Collaborative Verification of Information Flow for a High-Assurance App Store

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl*,
Stuart Pernsteiner, Franziska Roesner, Karl Koscher,
Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu

Computer Science and Engineering
University of Washington, Seattle, WA, USA
{mernst, rjust, smillst, spernste, franzi, supersat, pbfs,
bhora, syhan, plvines, edwardwu}@cs.washington.edu

*Electrical and Computer Engineering
University of Waterloo, Waterloo, ON, Canada
wdietl@uwaterloo.ca

ABSTRACT

Current app stores distribute some malware to unsuspecting users, even though the app approval process may be costly and time-consuming. High-integrity app stores must provide stronger guarantees that their apps are not malicious. We propose a verification model for use in such app stores to guarantee that the apps are free of malicious information flows. In our model, the software vendor and the app store auditor collaborate — each does tasks that are easy for her/him, reducing overall verification cost. The software vendor provides a behavioral specification of information flow (at a finer granularity than used by current app stores) and source code annotated with information-flow type qualifiers. A flow-sensitive, context-sensitive information-flow type system checks the information flow type qualifiers in the source code and proves that only information flows in the specification can occur at run time. The app store auditor uses the vendor-provided source code to manually verify declassifications.

We have implemented the information-flow type system for Android apps written in Java, and we evaluated both its effectiveness at detecting information-flow violations and its usability in practice. In an adversarial Red Team evaluation, we analyzed 72 apps (576,000 LOC) for malware. The 57 Trojans among these had been written specifically to defeat a malware analysis such as ours. 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 addition to the adversarial evaluation, we evaluated the practicality of using the collaborative model. The programmer annotation burden is low: 6 annotations per 100 LOC. Every sound analysis requires a human to review potential false alarms, and in our experiments, this took 30 minutes per 1,000 LOC for an auditor unfamiliar with the app.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660343.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/program verification;
D.4.6 [**Security and Protection**]: Information flow control

General Terms

Verification, Security

Keywords

Information flow, static analysis, Android security, collaborative verification

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, and every major app store has approved Trojans [6, 23, 28, 30, 32, 48, 51, 57].

We are exploring the practicality of a high-assurance app store that gives greater understanding of, and confidence in, its apps' behavior in order to reduce the likelihood that a Trojan is approved and distributed to users. A high-assurance app store would be particularly valuable in certain sensitive settings. For example, 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. This characterization provides insufficient limitations on 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 and understand.

We have developed a new approach to verifying apps that addresses each of these factors. (1) We have created a powerful, flow-sensitive, context-sensitive type system that verifies information flows. The type system is easy to use and works with Java and Android. (2) Our type system proves that apps conform to finer-grained information-flow specifications than current app stores. These specifications indicate not just which resources may be accessed but which information flows are legal — how the resources may be used by the program. (3) Our approach uses source code rather than binaries, because source code provides more information, enables more accurate and powerful analyses, and allows an auditor to evaluate false positive warnings. While not all application developers may wish to provide their source code to an app store, we argue that this requirement is reasonable for app stores in certain settings, e.g., in the context of corporate, military, government, or medical applications. (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. However, 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 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 the latter number to 100%.) As with any program analysis, a human must 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 (application vendors) 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 information-flow type-checker cannot catch all malware, but it raises 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. While today's commercial app stores do not require source code, we discuss in Sect. 1.2 the market forces that enable an app store such as we propose. This 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 [31]. As another example, Liu et al. developed proof-of-concept malware as Chrome extensions [36], which are essentially distributed as source code. The Heartbleed bug appeared in open-source software. We believe that source code analysis for security will become increasingly important, so it is worthy of attention from security researchers.

Our approach is for Java source code, but since the type qualifiers are persisted to the classfile, it would be possible to re-implement our type system for bytecode in order to verify compiled apps.

1.2 Collaborative verification model

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.

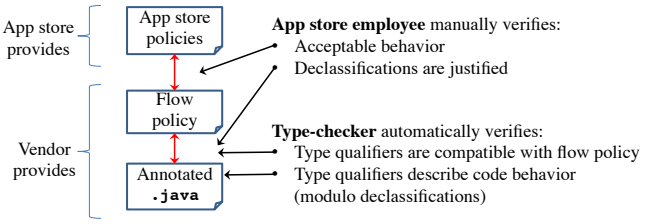

Figure 1: The collaborative verification model for information flow. The flow policy is a high-level specification that expresses application behavior in terms of user-visible information flows.

We propose to augment existing app store approval processes with a collaborative model (Fig. 1) for verification of information flow. The application vendor provides more information to the auditor (an app store employee). This information is easy for the vendor to provide, but it would be difficult for the auditor to infer. The auditor is able to make a decision about information flow 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 is implicitly forbidden. The second vendor-provided artifact is the source code, annotated with information flow type qualifiers. The annotation burden is low: on average 6 annotations per 100 lines of code.

Both the annotations and the vendor are untrusted. Our implementation, Information Flow Type-checker (IFT), automatically ensures that the type qualifiers are both permitted by the flow policy and are an accurate description of the source code's behavior (modulo any auditor-verified declassifications). If not, the app is rejected. Unannotated apps are also rejected. Thus, the application vendor must provide accurate type qualifiers 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 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 declassification, using some other verification methodology (e.g., [5]). Sect. 3.3.2 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. Increased security is especially important in sensitive contexts such as government, corporate, and medical applications. Even if some vendors will never participate in a high-assurance app store, we believe there is value in researchers investigating and improving the practicality of such stores.

