SIF: A Selective Instrumentation Framework for Mobile Applications[*]

Shuai Hao, Ding Li, William G.J. Halfond, Ramesh Govindan
Computer Science Department, University of Southern California
{shuaihao, dingli, halfond, ramesh}@usc.edu

ABSTRACT

Mobile app ecosystems have experienced tremendous growth in the last five years. As researchers and developers turn their attention to understanding the ecosystem and its different apps, instrumentation of mobile apps is a much needed emerging capability. In this paper, we explore a selective instrumentation capability that allows users to express instrumentation specifications at a high level of abstraction; these specifications are then used to automatically insert instrumentation into binaries. The challenge in our work is to develop expressive abstractions for instrumentation that can also be implemented efficiently. Designed using requirements derived from recent research that has used instrumented apps, our selective instrumentation framework, SIF, contains abstractions that allow users to compactly express precisely which parts of the app need to be instrumented. It also contains a novel path inspection capability, and provides users feedback on the approximate overhead of the instrumentation specification. Using experiments on our SIF implementation for Android, we show that SIF can be used to compactly (in 20-30 lines of code in most cases) specify instrumentation tasks previously reported in the literature. SIF's overhead is under 2% in most cases, and its instrumentation overhead feedback is within 15% in many cases. As such, we expect that SIF can accelerate studies of the mobile app ecosystem.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Design, Experimentation, Languages, Performance

[*]The first author, Shuai Hao, was supported by Annenberg Graduate Fellowship. This material is based upon work supported by the National Science Foundation under Grant No. CNS-905596 and a Zumberge Research Award from the University of Southern California. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

Keywords

App Instrumentation; Programming Framework; Smartphone; Separation of Concerns

1. INTRODUCTION

Mobile app ecosystems, such as the iPhone App Store and Google Play, have experienced tremendous growth in the last five years. Relative to ecosystems for desktop applications, mobile device app ecosystems are fast growing and have a large number of users, an evolving base of smartphone and tablet platforms, a large number of contributors and developers, as well as a wide range of functionality made possible by ubiquitous Internet access and the availability of various kinds of sensors (GPS, cameras *etc.*).

These factors, together with rapid growth in the use of mobile devices, have sparked an interest in understanding the properties of mobile apps. Recent research has developed methods to study performance properties [25, 29] and security properties [15, 30, 22, 26] of mobile apps. A common thread through this line of research is *instrumentation*: each work has developed customized ways to insert instrumentation for studying app behavior. More generally, app instrumentation is a crucial emerging capability that will facilitate future studies of the mobile app ecosystem.

Traditionally, instrumentation frameworks for programming languages have permitted some degree of flexibility in instrumenting software (Section 5). However, these are insufficient for mobile apps which rely on concurrency, event handling, access to sensors, and (on some mobile platforms) resource usage permissions integrated with the app. These differences, together with the constraints of mobile devices, motivate the need for an instrumentation framework with qualitatively different requirements from that considered in prior work.

A careful analysis of prior research that has used custom instrumentation reveals several interesting requirements of an instrumentation framework for mobile apps (Section 2). We find that the framework must permit *selective* instrumentation since the processing constraints on mobile devices preclude pervasive instrumentation. Furthermore, this capability must permit arbitrary user-level instrumentation that can alter the functionality of the app and not just measure performance. Moreover, the instrumentation framework must permit path inspection between specified codepoints, a capability motivated by device access control capabilities in some mobile OSs. Finally, because user-level instrumentation can add significant overhead, the framework must be able to accurately estimate the overhead of the specified instrumentation.

The paper describes the design and implementation of a Selective Instrumentation Framework (SIF) for mobile apps that satisfies these novel requirements (Section 3). Our first contribution is identifying the smallest set of instrumentation primitives that permit expressivity while admitting efficient implementation. SIF allows users to specify instrumentation locations using *codepoint sets* (collections of locations in the code) that can be selected at various levels of granularity from class hierarchy specifications down to individual bytecodes, and then specify user-defined instrumentation for each set. It also defines a *path set* abstraction, which allows users to dynamically trace inter-procedural paths between two arbitrary codepoints in the app. This capability is novel in an instrumentation framework, and can be used to explore privacy leakage and permissions violations in mobile apps. Taken together, these two abstractions can be used to express all instrumentation tasks considered in the literature. A second contribution is SIF's use of static and dynamic program analysis to derive instrumentation locations, minimize instrumentation overhead, and estimate instrumentation cost. In particular, implementing the path set abstraction requires sophisticated stitching of intra-procedural path segments to derive inter-procedural paths.

SIF's abstractions and implementation methods are, for the most part, independent of the underlying mobile app ecosystem, but we have implemented SIF for the Android platform. Using this implementation, we have evaluated SIF's expressivity and efficiency (Section 4). We demonstrate that SIF's abstractions can express many of the instrumentation tasks previously proposed in the literature, as well as other common tasks. Moreover, the SIF specifications are compact, requiring fewer than 100 lines of code even for the most complicated instrumentation tasks. Finally, SIF can often reduce instrumentation cost significantly, requires less than half a minute to instrument binaries, and provides accurate (within 15%) overhead feedback in many cases.

While much work remains (Section 6), we believe that SIF can accelerate studies of the mobile app ecosystem and lead to an improved understanding of app behavior and usage.

2. BACKGROUND AND MOTIVATION

In this section, we motivate the need for an instrumentation framework for mobile apps, and articulate the unique requirements posed by these apps. We then describe the challenges associated with satisfying these requirements; this discussion lays the groundwork for the design of SIF, described Section 3.

Instrumentation Frameworks. *Instrumentation* refers to the process of inserting code into an application, often by an entity (software or user) other than the original developer. An *instrumentation framework* is a software system that allows an entity to insert instrumentation at specific points in a program. In traditional software systems, instrumentation frameworks are widely used for a variety of tasks [18, 16], but, as we discuss in Section 5, these do not satisfy one or more of the requirements of mobile app instrumentation that we identify below.

Instrumentation frameworks are generally based on one of three different mechanisms. The first mechanism is to instrument the source code, an approach which requires source to be available to the instrumentor. A more general mechanism instruments the runtime system responsible for program execution; for example, an instrumented operating system or virtual machine can record every executed method. The drawback to this is that a customized runtime system must be developed for every platform on which an entity will want to perform instrumentation. Furthermore, once developed, it can be very difficult to modify the runtime system instrumentation.

