krinke, J., and Snelting, G. Information flow control for java based on path conditions in dependence graphs. In *ISSSE* (Mar. 2006), pp. 87–96.

[20] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS* (Oct. 2011), pp. 639–652.

[21] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW* (May 2004), pp. 40–52.

[22] Jones, C. *The Economics of Software Quality*. Addison-Wesley, 2011.

[23] Kassner, M. Google Play: Android's Bouncer can be pwned. `http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/`, 2012.

[24] Kitching, C., and McVoy, L. BK2CVS problem. `http://lkml.indiana.edu/hypermail/linux/kernel/0311.0/0635.html`, 2003.

[25] Li, P., and Zdancewic, S. Encoding information flow in Haskell. In *CSFW* (July 2006), pp. 16–27.

[26] Liu, L., Zhang, X., Yan, G., and Chen, S. Chrome extensions: Threat analysis and countermeasures. In *NDSS* (Feb. 2012).

[27] Mann, C., and Starostin, A. A framework for static detection of privacy leaks in Android applications. In *SAC* (Mar. 2012), pp. 1457–1462.

[28] Masri, W., Podgurski, A., and Leon, D. Detecting and debugging insecure information flows. In *ISSRE* (Nov. 2004), pp. 198–209.

[29] McConnell, S. *Software Estimation: Demystifying the Black Art*. Microsoft Press, 2006.

[30] Milanova, A., and Huang, W. Composing polymorphic information flow systems with reference immutability. In *FTfJP* (July 2013), pp. 5:1–5:7.

[31] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL* (Jan. 1999), pp. 228–241.

[32] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java + information flow. `http://www.cs.cornell.edu/jif`.

[33] Naumann, D. A. From coupling relations to mated invariants for checking information flow. In *European Symposium on Research in Computer Security (ESORICS)* (2006), vol. 4189 of *LNCS*, pp. 279–296.

[34] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security* (Aug. 2013), pp. 543–558.

[35] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. In *ACSAC* (Dec., 2009), pp. 340–349.

[36] PEROCO, N. J., AND SCHULTE, S. Adventures in BouncerLand. In *Black Hat USA* (July 2012).

[37] POTTIER, F., AND SIMONET, V. Information flow inference for ML. In *POPL* (Jan. 2002), pp. 319–330.

[38] RASHID, F. Android malware makes up this week's dangerous apps list. `https://www.appthority.com/news/android-malware-makes-up-this-weeks-dangerous-apps-list`, 2013.

[39] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *J. Sel. Areas in Commun. 21*, 1 (Sep. 2003), 5–19.

[40] SCHOUWENBERG, R. Malware in the amazon app store. `https://www.securelist.com/en/blog/208194054/Malware_in_the_Amazon_App_Store`, 2012.

[41] SMITH, S., AND THOBER, M. Refactoring programs to secure information flows. In *PLAS* (June 2006), pp. 75–84.

[42] SU, P. Broken Windows theory. `http://blogs.msdn.com/b/philipsu/archive/2006/06/14/631438.aspx`, June 2006.

[43] SUN, Q., BANERJEE, A., AND NAUMANN, D. A. Modular and constraint-based information flow inference for an object-oriented language. In *SAS* (2004), pp. 84–99.

[44] TERAUCHI, T., AND AIKEN, A. Secure information flow as a safety problem. In *SAS* (Sep. 2005), pp. 352–367.

[45] VIDAS, T., CHRISTIN, N., AND CRANOR, L. Curbing Android permission creep. In *W2SP* (May 2011).

[46] VOLPANO, D. M., AND SMITH, G. A type-based approach to program security. In *TAPSOFT '97* (Apr. 1997), pp. 607–621.

[47] WANG, T., LU, K., LU, L., CHUNG, S., AND LEE, W. Jekyll on iOS: When benign apps become evil. In *USENIX Security* (Aug. 2013), pp. 559–572.