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 type qualifiers are written as the code is first developed, they may even save time by preventing errors or directing the author to a better design.

Some vendors may be concerned with confidentiality of their source code. Large organizations already require their vendors to provide and/or escrow source code. For Android apps, it is easy to decompile a Java program from .class or .dex format, so even an app in binary format does not protect the vendor's algorithms, protocols, and other secrets. These facts may reduce vendors' reluctance to provide source code.

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, IFT uses a flow policy as a specification or formal model of behavior. If IFT 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. IFT issues a warning at every declassification, and manual checking is required for each one. IFT 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 and app source code (including type qualifiers) are not trusted.

There have been previous studies of the kinds of malware present in the wild [17, 62]. Felt et al. [17] 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%).

Our approach is intended to be augmented by complementary research and app store activities that focus on other threats. Our approach raises the bar for attackers rather than providing a silver bullet. Sect. 2.9 discusses limitations of our system in greater detail.

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 investigating the information flow of apps submitted to an app store. Our work explores a promising point in the trade-off between human and machine effort.

We have extended information-flow verification to a real, unmodified language (Java) and platform (Android). Our design is easy to use yet 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 (`http://types.cs.washington.edu/sparta/`), and we have experimentally evaluated our work. Our system effectively detected realistic malware targeted against it, built by skilled Red Teams. The effort to use our system was low for both programmers and auditors: our system is powerful, yet it requires less annotation overhead than previous systems and is simpler to use and understand.

2. INFORMATION FLOW TYPE-CHECKER

This section describes our implementation, called Information Flow Type-checker (IFT), and the type system it enforces. IFT guarantees that if a program is well typed, no information flows exist in the program beyond those expressed in the flow policy that expresses the high-level specification. IFT is sound and conservative: if IFT approves a program, then the program has no undesired information flows, but if IFT issues a warning, then the program might or might not actually have undesired information flows at run time. The guarantee is modulo human examination of a small number of declassifications, including ones about implicit information flow through conditionals.

As shown in Fig. 1, a programmer using IFT 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 type 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 type of the recorded data with `@Source(RECORD_AUDIO) @Sink(FILESYSTEM)`. IFT uses type-checking over an information flow type system to verify that the annotated code is consistent with the flow policy.

2.1 Types: 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. The type qualifier `@Sink(INTERNET)` indicates that `loc` might be output to the network.

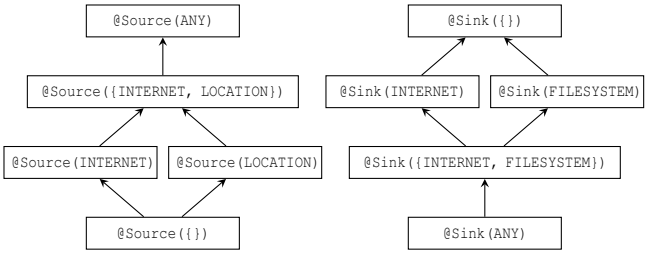

Figure 2: Partial qualifier hierarchy for source and sink type qualifiers `@Source` and `@Sink`.

The arguments to `@Source` and `@Sink` are permissions drawn from our enriched permission system (Sect. 2.2). The argument may be a set of permissions to indicate that a value might combine information from multiple sources or flow to multiple locations. The special constant `ANY` denotes the set of all sources or the set of all sinks; the empty set denotes the absence of sources or sinks.

2.1.1 Subtyping

Adding type qualifiers to the Java type system only requires extending the subsumption rule in a standard way; other Java typing rules remain unchanged. A type qualifier hierarchy indicates which assignments, method calls, and overridings are legal, according to standard object-oriented typing rules. Fig. 2 shows parts of the `@Source` and `@Sink` qualifier hierarchies.

`@Source(`B`)` is a subtype of `@Source(`A`)` iff B is a subset of A [10]. For example, `@Source(INTERNET)` is a subtype of `@Source({INTERNET, LOCATION})`. This rule reflects the fact that the `@Source` qualifier places an upper bound on the set of sensitive sources that were actually used to compute the value. If the type of `x` is qualified by `@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. For example, the type `@Sink({INTERNET, FILESYSTEM})` 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.

Based on these rules, the top type qualifiers of these hierarchies are `@Source(ANY)` and `@Sink({})`, and the bottom type qualifiers are `@Source({})` and `@Sink(ANY)`.

2.1.2 Polymorphism

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

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

to indicate that the elements of `myList` are strings that are obtained from `CONTACTS` and that may flow to `WRITE_SMS`.

IFT also supports qualifier polymorphism, in which the type qualifiers can change independently of the underlying Java type. 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. It also enables qualifier polymorphism even for non-generic Java methods.

For example, the method `@PolySource int f(@PolySource int x)` can be passed an `int` with any sources, and the result has exactly the same sources as the input. This qualifier polymorphism can be viewed as the declaration and two uses of a type qualifier variable. The implicit type qualifier variable is automatically instantiated by IFT at the point of use. Assume a variable `netarg` of type `@Source(INTERNET) int` and a variable `locarg` of type `@Source(LOCATION) int`; in an invocation `f(netarg)` the type qualifier variable is instantiated to `@Source(INTERNET)` and the return type of this method invocation is therefore `@Source(INTERNET) int`; on the other hand, in an invocation `f(locarg)` the type qualifier variable is instantiated to `@Source(LOCATION)` and the return type of this method invocation is therefore `@Source(LOCATION) int`.