The third mechanism, and the one we choose, is binary instrumentation, in which instrumentation code is directly inserted into the compiled binary or bytecodes. This does not require source code and is more portable and flexible than customized runtime systems. More broadly, the use of binary instrumentation also enables users to instrument and analyze apps after they have been released. This is an especially important capability in the mobile app ecosystem because its growth has spurred a number of independent efforts in understanding the performance and behavior of mobile apps. For example, AppInsight [25] developed a way to instrument apps for a specific purpose, namely, critical path monitoring. The code for doing this instrumentation was done manually, and was targeted for the purpose of critical path monitoring. In our work, we seek to provide a programming framework that can specify, at a high-level, the instrumentation required for AppInsight and other tasks, leaving the task of generating the low-level instrumentation to a compiler.

Framework Requirements. We conducted a survey of recent research work that has developed customized instrumentation, and have used these to develop a set of requirements for a binary instrumentation framework. We discuss three specific examples; Section 4 presents a more comprehensive discussion of these pieces of research.

- Many mobile apps, in response to a user action, perform multiple concurrent operations, and the user-perceived latency is dominated by the *critical path* (the concurrent operation which takes the most time) through the code. AppInsight [25] has attempted to develop general methods to instrument apps for critical path analysis.

- Some existing mobile operating systems provide coarse-grain access control to sensors and other system facilities: *e.g.,* Android requires app developers to explicitly require permission to access the network or GPS. Researchers [22] have developed methods to instrument apps to enable more *fine-grained permissions* checking: *e.g.,* preventing third-party libraries, often used to develop applications, from using the permitted resources.

- We have been developing a *sensor auditing* capability to instrument an app to understand what processing it performs on the sensor (*e.g.,* GPS or camera), and whether, after acquiring location sensor readings, the app uploads the sensor readings to a website.

Many of these studies are motivated by novel features of mobile app platforms: concurrent execution and event handling, per-app restrictions on resource usage, and the availability of novel sensors on mobile devices.

These studies drive the requirements for our binary instrumentation framework. A strawman approach to solving these problems is to instrument every method call or execution path. However, this can incur significant overhead on modern smartphones, to the point where app usability can be impacted. In some preliminary experiments, we have observed up to 2.5x greater CPU usage when using Android's Traceview [5] to instrument every method and system API invocation. Accordingly, the first requirement of a binary instrumentation framework for mobile apps is *selectivity*: users should be able to instrument only the code of interest to their study.

To support AppInsight and the fine-grained permissions study, our framework needs to provide selectivity by allowing users to *flexibly specify locations* for inserting instrumentation. Specifically, in these studies, the authors inserted instrumentation at specific points in the program: event handlers, API calls with certain permission capabilities, etc.

Many instrumentation frameworks permit selectivity of method calls or APIs. Our sensor auditing study motivates another requirement for a binary instrumentation framework that prior work lacks (Section 5), the ability to *inspect dynamic execution paths*. This capability would allow a user to determine which code paths were traversed between two points in the code, and examine what transformations might have been done on data along these paths.

Instrumentation frameworks differ in the kinds of instrumentation they allow a user to insert. To support AppInsight, a binary instrumentation framework that provides basic instrumentation primitives (such as timing or counting procedure invocations) would suffice. However, the fine-grained permissions study alters the functionality of an app. To support instrumentation for functional modifications, a binary instrumentation framework must allow arbitrary *user-specified instrumentation*, since it cannot anticipate the kinds of instrumentation that might be needed.

Finally, *efficiency* is an important requirement of instrumentation frameworks; the instrumentation overhead must be minimal and must preserve the usability of the mobile app. This is particularly difficult to achieve in a framework which permits user-specified instrumentation, since the framework has no control over the complexity of that instrumentation. Accordingly, we add one additional requirement for binary app instrumentation, *overhead feedback*. If the instrumentation framework can estimate the overhead of user-specified instrumentation, users can quickly adapt the instrumentation (*e.g.,* by being more selective) in order to reduce the overhead, without actually needing to run the instrumented binary.

These requirements raise significant research questions and challenges. What are the appropriate abstractions for specifying *where* instrumentation should be inserted? This is particularly challenging for path inspection since some apps are highly complex and contain several million distinct paths (a static analysis of the code paths involved in composing email using the Gmail app reveals nearly 0.15 million path segments). Additionally, how do we provide a flexible mechanism for allowing the user to provide *any instrumentation* without introducing extra overhead and complications? Finally, how do we minimize and report guidance on *overhead* in a way that can help users? In particular, how do we accurately predict the overheard of arbitrary instrumentation?

Our SIF instrumentation framework provides functionality to meet all of these challenges. SIF provides a domain-specific programming language and support libraries that allow users to selectively instrument an app with arbitrary user-specified code along any path or codepoint based location. The framework uses sophisticated program analysis techniques to introduce minimal overhead during instrumentation and provide overhead feedback for the user-specified instrumentation. We describe how SIF provides all of this functionality in the next section.

3. A SELECTIVE INSTRUMENTATION FRAMEWORK

In this section, we describe SIF, our binary instrumentation framework for mobile apps that satisfies the requirements listed in the previous section. We begin with an overview that describes how a user interacts with SIF and the instrumentation workflow within SIF. We then discuss the instrumentation specification language abstractions, and describe how we overcome some of the challenges listed in Section 2. We conclude the section by describing our implementation of SIF for Android.

3.1 Overview of SIF

Figure 1 describes the overall workflow for SIF. A user provides three pieces of information as input to SIF. The first is the original app binary to be instrumented. The second is the user-specified instrumentation code, written in a language called SIFScript[1]. We say that a SIFScript codifies an *instrumentation task*. The third input to SIF is a *workload description*. Intuitively, a workload description captures the app use-cases that the user is interested in instrumenting. For example, in the critical path analysis example above, the user may be interested in knowing the user-perceived latency for posting to Facebook. This use-case (posting to Facebook) is encapsulated in a workload description obtained from a *workload generator* (Figure 1). We describe later how a user provides a workload descriptor. The workload description is used by SIF to provide accurate overhead feedback, as described in Section 3.3.