[48] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security* (Aug. 2012).

[49] YAN, L. K., AND YIN, H. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security* (Aug. 2012).

[50] ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. C. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP* (Oct. 2001), pp. 1–14.

[51] ZHENG, L., CHONG, S., MYERS, A. C., AND ZDANCEWIC, S. Using replication and partitioning to build secure distributed systems. In *IEEE Security and Privacy* (May 2003), pp. 236–250.

[52] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *IEEE Security and Privacy* (May 2012), pp. 95–109.

[53] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS* (Feb. 2012).

A Appendix

Table 5 lists the malicious applications (Trojans) that were written by 5 independent corporate Red Teams and were analyzed using IFC.

Table 5: Trojan applications analyzed by IFC.

	Description	LOC	Malware Description	Information Flow Violation	IFC
1	Adventure Game	17,896	Overwrites all files on the SD-card and deletes all SMS	READ_SMS	✓
2	Countdown Timer	1,065	Drops all incoming SMSes	RECEIVE_SMS	✓
3	Note Taker	3,251	Sends audio recordings over the Internet	INTERNET	✓
4	Screen Saver 1	147	Corrupts the local file system	WRITE_EXTERNAL_STORAGE	✓
5	SMS Pager	1,834	Sends SMS to a remote web server	INTERNET	✓
6	System Monitoring 3	3,334	Blocks all SMSes	RECEIVE_SMS	✓
7	Battery Indicator	4,214	Reads and sends picture data from the external storage to a web address	INTERNET	✓
8	Block SMS	2,087	Sends all SMS messages to a server by adding them to a hardcoded URL	INTERNET	✓
9	Calculator 2	640	Writes calculations to a file and then sends the file to a server	INTERNET	✓
10	SMS Backup	293	Leaks SMS messages, browser history, and file names to the SD card log	BROWSER_HISTORY	✓
11	SMS Notification	9,678	Writes SMSes to the log	WRITE_LOGS	✓
12	System Monitoring 1	9,402	Sets the global proxy to 10.1.1.1	WRITE_SETTINGS	✓
13	Fortune	2,998	Sends the device ID to server	INTERNET	✓
14	WiFi Finder	852	Sends location data to server	ACCESS_FINE_LOCATION	✓
15	Cookbook	2,542	Deletes contact list	WRITE_CONTACTS	✓
16	Password Protects Apps	11,743	Locks all apps with a random password	MODIFY_PHONE_STATE	✓
17	Phone silencer	1,415	Blocks all outing and incoming calls	RECEIVE_BOOT_COMPLETED	✓
18	Replacement launcher	1,069	Writes SIM ID to the SDCARD	WRITE_EXTERNAL_STORAGE	✓
19	2D Game	33,017	Sends location data from photos on the SD card to a server	READ_EXTERNAL_STORAGE→ INTERNET	✓
20	Displays source code	242	Sends the device id to 127.0.0.0	READ_PHONE_STATE→INTERNET	✓
21	System Monitoring 2	9,530	Writes GPS data to SD Card	ACCESS_FINE_LOCATION→ WRITE_EXTERNAL_STORAGE	✓
22	SMS Encryption	27,764	Sends all outgoing SMSes to the intended recipient and to a number specified in a trigger SMS	READ_SMS→SEND_SMS	✓
23	Bible	19,775	Downloads a jar-file and executes it	INTERNET→ WRITE_EXTERNAL_STORAGE	✓
24	GPS 1	720	Sends device ID to attacker's server	READ_PHONE_STATE→INTERNET	✓
25	GPS Logger	6,907	Sends location to attacker's server	ACCESS_FINE_LOCATION→ INTERNET	✓
26	Shares Pictures	135	Sends location data from photos to attackers server	READ_EXTERNAL_STORAGE→ INTERNET	✓
27	Cat Pictures	639	Reads location from pictures and sends it to a malicious server	READ_EXTERNAL_STORAGE→ INTERNET	✓
28	SMS Messenger	1,210	Sends a spoofed SMS message to a contact	READ_SMS→WRITE_SMS	✓
29	Running Log	1,333	Writes phone number to NFC tag	READ_PHONE_STATE→NFC	✓

✓The malicous flows or permissions in these apps were found using IFC

Continued on next page

Table 5: Trojan applications analyzed by IFC — continued from previous page.