Polymorphism allows IFT to be context-sensitive.

Table 1: Additional sources and sinks used by IFT, 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

2.2 Comparison to Android permissions

IFT's permission model differs from the Android permission model in three ways. (1) IFT's permissions are statically guaranteed at compile time, whereas Android permissions are enforced at run time, potentially resulting in an exception during execution. If an app inherits a permission from another app with the same `sharedUserId`, IFT requires that permission to be listed in the flow policy. (2) IFT's permission flows are finer-grained than standard Android manifest permissions. 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. (3) IFT refines Android's permissions, as discussed in this section.

2.2.1 Sinks and sources for additional resources

IFT adds additional sources and sinks to the Android permissions. For example, IFT 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 (18%) to the 145 Android permissions.

Some researchers feel that the Android permission model is already too complicated for users to understand [16], 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 (Sect. 3.3.2 provides empirical measurements). The more detailed flow policy yields more insight than standard Android permissions, because the flow policy 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.

Literal. The LITERAL source is used for programmer-written constants (in the source code, Android manifest, or resource files) such as `"Hello world!"`, and for any variable whose value is computed using only those constants. This enables IFT to distinguish information derived from the program source code from other in-

puts. Program literals are not trusted, since the app vendor may be malicious. The flow policy shows how they are used in the program.

Conditional. The CONDITIONAL sink is used for conditional expressions — every value used in a conditional expression flows to that sink. This enables IFT to raise a warning at locations where the control flow of the program branches on sensitive information. The auditor reviews those warnings to detect implicit information flows, as explained in greater detail in Sect. 2.7.

2.2.2 *Restricting existing permissions*

The standard Android permissions might be too coarse-grained to express the developer's intention. For example, Android's INTERNET permission represents all reachable hosts on the Internet. IFT allows this permission to be parameterized with a domain name, as in INTERNET("*.google.com"). Other permissions can be parameterized in a similar style, and the meaning of the optional parameter varies based on the permission it refines. For example, a parameter to FILESYSTEM represents a file or directory name or wildcard, whereas the parameter to SEND_SMS represents the phone number that receives the SMS. Other permissions that can be parameterized include CONTACTS, *_EXTERNAL_FILESYSTEM, NFC, *_SMS, and USE_SIP. Several of the additional sources and sinks (Table 1) can also be parameterized, such as USER_INPUT to distinguish sensitive from non-sensitive user input.

IFT performs intraprocedural constant value propagation to enable precise analysis of parameterized permissions.

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.

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 flow policy 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 specifies 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 Internet (modulo conditionals and transitive flows). The expression's type might be written by a programmer or might be automatically inferred by IFT.

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

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

Transitive flows. Transitive flows through on-device source-sink pairs must be explicitly written in the flow policy. This is because apps can use on-device sinks to whitewash sensitive information. For example, if a flow policy permits USER_INPUT→FILESYSTEM and FILESYSTEM→INTERNET, then an application might write user input to a file and then send the contents of that file to a malicious server. Therefore, the transitive flow USER_INPUT→INTERNET must be explicitly stated in the flow policy.

Parameterized permissions (Sect. 2.2.2) reduce the number of transitive flows. For example, if user input is only written to files in the notes directory (USER_INPUT→FILESYSTEM("notes/*")) and only files in the cat-photos directory are sent to the Internet (FILESYSTEM("cat-photos/*")→INTERNET), then the transitive flow USER_INPUT→FILESYSTEM is not required.

On-device source-sink pairs that are outside of an app's sandbox may be accessed by other apps. Two apps could use such a source-sink pair to leak information. To prevent this, the flow policies of all apps on a device or in an app store can be checked against each other for inter-app transitive flows. If a transitive flow is found that violates a flow policy of an app, then one or more apps may need to be excluded or rewritten. In practice, app stores will specify standard policies for flows including these source-sink pairs, so that developers can avoid writing conflicting apps.

An off-device sink, such as a website or the recipient of an SMS, might leak data to some sink not allowed by the flow policy. Off-device sinks must be either trusted or verified by other means.

2.4 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 rather than written as type annotations. A programmer need not write type annotations within method bodies, because such types are inferred by IFT. For method signatures and fields, a programmer generally writes either `@Source` or `@Sink`, but not both. We now explain the inference and defaulting features. For experimental measurements, see Sect. 3.3.1.

2.4.1 *Type inference and flow-sensitivity*

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

IFT implements this inference via flow-sensitive type refinement. Each local variable declaration (also casts and resource variables) defaults to the top type qualifiers, `@Source(ANY) @Sink({})`. At every properly-typed assignment statement, the type of the left-hand side 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.

Consider the following simple method:

```
void process(@Source(INTERNET) int netint,
             @Source(LOCATION) int locint) {
  int x;       // x is defaulted to @Source(ANY) @Sink({}) int
  x = netint;  // x is refined to @Source(INTERNET) int
  x = locint;  // x is refined to @Source(LOCATION) int
}
```

Flow-sensitive type refinement spares the programmer from writing type qualifiers on local variable `x`, and the system automatically determines the most precise type in each context.

IFT 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, could cause a change in one part of a program to cause new type-checking errors elsewhere, and would provide fewer documentation benefits.

2.4.2 *Determining sources from sinks and vice versa*

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

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({})`

Suppose the flow policy contains the following:

```
CAMERA -> DISPLAY,DATABASE
LOCATION -> DATABASE
```

Then these pairs are equivalent:

```
@Source({LOCATION}) = @Source({LOCATION}) @Sink(DATABASE)
@Sink(DATABASE) = @Source({CAMERA,LOCATION}) @Sink(DATABASE)
```

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.