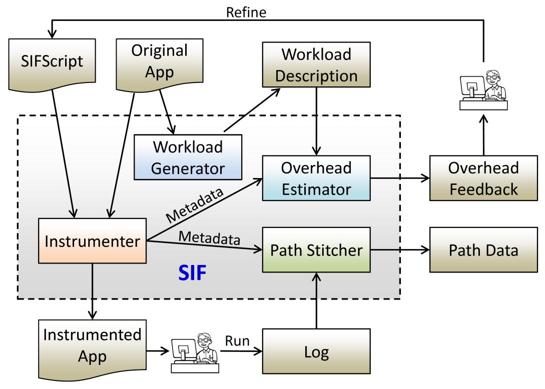


Figure 1: Overview of SIF

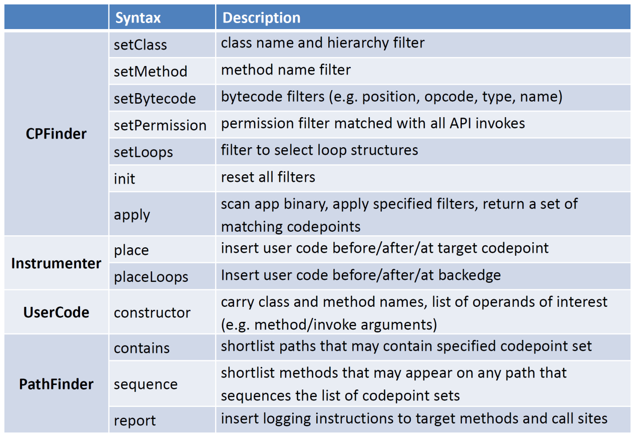
In the first step of SIF's workflow, the *instrumenter* component interprets the SIFScript specification and generates an instrumented version of the app. This *instrumenter* realizes the user-level specification and path inspection capabilities in SIF by inserting the user-specified instrumentation code at the appropriate locations. The instrumenter also outputs some additional metadata used in later stages. In our current instantiation of SIF, all instrumentation output is stored locally on the mobile device, then extracted for post-processing. In future work, we plan to explore automatic export of instrumentation output to a cloud server, a capability that can enable large-scale debugging and app analytics in the wild.

The metadata generated by the instrumenter, together with the workload information input by the user, is fed to an *overhead estimator*. That component calculates the impact of the instrumentation for the given workload description. Impact may be measured in terms of the extra execution time or additional resource usage (*e.g.,* CPU cycles, memory, energy) incurred as a result of the instrumentation. If the estimated impact is unacceptable, users can refine their instrumentation specifications.

When the instrumented application is run, the instrumentation outputs either log data generated by SIF defaults or the data collected by the user-specified instrumentation. An example of the latter is execution timings generated by user-specified instrumentation. In addition, SIF produces output whenever the user employs its path inspection capability. This output is an intermediate description of paths traversed; a SIF module called the *path stitcher* component is automatically invoked on this output to generate user readable path information. In the remainder of this section, we describe the components of SIF.

Before we do so, a word about the potential users of SIF. In our

[1]In what follows, we will use SIFScript to denote both the language and the specification program; the usage will be clear from the context.



	Syntax	Description
CPFinder	setClass	class name and hierarchy filter
	setMethod	method name filter
	setBytecode	bytecode filters (e.g. position, opcode, type, name)
	setPermission	permission filter matched with all API invokes
	setLoops	filter to select loop structures
	init	reset all filters
	apply	scan app binary, apply specified filters, return a set of matching codepoints
Instrumenter	place	insert user code before/after/at target codepoint
	placeLoops	Insert user code before/after/at backedge
UserCode	constructor	carry class and method names, list of operands of interest (e.g. method/invoke arguments)
PathFinder	contains	shortlist paths that may contain specified codepoint set
	sequence	shortlist methods that may appear on any path that sequences the list of codepoint sets
	report	insert logging instructions to target methods and call sites

Figure 2: Operations on SIF abstractions

view, SIF is an instrumentation tool at an intermediate level of abstraction. It is intended for an expert user, such as a researcher or a software engineer, who understands the app code and/or the mobile OS API well, and who might, without SIF, have manually instrumented apps for whatever task he/she is interested in (or developing custom software for this instrumentation). It can be made more broadly available to other users by adding front-end code that provides a higher-level of abstraction: for example, a security researcher can make available a web page which takes a binary and instruments it for some purpose (say to block ads), and users wishing an ad-free version of their app can upload a binary, retrieve the instrumented binary and run it. Finally, it is not unreasonable to expect developers to use SIF even when they have access to app source code: instrumentation is a programming concern that is separable from application logic, and a tool like SIF, which allows developers to treat instrumentation as a separate concern rather than having to weave instrumentation into application logic, might be helpful in many cases.

3.2 The SIFScript Language

Our first design choice for SIF was to either define a new domain-specific language for SIFScript or to realize SIFScript as an extension to an existing language. A new language is more general since it can be compiled to run on multiple mobile platforms, but it also incurs a higher learning curve. Instead, we chose to instantiate SIFScript as a Java extension. This design option has the advantage of familiarity, but may limit SIF's applicability to some mobile OS platforms. However, we emphasize that the abstractions and the underlying instrumentation methods based on program analysis are independent of specific mobile app programming platforms, and are extensible to multiple platforms.

The next design challenge for SIF was to identify abstractions that provided sufficient expressivity and enabled a variety of instrumentation tasks. In addressing this challenge, we were guided by the requirements identified in Section 2 and the instrumentation tasks described in Section 4.

An instrumentation specification language should permit instrumenting code according to different attributes, such as method invocations, specific bytecodes, or classes. The language should also allow for combining these attributes in different ways to build up sophisticated instrumentation specifications. To permit maximum flexibility and cover the use cases discussed in Section 2 and Section 4, SIFScript incorporates two qualitatively different instrumentation abstractions, codepoint sets and path sets. These abstractions

```
class TimingProfiler implements SIFTask {
  public void run() {
    CPFinder.setBytecode("invoke.*", ".* native .*");
    UserCode code;
    for (CP cp in CPFinder.apply()) {
      code = new UserCode("Logger", "start", CPARG);
      Instrumenter.place(code, BEFORE, cp);
      code = new UserCode("Logger", "end", CPARG);
      Instrumenter.place(code, AFTER, cp);
    }
  }
}
class Logger {
  private static Map map = new HashMap();
  public static void start(int mid, int pos) {
    long id = Thread.currentThread().getId();
    String k = mid + "," + pos + "," + id;
    long start = System.nanoTime();
    map.put(k, start);
  }
  public static void end(int mid, int pos) {
    long end = System.nanoTime();
    long id = Thread.currentThread().getId();
    String k = mid + "," + pos + "," + id;
    long start = map.get(k);
    Log.v(TAG, k + "," + (end - start));
  }
}
```

Listing 1: Timing profiler for native invokes

```
class LocationAuditor implements SIFTask {
  public void run() {
    CPFinder.setPermission(LOCATION);
    Set<CP> X = CPFinder.apply();
    CPFinder.setPermission(INTERNET);
    Set<CP> Y = CPFinder.apply();
    PathFinder.sequence(X, Y);
    PathFinder.report();
  }
}
```

Listing 2: Location auditor

specify *where* in a binary program the user wishes to insert instrumentation.