	Description	LOC	Malware Description	Information Flow Violation	IFC
30	Calculator 1	510	Uses a randomized value as left operand	RANDOM→DISPLAY	✓
31	RSS Reader	3,503	Vibrates randomly	RANDOM→VIBRATE	✓
32	Text to Morse code	263	DoS on storage system	USER_INPUT→FILESYSTEM	✓
33	Shares Location	248	Sends user's location to 10.0.1.8 using ProcessBuilder and ping	ACCESS_FINE_LOCATION→ PROCESS_BUILDER	✓
34	Calculator 4	482	Displays a random number instead of result	RANDOM→DISPLAY	✓
35	Device Admin 1	1,474	Leaks location data to a service via an intent	ACCESS_FINE_LOCATION→INTENT	✓
36	Device Admin 2	1,700	A file containing the user's phone number is sent to the Internet	FILESYSTEM→INTERNET	✓
37	DropBox Uploader	5,902	Leaks phone number via screenshot to Internet	DISPLAY→INTERNET	✓
38	GPS 3	1,512	Location data is sent to maps.google-com.cc rather than maps.google.com	LOCATION→INTERNET("maps.google-cc.com")	✓
39	Geocaching	27,892	Sends data from any NFC tag in range to server	NFC("*")→INTERNET	✓
40	Instant Messenger	1,253	Sends all chats to user 0xFFFF	LITERAL("0xFFFF")→INTERNET	✓
41	App Backup	2,010	Deletes SDCARD	LITERAL→ WRITE_EXTERNAL_STORAGE("*")	✓
42	Mapping	5,587	Sends location data to malicious server	LOCATION→ INTERNET("mapxplore.com")	✓
43	SIP VoIP Phone	1,480	Allows third party to listen to phone calls	USER_INPUT → USE_SIP("2233520413@sip2sip.info")	✓
44	Word Game	1,191	Sends contact information to hardcoded phone number	LITERAL →SEND_SMS("12025551212")	✓
45	PGP Encryption 1	9,904	Appends BASE64 encoded passphrase to the version string of the message	USER_INPUT("EditText.passPhrase")→ EMAIL	✓
46	PGP Encryption 2	9,945	Appends unencrypted message to the encrypted text	USER_INPUT("EditText.message")→ EMAIL	✓*
47	Password Saver	508	Saves passwords in plain text in shared preferences	USER_INPUT("EditText.createPassword") →SHARED_PREFERENCES	✓*
48	Podcast Player	1,711	Battery DoS: continually plays a song that has no sound	none	
49	Screen Saver 2	419	Battery DoS: Disables Back and Menu Button, replaces home launcher, and uses 100% of the CPU	none	
50	To Do List	5,123	Battery DoS: High refresh rate and display brightness	none	
51	Sudoku	1,505	Battery DoS: Spawns a thread with an infinite loop	none	
52	Expense reports	2,293	Performance DoS: does not kill threads	none	
53	Automatic SMS replies	33,296	Performance DoS: Infinitely sends SMSes	none	
54	Screen Saver 3	457	Performance DoS: throttles the loop-back interface	none	
55	Backup	2,554	Transposes digits in phone number backup	none	
56	SMS Reminders	2,917	If a SMS with the text '000000000' is received, all reminders are deleted	none	
57	Game 3	1,211	Loads preferences under another screen and passes touch events to preferences screen	none	

✓The malicous flows or permissions in these apps were found using IFC

✓*These malicious flows will be caught by IFC after future work is complete. See Sect. 3.6