An example of a method that uses only a `@Source` qualifier is the `File` constructor: a newly-created readable file should be annotated with `@Source(FILESYSTEM)`, but there is no possible `@Sink` qualifier that would be correct for all programs. Instead, the `@Sink` qualifier is omitted, and our defaulting mechanism provides the correct value based on the application's flow policy.

This defaulting mechanism is essential for annotating libraries. We wrote manual annotations for 10,470 methods of the Android standard library. Only 7 of the API methods annotated so far use both a `@Source` and a `@Sink` qualifier. For example,

```
Camera.setPreviewDisplay(
    @Source(CAMERA) @Sink(DISPLAY) SurfaceHolder holder)
```

The parameter `holder` both receives photos from the camera and displays them.

This mechanism can be viewed as another application of type polymorphism: defaulting of types depends on the flow policy and the same source code can be reused in different scenarios by using a different flow policy.

2.4.3 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 flow policy as explained in Sect. 2.4.2.

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 a simple computation involving only constants does not require annotations.

As is standard, the `null` literal is given the bottom type qualifiers `@Source({}) @Sink(ANY)`, enabling an assignment to any variable.

2.5 Declassifications

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 downcast in a Java program is an example of such conservatism in the Java type system. In the context of information flow analyses, an example would be a database: in general, a database query can return arbitrary sensitive data, but application invariants might guarantee that a particular query always returns non-sensitive data. IFT would warn about use of any database query result in a context that could leak the result, but in the example the warning would be a false positive.

In order to suppress a warning that is a false positive, the developer declassifies data that was typed too conservatively using a downcast. The developer is required by the app store to write a justification for each declassification. The app store auditor manually verifies both the justification and the declassification. Thus, the auditor validates the developer's claim that the code is well-behaved for some reason that is beyond the precision of the type checker.

In 11 Android apps (9437 LOC), IFT suffered 26 false positives, or fewer than 3 per 1,000 LOC (see Sect. 3.3.1).

2.6 Indirect control flow

Indirect control flow, for example in reflection, intents, or exception handling, is challenging for a static analysis. IFT soundly handles these constructs through additional analyses and conservative assumptions.

IFT analyzes Java reflection to determine the target method of a reflective call. This enables a downstream analysis, such as IFT'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`. IFT's 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. The library's conservative annotations for `Method.invoke` ensure that any unresolved reflective call is treated soundly.

Intents are an Android mechanism for interprocess communication, and they can also create processes (Android activities). To handle intents, we extended IFT with map types (similar to record types) that represent the mappings of data in an intent payload. Each app implements intent-receiving methods, and their type signatures act as interface specifications and permit modular checking. As long as new apps are consistent with annotations on previously-checked apps that they may communicate with, the old apps need not be re-checked.

IFT soundly handles other indirect control flows, such as exception handling. For example, types in catch clauses are enforced to be supertypes of any exception they may catch.

2.7 Implicit information flow

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

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

This code leaks the credit card number to the Internet using the flow LITERAL→INTERNET and the fact that `i` is only sent to the Internet when `i == creditCard` evaluates to true.

The classic approach of Denning and Denning [11] to detect implicit information flow is to taint all computations in the dynamic scope of a conditional statement with all the sources from the conditional's predicate. This includes all statements in the body of the conditional and all statements in any method directly or indirectly called by the body. Over-tainting of computations within the dynamic scope of conditionals leads to many false positive alarms. These alarms occur far from the conditional statement or other statement(s) that caused them. In order to determine whether an implicit information flow truly occurs, the auditor has to work

backward from the location of an alarm to the conditional statement or statements that caused it.

In our approach, the auditor reviews every conditional statement that uses a sensitive source in its predicate. The auditor first decides whether the knowledge about the boolean result of the predicate is sensitive information. For example, checking whether a credit card number has 16 digits does not reveal anything sensitive — in this case, the auditor need not review the body of the conditional. However, if the auditor decides that the conditional predicate is sensitive, he/she must rule out any implicit information flow that violates the flow policy. In order to determine whether an implicit information flow truly occurs, the auditor works from the body of the conditional forward to all statements in dynamic scope that might implicitly leak information.

In both the classic approach and our approach, the auditor has to carefully review the dynamic scope of the conditional body to rule out false positives. However, unlike the classic approach, in our approach, the reviewer is aware of the context of the conditional and can make a more informed decision about whether an implicit information flow might occur at runtime.

The auditors in our experiments (Sect. 3.3.2) felt that our approach was easier for them than the classic one. They preferred to think about an entire conditional expression at once rather than statement-by-statement. Oftentimes, examining a conditional expression enabled the auditors to rule out bad behavior without needing to examine any statement in its dynamic scope; this was particularly true for simple conditionals such as tests against `null`.

2.8 Implementation

IFT is implemented as a pluggable type system built on top of the Checker Framework [12] and uses standard Java type annotations. The implementation of IFT consists of 3,731 lines of Java, plus annotations for 10,470 library methods. IFT's source code is available at `http://types.cs.washington.edu/sparta/`. Version 0.9.6 was used for the experiments presented in this paper.