Codepoint Set This abstraction encapsulates a *set* of instructions (*e.g.,* bytecodes or invocations of arbitrary functions) in the binary program that share one or more *attributes*. For example, a user might define a codepoint set that consists of all invocations to a specified library (we discuss other attributes below).

Path Set This abstraction encapsulates the set of dynamically traversed paths that satisfy a user-specified *constraint*. Currently, SIF supports two forms of constraints: paths traversing any codepoint in a codepoint set or paths containing at least one codepoint from each of two or more codepoint sets.

Figure 2 documents the operations on these abstractions; these operations are discussed in greater detail below.

3.2.1 *Codepoint Sets*

We now discuss the semantics of SIFScript abstractions using a simple example instrumentation task. Listing 1 shows the complete SIFScript listing of a *timing profiler*, which selectively profiles the execution time of *native code* invocations. Modern smartphone OSs (e.g., iOS and Android) permit apps to implement part of their functionality at a lower-level in native code (usually C) for performance reasons. Native code is used in many apps such as browsers, video display and gaming.

SIFScript allows users to specify instrumentation tasks by defining separate classes for each task; each instrumentation task inherits from a `SIFTask` base class. Users can *select* arbitrary codepoints, *define* user-specified instrumentation, and specify *where* to place instrumentation. Line 3 of Listing 1 is an example of a SIFScript construct for selecting codepoints. `CPFinder` is a class that provides methods to specify *codepoint attributes* and iterate on the identified codepoint sets. In line 3, the `setBytecode()` method selects all points in the binary that are invocations to native methods. More generally, `setBytecode()` takes as its first argument a regular expression that specifies the kind of bytecode (in this example, invocations), followed by an optional argument that specifies a regular expression matching the name (in this example, native invocations).

Although not shown in our example, SIFScript contains a hierarchy of attribute specifications, which users may use to progressively narrow codepoint selections. The `setClass()` method of `CPFinder` selects classes whose name or whose class hierarchy matches specified regular expressions. If this method is invoked, only codepoints within matching classes are considered for inclusion in a codepoint set. Within these classes, users may narrow down the scope of codepoint selection by using `setMethod`, which takes, as an argument, a regular expression for the method names. Only codepoints within the matching methods are considered. Thereafter, users may invoke `setBytecode()` to specify codepoints inside the relevant classes and methods. Users may also use `setPermissions()` to refine the selection to those codepoints that require resource access permissions (e.g., network or location access) and `setLoops()`, which allows users to instrument loop edges.

If any of these attributes are not defined, the effect is equivalent to specifying *no* refinement. For example, if `setClass()` is omitted, all classes are considered when selecting codepoints. Thus, in Listing 1, line 3 selects native methods in *all* classes.

The `CPFinder` class also contains two other methods. `init()` resets a selection, since a SIFScript might contain multiple instrumentation steps, with each step instrumenting a different selection of codepoints. `apply()` (line 5) analyzes the specified attribute and computes the resulting codepoint set.

Once a codepoint set has been defined (as in line 3), the next step in writing a SIFScript is to specify what instrumentation to insert, and where to insert instrumentation. For the former, SIFScript defines a `UserCode` type which declares a code block; a new code block can be specified in the constructor to `UserCode` which takes as arguments a class name, a method name, and arguments to the method. Thus, an instance of `UserCode` effectively specifies arbitrary user-specified instrumentation. For example, in line 6, the SIFScript defines `code` to be the `start` method of the `Logger` class, and in line 8, the `end` method. Lines 13-27 provide the definitions of these methods. The `start` method generates and stores a timestamp along with a thread specific key. The `end` method computes the invocation time and writes it out to a log.

To specify where to insert instrumentation, SIF provides an `Instrumenter.place()` method. This method takes three arguments: a `UserCode` instance, a *location specification*, and a codepoint. The semantics of `place()` are as follows: it places the `UserCode` code block instance at the specified codepoint. SIF currently supports three location specifications: `BEFORE` inserts the code block before a codepoint, `AFTER` inserts it after the codepoint, and `AT` replaces the codepoint. In Listing 1, line 7 shows the `start` method of `Logger` being inserted before the codepoint, and line 9 shows the `end` method being inserted after the codepoint. The `Instrumenter` also supports a `placeLoops()` method to instrument loop back-edges; this is discussed in Section 3.3.

3.2.2 Path Sets

In Section 2, we motivated the need for a path inspection capability in a mobile app instrumentation framework. To illustrate SIF's abstraction for path inspection, consider the context-aware app `aroundme`, which searches for points of interest near a mobile device's current location. To achieve this, the app requests access permissions for both location data and the Internet. But, since it displays advertisements in the free version, privacy-conscious users may be interested in knowing whether the app leaks their location information. To audit how their location information is used, users can write a *location auditor* instrumentation task in SIF using the path inspection abstraction, as shown in Listing 2. This auditor provides the user with a listing of all sub-paths that access location data and then access the Internet.

The basic abstraction for path inspection in SIF is the *path set* provided by the `PathFinder` class. Conceptually, a path set consists of a collection of *paths traversed by the app when it is executed*. Thus, unlike the codepoint set abstraction, the set of paths belonging to a path set cannot be enumerated statically (i.e., before execution).

As with codepoint sets, path sets are specified by describing attributes of paths for interest. SIF currently supports two forms of attribute specifications. The `contains(`C`)` method of `PathFinder` takes as an argument a codepoint set, and returns all *intra-procedural* paths (i.e., paths that begin and terminate within the same procedure) that contain at least one of the codepoints in the argument. `sequence(`$C_1, C_2, \ldots, C_n$`)` specifies all inter-procedural paths that contain, in sequence, a member of each of the n codepoint sets. Thus, a path in this set contains a codepoint $c_i \in C_1$ followed by a $c_j \in C_2$, and eventually $c_k \in C_n$. The path starts in the method containing c_i and ends in the method containing c_k.