2.9 Limitations

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

As with any static analysis, IFT'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 induced by those components. IFT currently forbids dynamic code loading, because IFT type-checks source code. Dynamic class loading could be soundly allowed if the loaded classes type-check and their public signatures are the same as were assumed at compile time. To achieve this would require load-time type-checking of compiled (.class or .dex) files. Re-implementing the IFT type rules for binaries would be an engineering challenge, but not a conceptual one.

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 IFT or that exploit IFT'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 declassification, in an effort to convince the auditor to approve malware. Our work does not address how to establish an app store's policies.

Despite these limitations, use of IFT increases the difficulty of hiding Trojans in source code.

2.10 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.

3. EMPIRICAL STUDIES

This section describes three different evaluations of IFT. Sect. 3.1 examines the effectiveness of IFT in an adversarial Red Team evaluation. Sect. 3.2 evaluates the effectiveness and efficiency of IFT in a control team study. Sect. 3.3 presents a study of IFT's usability for both vendors during the development of apps and app store auditors while reviewing those apps.

3.1 Red Team evaluation

The sponsor of our research wished to evaluate IFT. 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 IFT, including source code, documentation, and our own analysis of IFT'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 IFT. They had two goals: to evaluate how well IFT might work in practice, and to see how IFT 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. For each batch, we were given a few hours or days to analyze the applications with IFT. The Red Teams were given our results for the first three batches, and they used this information to create malware that was harder for IFT to find.

We received the applications in source code form. IFT 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 Red Teams also had no incentive to provide code documentation or follow a specific design — code comments and design documentation were absent, and the apps contained neither flow policies nor the information flow annotations used by IFT.

3.1.1 Summary of results

Overall, 57 out of 72 apps are malicious and 47 of them contain malware related to information flow. IFT detected 96% of those

apps that contain malicious information flow. Specifically, of the 57 malware apps (Table 5 details all malicious apps):

- 47 exhibit an information flow that is at odds with the application's description:
  - 19 use an information flow between Android permissions; see Sect. 3.1.2.
  - 17 use an information flow involving our new sources or sinks; see Sect. 3.1.3.
  - 11 use an information flow involving parameterized sources or sinks; see Sect. 3.1.4.
- 10 are not detected by IFT because the malware is not related to information flow; see Sect. 3.1.5.

3.1.2 Unjustified information flows

For 19 apps, the Android permissions in the manifest can be justified based on the purpose of the app; however, the apps leak information from one Android permission to another. For example, the app 2D Game has a malicious flow, READ_EXTERNAL_STORAGE →INTERNET. The app accesses 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 a user expect a simple game to do so. Therefore, READ_EXTERNAL_STORAGE→ INTERNET is a malicious flow.

An unjustified Android permission would be grounds for rejection from a high-assurance app store; however, some permissions can be easily justified. For example, one of the Red Teams used an automatic update functionality as a reason to justify the INTERNET permission. In our experiments, we did not reject any app based on requested permissions since none of them were at odds with the app's purpose or description.

3.1.3 Information flows using new sources/sinks

For 17 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. RANDOM is not an Android permission and the description of the app gives no reason to use a random number. 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.

The CONDITIONAL sink detected triggers for malicious behavior. 2 malicious apps, Countdown Timer and System Monitoring 3, triggered non-information flow related malware after receiving SMSes with certain characters. Other apps used time of day, random numbers, or location to trigger information-flow malware. Analysts found these triggers while reviewing the conditional statements.

3.1.4 Flows using parameterized permissions

For 11 apps, the malicious information flow is apparent only via use of parameterized permissions (Sect. 2.2.2). 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 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. IFT cannot yet express this property, but Sect. 2.10 describes how to extend IFT to catch this sort of vulnerability.

3.1.5 Malware not related to information flow

The malware in 10 out of the 57 malicious applications is not related to information flow — these apps do not exhibit an unjustified information flow and implement types of attacks that are out of the scope of IFT. For example, Backup transposes digits in a phone number during backup. This is a functional correctness error, which IFT does not address. In a high-assurance app store, IFT would be used with complementary tools designed to find malware not related to information flow. The auditor's confidence that an app has no information-flow malware would let the auditor spend more time looking for other malware, such as denial of service.

3.1.6 Bugdoors

In 8 apps, IFT 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 way-point ID 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 model (see Sect. 2.2). Consequently, IFT reported an information flow violation.

3.2 Control team study

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 IFT 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.

3.3 Usability study

The collaborative verification model and IFT provide guarantees, but can they be used in the real world? This section presents three studies addressing this question. Sect. 3.3.1 measures the time to add information-flow type annotations to apps. Sect. 3.3.2 measures how quickly and accurately the app store auditors can approve or reject apps. Sect. 3.3.3 evaluates how hard it is for information-flow type system novices to learn to use IFT.

3.3.1 Annotation burden

In order to estimate the cost of adding information flow annotations, five members of our team annotated 11 arbitrarily chosen 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 annotator was given an unannotated application and a flow policy file. The annotators annotated the application until IFT issued no more warnings; if they found malware, they used a declassification and continued the task. The annotators had never seen the applications before, so the vast majority of their time was spent reverse-engineering the application.

Table 3: Results from the annotation burden experiment.

App Name	LOC	Time (min.)		De-class.	Annotations 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. "Declass." is declassifications. 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.

Table 4: Results from the collaborative app store experiment.

App Name	Review time (min.)		Reviewed Declass.	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 declassifications were reviewed. The Reviewed Cond. column gives the number of conditions with a sensitive source, all of which were reviewed, and the percentage of all conditionals with a sensitive source. †Malicious applications.

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 [7, 39, 53, 29]. (On average, the annotators annotated 20 lines of code in 3.3 minutes.) Recall that in the proposed collaborative verification model, the app's developer would annotate the code, which would be faster.

The annotated code contained on average 6 annotations per 100 lines of code. This is less than 1/4 of the annotation burden for Jif, another information-flow system for Java [2, 9, 61]. 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. 2.4.1).

3.3.2 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 who writes declassifications to suppress false positives, then again on the auditor who must review the declassifications. We wished to determine the cost of approving an app, which in addition to reviewing declassifications requires auditing the flow policy and reviewing implicit information flow.

Two graduate students acted as app store auditors. Neither one had previously used IFT or a similar framework. The auditors had never before seen the applications that they reviewed, and they did not know whether the apps were malware. The review was split into two phases: a review of the app description and flow policy, then a review of the declassifications 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 description or flow policy. The flow policy review took 35% of total auditing time.

The second part of the review ensures that all declassifications and implicit information flows are valid. The auditor first reviewed the developer-written justification for each declassification. Only CameraTest had one rejected justification, which the developer rectified in a re-submission. The other justifications were accepted by the auditors. Then, the auditors investigated possible implicit information flow via conditionals (Sect. 2.7). Out of a total of 789 conditional statements, only 27% contained data from a sensitive source, so the auditors only reviewed those to rule out implicit information flows. For some of these conditionals, the auditor did not need to review the conditional body, because the conditional expression did not reveal anything about the content of the source. For example, 41 of the 271 conditionals with sensitive data (15%) were comparisons against null.

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 most or all of the previous flow policy specification, annotations, and justifications for declassifications.

3.3.3 Learnability

IFT integrates smoothly with Java and re-uses type system concepts familiar to programmers. Nonetheless, learning about information flow, or learning how to use IFT, may prove a barrier to some programmers. The programmers in the study of Sect. 3.3.1 were already familiar with Android and IFT. We wished to determine how difficult it is to come up to speed on IFT.

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 LOC. 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 IFT, then to reverse-engineer and to annotate the app such that IFT 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 IFT; after that the task was easier.

This learnability study was extremely preliminary, but it does suggest that a developer with little experience can quickly come up to speed on IFT.

3.4 Lessons learned

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

Generality of our analysis. Our information-flow based approach turned out to be surprisingly general. IFT 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 IFT to handle — we did so over the course of our own usage and also between batches of malware analysis in the experiments.

In response to Red Team apps, we added new permissions (like RANDOM and READ_TIME), inference, intents, reflection, parameterized permissions, and more.

Effectiveness of CONDITIONAL. Initially, the Red Teams used location data, time of the day, or random numbers to trigger malware, but stopped, because IFT warnings made it quite easy to detect those triggers. None of the Red Teams' apps used implicit information flow maliciously — we do not know if this was because it was too hard for them or if they did not consider this attack vector.

3.5 Threats to validity

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

Characteristics of malware. The malware we analyzed was created by five different Red Teams, each consisting of multiple engineers working full-time on the task of creating malware. The teams had previously surveyed real malware, and they created malware representative both of commercial malware that makes a profit and of 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 IFT 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 IFT; we have found that a few weeks is sufficient for an undergraduate working part time, as confirmed by experiments (Sect. 3.3.3). 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.

4. RELATED WORK

4.1 Information flow

Information flow tracking has been investigated for several languages and paradigms [18, 47, 35, 25, 58]. 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 [52], dynamic analysis [38], or encoding as safety properties [55, 43]. Milanova and Huang [40] 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. [15] discuss handling intransitive information-flow policies; IFT makes transitive flows explicit. Sun et al. [54] discusses modular inference for information flow; IFT 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) [42, 41, 50]. 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, IFT uses standard Java annotations and the code can be compiled with the standard Java compiler. Furthermore, IFT 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, IFT aims at practicality and scalability to be applicable on large real-world Android applications. IFT has better support for defaults, inference, reflection, intents, libraries, separate compilation, and many other language features. Jif has not been evaluated in an adversarial challenge exercise comparable to our experiments using IFT.

WebSSARI [27] 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 run-time checks. In contrast, IFT statically verifies information flow properties for Android applications.

4.2 Work on Android malware

Many recent studies have focused on understanding the weaknesses of the Android platform and characterizing Android malware in the wild. For example many Android apps are overprivileged, i.e., they are granted more permissions than they use [4, 16, 56]. These studies also provided a mapping of API calls to required permissions. IFT utilizes those existing mappings and enhances the Android permission system by adding finer-grained sources and sinks for sensitive APIs. Chin et al. [8] described a weakness of Android Intents: implicitly sent intents can be intercepted by malicious applications. IFT analyzes communication through intents to detect such attacks.

Ongtang et al. [44] 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 [46, 30], 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 [21] uses static analysis to detect capability leaks and ComDroid [8] to locate Intent-related vulnerabilities. In addition, several systems have been proposed to detect the leakage of personal data (e.g., [19, 37]). In this context, PiOS [13] detects privacy leaks in iOS applications by constructing a control flow graph from compiled code and performing data flow analysis. FlowDroid [3] is a static taint analysis tool for Android apps that that has not been used to find malware. FlowDroid propagates sources and sinks found using SuSi [49], which uses machine learning to classify and categorize Android library methods as source and sinks. Unlike those existing approaches, IFT uses a finer-grained model for sources and sinks, operates on the source code, and is not limited to explicit information flow. RiskRanker [22] and DroidRanger [63] combine multiple analyses in an attempt to detect likely malware.