SIF supports one *action* on path sets, `report()`, which logs all paths in a path set, so a human can inspect them. This log contains every instruction in the path, so a user can understand what operations are performed along a path.

In the location auditor (Listing 2), the user defines two codepoint sets, the first is all invocations (e.g., API calls) with permission to access location data and the second is all invocations with permission to perform network operations. At runtime, the location auditor logs all paths between an invocation in the first codepoint set and an invocation in the second codepoint set. If the output is empty, the user knows that there is no direct leakage of location information, for the tested use cases. If the output is non-empty, the user can examine the processing done on the location data before a network operation occurs, for example, to determine whether the location granularity was coarsened to the zip code level.

3.3 SIF Component Design

In this section, we describe how various components of SIF are designed and how they collectively realize the abstractions described above. SIF's design borrows from program analysis techniques and abstractions. Before we discuss the SIF design, we introduce some of these techniques and abstractions.

3.3.1 Preliminaries

A *control flow graph* (CFG) represents the flow of control (branching, looping, procedure calls) in a program or within a method. Nodes in the graph represent basic blocks of code and edges represents jumps or branches. CFGs are often used in many static analysis applications.

A *call graph* captures the invocation relationship among methods within a program. A static call graph can be constructed by analyzing a program and constructing relationships between callers and callees. A dynamic call graph depicts the sub-graph of the static call graph that is encountered during an execution and can be constructed by instrumenting and logging invocations.

SIF uses a technique called *efficient path profiling* [8] proposed by Ball and Larus. This technique instruments programs to accurately, but with minimal overhead, measure path execution statistics. The Ball-Larus profiler assigns weights to edges of a method's control-flow graph (CFG) such that the sum of the edge weights along each unique path through the CFG results in a unique path identifier; a single instrumentation counter per method then suffices to record the path traversed during each invocation of the method. When a program instrumented with these counters is executed, the output is a count, for each path, of the number of times the path is executed.

More precisely, the Ball-Larus profiler instruments *path segments*. For example, in methods with two branches, there are two path segments, the *then* and *else* branches. In methods with a single loop, there are four acyclic path segments: one that runs through the method without executing the loop body, a second that starts at the beginning and terminates at the end of the loop body, a third containing the execution of the loop body to the end of the method, and a fourth containing only the loop body. Intuitively, any loop execution can be described using a linear combination of these path segments. An execution that does not execute the loop body will result in a count vector $\langle 1,0,0,0\rangle$; one iteration will result in $\langle 0,1,1,0\rangle$ and k iterations in $\langle 0,1,1,k-1\rangle$. The complete path can be reconstructed *post facto* by correlating the outputs of the Ball-Larus profiler with the CFG.

We extend the Ball-Larus profiler to handle nested method calls, exceptions, and concurrency. We identify and discard paths that result in exceptions, since their catch blocks may cause control-flow to jump outside of the method, a behavior for which the Ball-Larus profiler is undefined. Concurrency is handled by using the thread's ID to identify the counter that must be updated to track the path ID.

3.3.2 Realizing the Codepoint Set Abstraction

There are three distinct parts to realizing the codepoint set abstraction: finding target instrumentation positions, enabling access to local data variables, and inserting user-defined action code.

Finding the target instrumentation positions is performed by the method `CPFinder.apply()`. This method first combines the regular expression based attribute specifications for class hierarchy, classes, methods, bytecodes and permissions into a set of constraints. Then, the method hierarchically applies these constraints during successive scans of the code, ultimately identifying the set of instructions that need to be instrumented.

A challenge for SIF is to provide user-defined instrumentation with access to program state that is available at each instrumentation codepoint. For example, consider a codepoint `invoke-bar(x,y)` inside method `foo(a,b)`. User-specified instrumentation code should be able to access the method signature (`foo(int,int)`), method arguments `(a,b)`, and operands of the instruction (*i.e.,* the method reference to `bar`, invocation arguments `(x,y)`, and the return value, if any). This type of access to the program state is necessary, for example, in code that tracks the values of arguments supplied to `bar`.

For method signatures, SIF scans the binary to extract these signatures, then inserts instructions to load at runtime the corresponding signature at the appropriate locations, so that they will be accessible to user-specified instrumentation code. SIF also provides users with access to other information discussed above. To access this information, SIF makes special symbols available to users that notify SIF to instrument in such a way so that this data is provided to the inserted instrumentation code as arguments. When SIF encounters these special symbols, it inserts additional instrumentation to make this data available to the user-specified instrumentation code.

SIF inserts the user-specified instrumentation at all of the identified codepoints via the `Instrumenter.apply()` method. There are two approaches to this: one is to inline the instrumentation code at each codepoint, and the other is to insert an invocation to the instrumentation code. SIF employs the latter approach, which results in more compact instrumentation if there is more than one codepoint that must be instrumented with the same code.

As an aside, we note that code obfuscation frameworks, which obfuscate binaries without affecting functionality, can limit the applicability of SIF's codepoint abstraction. These frameworks cannot obfuscate invocations to system APIs and methods, so SIF will still be able to instrument those codepoints.

Supporting Distinguished Codepoints. Apps contain distinguished codepoints that correspond to higher-level program constructs of interest to users. These codepoints are method entry and exit, CFG branch edges, exception entry points, and loop back-edges. Our current instantiation of SIF supports only the subset required to achieve the instrumentation tasks discussed in Section 4, but the remainder are straightforward to support using CFG analysis. To instrument method entry and exit, a user would first identify codepoints that define a method using the `CPFinder` then use the `Instrumenter.place()` method with the location specification of either `ENTRY` or `EXIT`. To identify codepoints related to loop "back-edges" (jumps back to the beginning of the loop), SIF provides the `CPFinder.setLoops()` method. This method uses a depth-first search of the CFG to identify back-edges. Then the `Instrumenter`'s `placeLoops()` function inserts instrumentation `before` a back-edge (at the loop exit), `after` a back-edge (at the loop entry), and `at` a back-edge.

3.3.3 Realizing the Path Set Abstraction

In SIF, path sets support path inspection capabilities. The methods to support path inspection take codepoint sets as arguments; the techniques discussed above can be used for identifying the relevant codepoints. The remainder of this section discusses how SIF realizes path inspection. In general, SIF provides path inspection capabilities by appropriately adapting the Ball-Larus path profiling method discussed above, and using a *path stitcher* as a post-processing step to aggregate the path segments produced by path profiling into method-level or inter-procedural paths. SIF's adaptations reduce the overhead of path profiling.

Implementing the `contains(`C`)` method of `PathFinder` is conceptually straightforward. One could simply instrument all path segments of all methods in an app using the Ball-Larus profiler, then `report` only those path segments containing the specified codepoints (these can be identified in a post-processing step). Instead, in SIF, we use each $c \in C$ to identify the methods that contain c, and only instrument those methods.

Implementing `sequence()` is a little bit more involved. Extending Ball-Larus profiling to inter-procedural paths is known to be a hard problem, and it is not one we solve in this paper. Our approach relies on the observation that we know which inter-procedural paths are of interest, namely, the ones that traverse specified codepoint sets. We can use the Ball-Larus profiler, together with additional instrumentation, to find these inter-procedural paths.

We discuss the algorithm for the case when the input to `sequen-`

`ce()` contains two codepoint sets C_1 and C_2. The extension of this algorithm to multiple codepoint sets follows in a straightforward manner.

For `sequence`(C_1, C_2), we wish to record all paths at runtime that execute a $c_i \in C_1$ followed by a $c_j \in C_2$. We could instrument every method in the app using the Ball-Larus profiler, but this approach adds unnecessary instrumentation and does not give us enough information to determine inter-procedural paths between each c_i and c_j.

Instead, we use a more sophisticated program analysis to statically determine an over-approximation of all the methods invoked between the codepoints in C_1 and C_2. To identify these methods that could be invoked between codepoints, we perform a standard reachability analysis (with some additional inputs to specify concurrency and event handling constraints) between all $c_i \in C_1$ and $c_j \in C_2$. All intervening methods in the call graph are marked for instrumentation. We then add additional instrumentation beyond that required by the Ball-Larus profiler, described below, to instrument these methods. This approach trades off slightly higher instrumentation costs for a much more compact instrumented binary.

The additional instrumentation identifies the identifier of the path segments and ordering of the path segments, so that the path stitcher can reconstruct the intra-procedural path. For example, suppose that codepoint c_i is invoked in procedure A along path $p_{l,A}$ (where l is an identifier for the path). One approach to obtaining the inter-procedural path is to record l, A and the timestamp at which that path was executed. Then, when codepoint c_j is invoked in procedure B along path $p_{m,B}$, m, B is also output by the instrumentation. If l, A and m, B occur successively in the output, then we can infer that B was called by A and the corresponding inter-procedural path consists of l followed by m. In SIF, a *path stitcher* performs this analysis of paths.

Rather than output timestamps and path segment identifiers, we note that it suffices to simply output the sequence of path segment identifiers encountered during the execution. From this sequence, and by logging all call sites encountered during execution[2] and every method entry and exit, it is possible to stitch together inter-procedural paths. We optimize this logging by run-length encoding the path identifier sequence and compactly encoding the call site information.

Finally, the path stitcher performs a *call stack simulation* in order to determine the calling sequence of the path segments from the executed methods. To do this, it uses the logged records from path profiling and simulates the call stack encountered during execution. The output of this step is the set of all inter-procedural paths between each $c_i \in C_1$ and $c_j \in C_2$.

Dealing with Exceptions and Concurrency. Mobile apps may throw exceptions and contain concurrent threads. To handle exceptions, we conservatively include every exception handler when performing the reachability analysis. To handle concurrency, we log thread identifiers so the path stitcher can separate path segments executed by different threads. Our path stitcher can also splice thread paths when one thread creates another (by searching for a thread fork call) and identify thread identifier reuse (by determining when the thread has exited).

3.3.4 Overhead Feedback

The final component of SIF is the overhead feedback estimator. In SIF, overhead can come from two sources: instructions inserted by SIF components (*e.g.,* instructions to load method templates or perform path logging) and user-specified instrumentation code. As we show in Section 4, the former component is small. However, as in any programming framework that provides flexibility, users can insert instrumentation that adds significant processing overhead to an app, making the app unusable. To give the user approximate feedback on the overhead introduced by their instrumentation, SIF provides an overhead feedback estimator.

The overhead of instrumentation is difficult to define statically (*i.e.,* without running the app) since, in many cases, execution time is dependent on code structures that can execute a variable number of times. So, SIF provides users with a way to provide a *workload* as input. Intuitively, a workload captures the dynamic execution statistics for a given use of the app (*e.g.,* playing one instance of a game or sending an email). To translate the workload into paths that can be analyzed, SIF provides the user with a version of the app instrumented only with the Ball-Larus profiler (*i.e.,* the version does not contain the user-specified instrumentation). The user can execute the workload on this instrumented app and obtain the frequency of execution of every path segment in the app. This constitutes the workload.

Based on this information, the overhead estimator knows precisely which instructions inserted by SIF are executed and how many times each is executed. In addition, the overhead estimator can determine how many times user-specified instrumentation is invoked, and can account for the overhead of this as well. This estimation works well if user-specified instrumentation's CFG is acyclic; extending overhead estimation to more complex user-specified instrumentation is left to future work.

The overhead estimator combines the instruction counts derived from the workflow description and profiled estimates of execution time for each instruction to provide an estimate of the total execution time incurred by instrumentation (feedback based on other measures of overhead, such as energy, is left for future work). With this estimate, users can refine their instrumentation specification in order to reduce overhead.

We emphasize that, to determine overhead, the overhead estimator runs a version of the app instrumented using the Ball-Larus profiler (to generate the workload description), but does not need to run a version of the app that includes user-specified instrumentation. In this sense, the feedback provided to the user is an estimate. This is convenient because the user has to generate the workload description once, but may iteratively refine instrumentation several times.

3.4 Implementation of SIF for Android

We have designed SIF to be broadly applicable to different mobile computing platforms. The abstractions on which SIF depends are general in that they are based on standard programming constructs (instructions, invocations and paths). Furthermore, the components use program analysis techniques common to imperative programming, but not specific to a given programming language. We have chosen to instantiate SIF for the Android platform because of the platform's popularity and active research targeting the platform. That said, it should be conceptually straightforward to instantiate SIF on other platforms like Windows, iOS and we have left this to future work.