Beyond detection, dynamic enforcement tools have been proposed to monitor the execution of an application at run time and intervene, if necessary, to ensure safe behavior. TaintDroid [14] and DroidScope [60] use taint-tracking to monitor the flow of sensitive data that is sent from the phone, whereas AppFence [26] automati-

cally 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 [59] uses inlined dynamic enforcement, which rewrites the target application to embed run-time checks. Although inlined dynamic enforcement improves portability, the performance and code size overhead may affect its suitability for resource-constrained mobile platforms.

Another disadvantage of a dynamic analysis is that it may cause an app to fail at run time. By contrast, a static analysis such as IFT gives a guarantee ahead of time, with no run-time overhead, no special runtime environment, and no risk of failures in the field.

4.3 Collaborative model

A similar collaborative verification model has been proposed in prior work on the verification of browser extensions. For example, Guha et al. [24] describe a model in which browser extension developers specify a policy; as in our approach, the program's adherence to the policy is statically verified, and the reasonableness of the policy is manually verified by an auditor. IFT applies the collaborate verification approach to Android, with a significantly simpler policy language, easing the auditor's burden of verifying the reasonableness of the policy.

Similarly, Lerner et al. [33, 34] extend JavaScript with a type system to statically verify that extensions do not violate the browser's private browsing mode; their approach requires developers to write annotations only where code might violate private browsing expectations. It also requires a skilled auditor to manually verify declassifications. IFT poses a lower annotation and audit burden, and supports a broader range of information flow guarantees.

Our collaborative verification model requires a trained auditor to ensure an app's description matches the app's flow policy. Other work has used crowd sourcing [1], natural language processing [45], or clustering [20] to verify that an apps's description matches an app's functionality. The functionality is modeled by private information accessed, permissions requested, or APIs used. These techniques could be modified to compare a flow policy to an app's description thereby reducing the auditors effort.

5. CONCLUSIONS

We have described a collaborate verification model for high assurance app stores, in which app developers provide annotated source code whose information flow properties are verified by the app store's auditors. In this model, the application developer and the auditor each do tasks that are easy for them, reducing the overall cost.

We designed IFT, 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 IFT by analyzing 72 new applications (57 of them malicious), which were written by 5 different corporate Red Teams who were not under our control. IFT detected 96% of the information-flow-related malware (Sect. 2.10 describes an extension to IFT that would increase this to 100%), and 82% of all malware. Other experiments show that IFT is easy to use for both programmers and auditors, making a collaborative verification model practical for a high-assurance app store.

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.

7. REFERENCES

[1] Y. Agarwal and M. Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *MobiSys*, pages 97–110, 2013.

[2] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *IEEE S&P*, 2012.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269, 2014.

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

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE S&P*, 2008.

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

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

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

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

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

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–513, 1977.

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

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

[14] 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 *OSDI*, 2010.

[15] K. Engelhardt, R. van der Meyden, and C. Zhang. Intransitive noninterference in nondeterministic systems. In *CCS*, 2012.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, pages 627–638, 2011.

[17] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *SPSM*, 2011.

[18] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *IEEE S&P*, pages 130–140, 1997.

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

[20] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, pages 1025–1035, 2014.

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

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

[23] A. Greenberg. iphone security bug lets innocent-looking apps go bad. http://www.forbes.com/sites/andygreenberg/2011/11/07/iphone-security-bug-lets-innocent-looking-apps-go-bad/, 2011.

[24] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE S&P*, 2011.

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

[26] 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 *CCS*, pages 639–652, 2011.

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

[28] M. Isaac. Android malware found in angry birds add-on apps. http://www.wired.com/2011/06/android-malware-angry-birds/, 2011.

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

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

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

[32] D. Kravets. Android market apps hit with malware. http://www.wired.com/2011/03/android-malware-2/, 2011.

[33] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, 2013.

[34] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. Technical Report CS-13-02, Brown University, 2013.

[35] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW*, pages 16–27, 2006.

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

[37] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *SAC*, pages 1457–1462, 2012.

[38] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *ISSRE*, pages 198–209, 2004.

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

[40] A. Milanova and W. Huang. Composing polymorphic information flow systems with reference immutability. In *FTfJP*, pages 5:1–5:7, 2013.

[41] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

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

[43] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS*, 2006.

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

[45] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security*, pages 527–542, 2013.

[46] N. J. Peroco and S. Schulte. Adventures in BouncerLand. In *Black Hat USA*, 2012.

[47] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL*, pages 319–330, 2002.

[48] F. Rashid. 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.

[49] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[50] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *J. Sel. Areas in Commun.*, 21(1):5–19, 2003.

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

[52] S. Smith and M. Thober. Refactoring programs to secure information flows. In *PLAS*, pages 75–84, 2006.

[53] P. Su. Broken Windows theory. http://blogs.msdn.com/b/philipsu/archive/2006/06/14/631438.aspx, 2006.

[54] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *SAS*, 2004.

[55] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[56] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *W2SP*, 2011.

[57] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When benign apps become evil. In *USENIX Security*, pages 559–572, 2013.

[58] X. Xiao, N. Tillmann, M. Fähndrich, J. De Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In *ASE*, pages 80–89, 2012.

[59] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security*, 2012.

[60] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security*, 2012.

[61] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE S&P*, pages 236–250, 2003.

[62] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE S&P*, 2012.