We made two exceptions to the generalizability goal of SIF. The first is that, along with handling concurrent execution, we implemented path set abstractions to handle Android-specific asynchronous tasks that can be executed in the background. Second, permissions support is closely modeled on Android's permissions model. The reason for this is that prior work [6] has identified, for a given per-

[2] Within a given method, another method m may be invoked at several points. Logging the call sites helps disambiguate these during path stitching.

mission capability, a list of APIs which match that capability, for example, the set of API calls that require `INTERNET` permission to use the network. SIF uses this list to implement permissions-based codepoint selection. We have left it to future work to identify the permission mappings of other platforms.

Although Android uses a dialect of Java, our specification language abstractions are implemented as an extension of Java. We do this because there are robust tools for Java bytecode manipulation. To manipulate Android programs, we first convert Dalvik bytecode to Java. To achieve this translation, we use `apktool` [1] to unpack and extract app binaries and resource files, and `dex2jar` [3] to convert Dalvik bytecode to Java bytecode. The `BCEL` [2] library is used for reading and modifying Java bytecode. Finally, Android SDK tools convert Java bytecode back to Dalvik and repack the instrumented app. We have implemented EPP [8] in Java. The total implementation of SIF is about 5,000 lines of code.

Our implementation does not handle Java reflection and dynamically loaded code instantiated by the Java class loader. Program analysis techniques for handling reflection and dynamic class loading have not progressed to the point where it is possible to accurately analyze a broad range of code. Furthermore, SIF has no visibility inside native code, which is used by several applications, but can instrument invocations to native methods. We have left these issues to future work. Finally, we note that SIF can also be used, with minor modifications to instrument arbitrary Java programs, not just mobile apps. We have left an exploration of this capability to future work also.

4. EVALUATION

The primary research question with SIF is its applicability for instrumentation. There are several aspects to this applicability: is SIF expressive enough to express a variety of instrumentation tasks, is its language compact enough to permit rapid instrumentation, is its framework efficient enough to be usable, and is its overhead feedback accurate enough for users to rely on its estimates. In this section, we address these aspects using our Android instantiation of SIF. All our experiments are conducted on Galaxy Nexus smartphones running Android 4.1.2.

4.1 Expressivity of SIF

To demonstrate the expressivity of SIF, we have implemented ten different instrumentation tasks, shown in Table 1, which demonstrate different facets of SIF. These tasks exhibit variety along several dimensions. First, they range from a simple timing profiler task that requires a single instrumentation step to sophisticated multi-step tasks, such as injecting privacy leaks and performing critical path analysis in the presence of concurrent events. Second, some of these tasks illustrate traditional uses of instrumentation, such as performance monitoring or dynamic tracing, while others are more specific to mobile apps, focusing on sensor usage and sensor data security. Third, while some of these instrumentation tasks are common, a majority of them have been motivated by recent research. Finally, these tasks use different combinations of the SIF abstractions: path sets or codepoint sets specified at different granularities (class hierarchy, method, bytecode); permissions; and the ability to instrument loops.

As Table 2 shows, the SIFScript for each instrumentation task is very compact. No SIFScript exceeds 100 lines; if we exclude FreeMarket, AlgoProf and AppInsight, the remaining tasks require less than 30 lines. This demonstrates the conciseness of the abstractions; as we shall discuss later, the larger tasks, AlgoProf and AppInsight, are approaches that use extensive instrumentation to study application behavior.

Task	Description	#Steps	Granularity	Reference
Timing Profiler	collects timing profile of native method invokes	1	bytecode	-
Call Graph Profiler	builds runtime call graph between two app methods	4	method	-
Flurry-like Analytics	collects and uploads user-specific analytics data for some methods	2	class hierarchy, method	[4]
Fine-grain Permission	allows user to decide on Internet usage for third-party library code	2	class hierarchy, class name, permission	[15] [30]
AdCleaner	removes ads from an app	1	bytecode	[24]
Privacy Leakage	exploits permission privilege to collect and upload sensor data	1	class hierarchy, method, permission	-
FreeMarket	exploits Google IAB vulnerabilities	3	bytecode	[26]
AlgoProf	estimates programs' cost function	10	method, bytecode, loop	[29]
AppInsight	profiles upcalls and critical paths	12	method, bytecode	[25]
Location Auditor	collects all execution paths between location and internet use	3	permission path set	-

Table 1: Implemented instrumentation tasks

	LOC* (SIF)	LOC (user)	LOC (total)
Timing Profiler	12	16	28
Call Graph Profiler	30	22	52
Flurry-like Analytics	13	18	31
Fine-grained Permission	20	44	64
AdCleaner	10	4	14
Privacy Leakage	12	44	56
FreeMarket	22	-	-
AlgoProf	61	-	-
AppInsight	91	-	-
Location Auditor	10	0	10

Table 2: Implemented SIF tasks (*Line Of Code)

Timing Profiler. The Timing Profiler shown in Listing 1 profiles the timing of native method invocations. For space reasons, instead of showing SIF code for subsequent instrumentation tasks, we use a pictorial representation of these tasks. Figure 3 shows this representation for the timing profiler. Attribute specifications for codepoint sets (line 3 in Listing 1) are represented as boxes with a gray background. User-specified instrumentation (lines 13-27 in Listing 1) is represented as a blue box with a brief description of the instrumentation code. The relative positioning of these boxes indicates whether the user-specified instrumentation is inserted before (top left), after (bottom right), or replaces (middle) the corresponding codepoints. Double arrows between filter boxes indicate multiple instrumentation steps (the Timing Profiler has only one step, but subsequent instrumentation tasks have more than one).

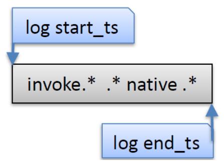


Figure 3: Timing profiler

To demonstrate the timing profiler, we have applied it to the `Angry Birds` app and measured the app's native method invokes. `Angry Birds` uses native methods to optimize UI event handling. As an aside, we note that we cannot instrument entry and exit of the Java definitions of the native methods, since these methods always have an empty body. Instead, we have to select all invoke instruction types (Android permits several invocation types), refine this codepoint set to native method invocations, and then insert the timestamp logging code before and after each such invoke.

We ran the instrumented app with a common use case (for this application, playing a single game). The results show a mismatch between the static and dynamic view of native method usage in the app. While a static analysis of the binary shows 18 distinct native method invocations, only 7 are actually involved in our use case. As Table 3 shows, some of these are used significantly more than others but exhibit significant variability in execution time (*e.g.,* update). Other are computationally expensive (*e.g.,* init and pause) and infrequently invoked.