[63] 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 *NDSS*, 2012.

APPENDIX

A. SUMMARY OF MALICIOUS APPS

Table 5: Applications analyzed by IFT. All listed applications are malicious and were written by 5 independent corporate Red Teams.

	Description	LOC	Information Flow Violation	IFT
			Information flow violations involving only Android permissions	
1	Adventure Game	17,896	READ_EXTERNAL_STORAGE→WRITE_EXTERNAL_STORAGE	✓
2	Note Taker	3,251	CAPTURE_AUDIO_OUTPUT→INTERNET	✓
3	SMS Pager	1,834	READ_SMS→INTERNET	✓
4	Battery Indicator	4,214	READ_EXTERNAL_STORAGE→INTERNET	✓
5	Block SMS	2,087	RECEIVE_SMS→INTERNET	✓
6	Fortune	2,998	READ_PHONE_STATE→INTERNET	✓
7	WiFi Finder	852	ACCESS_FINE_LOCATION→INTERNET	✓
8	Replacement launcher	1,069	READ_PHONE_STATE→WRITE_EXTERNAL_STORAGE	✓
9	2D Game	33,017	READ_EXTERNAL_STORAGE→INTERNET	✓
10	Displays source code	242	READ_PHONE_STATE→INTERNET	✓
11	System Monitoring 2	9,530	ACCESS_FINE_LOCATION→WRITE_EXTERNAL_STORAGE	✓
12	SMS Encryption	27,764	READ_SMS→SEND_SMS	✓
13	Bible	19,775	INTERNET→WRITE_EXTERNAL_STORAGE	✓
14	GPS 1	720	READ_PHONE_STATE→INTERNET	✓
15	GPS Logger	6,907	ACCESS_FINE_LOCATION→INTERNET	✓
16	Shares Pictures	135	READ_EXTERNAL_STORAGE→INTERNET	✓
17	Cat Pictures	639	READ_EXTERNAL_STORAGE→INTERNET	✓
18	SMS Messenger	1,210	READ_SMS→WRITE_SMS	✓
19	Running Log	1,333	READ_PHONE_STATE→NFC	✓
			Information flow violations involving IFT's additional permissions	
20	Countdown Timer	1,065	RECEIVE_SMS→CONDITIONAL	✓
21	Cookbook	2,542	LITERAL→WRITE_CONTACTS	✓
22	SMS Notification	9,678	READ_SMS→WRITE_LOGS	✓
23	Calculator 2	640	USER_INPUT→FILESYSTEM	✓
24	SMS Backup	293	READ_EXTERNAL_STORAGE→WRITE_LOG	✓
25	Password Protects Apps	11,743	RANDOM→MODIFY_PHONE_STATE	✓
26	System Monitoring 1	9,402	LITERAL→WRITE_SETTINGS	✓
27	Calculator 1	510	RANDOM→DISPLAY	✓
28	RSS Reader	3,503	RANDOM→VIBRATE	✓
29	Text to Morse code	263	USER_INPUT→FILESYSTEM	✓
30	Shares Location	248	ACCESS_FINE_LOCATION→PROCESS_BUILDER	✓
31	Calculator 4	482	RANDOM→DISPLAY	✓
32	Device Admin 1	1,474	ACCESS_FINE_LOCATION→INTENT	✓
33	Device Admin 2	1,700	FILESYSTEM→INTERNET	✓
34	DropBox Uploader	5,902	DISPLAY→INTERNET	✓
35	System Monitoring 3	3,334	RECEIVE_SMS→CONDITIONAL	✓
36	Phone silencer	1,415	LITERAL→MODIFY_PHONE_STATE	✓
			Information flow violations involving parameterized permissions	
37	Screen Saver 1	147	LITERAL("")→WRITE_EXTERNAL_STORAGE	✓
38	GPS 3	1,512	LOCATION→INTERNET("maps.google-cc.com")	✓
39	Geocaching	27,892	NFC("*")→INTERNET	✓
40	Instant Messenger	1,253	LITERAL("0xFFFF")→INTERNET	✓
41	App Backup	2,010	LITERAL→WRITE_EXTERNAL_STORAGE("*")	✓
42	Mapping	5,587	LOCATION→INTERNET("mapxplore.com")	✓
43	SIP VoIP Phone	1,480	USER_INPUT →USE_SIP("2233520413@sip2sip.info")	✓
44	Word Game	1,191	LITERAL →SEND_SMS("12025551212")	✓
45	PGP Encryption 1	9,904	USER_INPUT("EditText.passPhrase")→EMAIL	✓
46	PGP Encryption 2	9,945	USER_INPUT("EditText.message")→EMAIL	*
47	Password Saver	508	USER_INPUT("EditText.createPassword")→SHARED_PREFERENCES	*
			Malware not related to information flow	
48	Podcast Player	1,711	none — Battery DoS	
49	Screen Saver 2	419	none — Battery DoS	
50	To Do List	5,123	none — Battery DoS	
51	Sudoku	1,505	none — Battery DoS	
52	Expense reports	2,293	none — Performance DoS	
53	Automatic SMS replies	33,296	none — Performance DoS	
54	Screen Saver 3	457	none — Performance DoS	
55	Backup	2,554	none — Data corruption	
56	SMS Reminders	2,917	none — Data corruption	
57	Game 3	1,211	none — Clickjacking	

✓The malicious flows or permissions in these apps were found using IFT.

∗These malicious flows will be caught by IFT after future work is complete. See Sect. 2.10.