The total size of the SIF implementation of the Timing Profiler is 28 lines of code. This illustrates the conciseness of SIFScript; even small and simple programs are able to provide meaningful and useful insights into app behavior.

	#invokes	Avg (ms)	Min (ms)	Max (ms)
setVideoReady	1	0.061	0.061	0.061
nativeInit	1	507.996	507.996	507.996
nativeInput	156	0.046	0.031	0.336
nativeKeyInput	10	0.027	0.031	0.061
nativePause	1	182.007	182.007	182.007
nativeResize	1	0.031	0.031	0.031
nativeUpdate	3696	5.605	0.366	3894.928

Table 3: Native methods invoked during a game run

Call Graph Profiler. Another common use of instrumentation is to log the dynamic call graph of an application. This task is different from the timing profiler in two ways. First, rather than instrumenting invocations, it instruments method entries and exits. Second, while the timing profiler uses a single instrumentation step, the call graph profiler uses multiple steps, where each step refers to instrumenting a distinct codepoint set or reporting a distinct path set.

Figure 4 illustrates the SIF steps to construct a call graph for all methods invoked between `X.foo()` and `Y.bar()`. The first step instruments every method entry to check a global variable that controls call graph generation. If this variable is set, the instrumentation increments a global variable that tracks the *level* of the current method in the call graph, and logs both the level and method identifier. The second step performs analogous actions for the exit of every method. In these two steps, the instrumentation location is specified by indicating a bytecode position; by default, SIF applies this to every method in every class. Finally, the last two steps instrument the entry and exit points of the target methods to (respectively) enable and disable the global variable that controls call graph generation. From the sequence of log records and level indicators, it becomes easy to infer the dynamic call graph. The SIFScript for this instrumentation task is 52 lines of code.

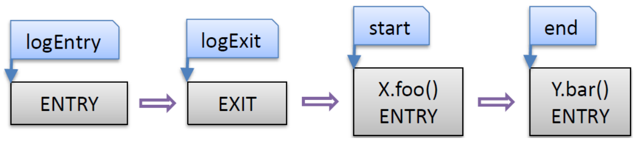
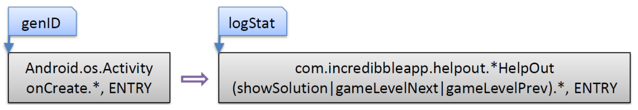


Figure 4: Call graph profiler

We verified our call graph profiler on `Angry Birds`, instrumenting the app to identify the dynamic call graph generated between calls to the app's `onCreate()` and `onDestroy()` methods. This logs 100 distinct app methods out of a total of 1958 methods in the app. The call graph has over 22K edges and a max depth of 10, with 90% of the calls being 5-6 methods deep. This example demonstrates how one can use SIF to develop insights about app structure and complexity.

An alternative implementation could use the path set `sequence` method to find all paths between `X.foo()` and `Y.bar()`, then post-process the returned paths to determine invocations and entry/exit pairs. The relative performance of these approaches depends on the app and the inputs; SIF's feedback estimator can be used to determine which approach might be better for a given workload.

Flurry-like App Usage Analytics. In mobile apps, it is common practice for developers to collect data about post-deployment app usage. In fact, there are several services, Flurry [4] being the most popular, which provide developers with an API for logging app usage information. When users use an app, these logs are uploaded to the service's site and made available to developers. This capability enables developers to refine their apps or introduce new features based on customer preferences, expertise, or other factors.



genID
Android.os.Activity
onCreate.*, ENTRY
logStat
com.incredibbleapp.helpout.*HelpOut
(showSolution|gameLevelNext|gameLevelPrev).*, ENTRY

Figure 5: Flurry-like analytics

We now show how SIF can provide usage statistics for apps whose developers have not used the analytics API while developing the app. Our code is specific to `helpout`, a multi-level puzzle game app. In this game, users advance levels when they successfully complete a level, and can choose to move down a level or have the app display a solution. We develop instrumentation to count how many times a user advances levels, chooses to move down a level, or displays a solution.

To do this in SIF requires only two steps (Figure 5) and 31 lines of code. In the first step, the SIFScript loads a unique identifier for the game player from storage (generating one if necessary) when a game starts. The SIFScript identifies the game start codepoints by create a class hierarchy based attribute specification that selects all app-defined classes derived from the Android `Activity` class and all `onCreate` methods of those classes. Then, it instruments these codepoints by inserting code for loading and generating the user identifier. This identifier is used to distinguish between multiple game users.

In the second step, the SIFScript instruments the methods that implement the functionality discussed above: going up a level, going down a level, and asking for help. Whenever one of these methods is invoked, the user-specified instrumentation sends a message to a server that includes the method name and the user identifier.

App usage analytics can help developers of `helpout` understand the distribution of expertise among their users. We had three users play an instrumented version of this game, and recorded their usage. As Table 4 clearly shows, user C has the least expertise in this game, going down a level twice and asking many times for the solution, while user B has the highest expertise, going up to level 22.

Beyond illustrating SIF's ability to support application diagnostics, this instrumentation task demonstrates how SIF allows scripts to perform more sophisticated actions beyond simply counting or timing method usage (*e.g.,* uploading data to a server).

	showSolution	gameLevelNext	gameLevelPrev
user A	2	10	0
user B	0	22	0
user C	5	7	2

Table 4: Analytics results collected from 3 users

Fine-grained Permissions. In Android, permissions to access resources, such as sensors and devices, are granted and enforced at the granularity of an entire app. However, many apps are composed of multiple “packages” obtained from different developers. For example, `aroundme` is an app that returns context-sensitive results, and its free version uses two additional packages developed by `ads` and `flurry`. The Android security model does not distinguish between the developer of the app and the developers of other packages, and treats them all as identical principals from a security perspective.

Motivated by this observation, some recent work [15, 30] has proposed finer-grained permissions granting and enforcement. This work analyzes the app to infer the right set of permissions needed for each package, then instruments the app to enforce those permissions. We now show how SIF can be used to develop functionality analogous to fine-grained permissions enforcement.

Our SIFScript code pops up a dialog box to obtain explicit permission the first time that a method in a given package invokes a certain permission. In our example, the SIFScript checks methods in the `flurry` API that require INTERNET permission. Thereafter it remembers the user's choice and only invokes the API if the user has granted the necessary permission.

Our SIFScript for this task (Figure 6(a)) contains two steps. The first step of our SIFScript selects the `onCreate` method of the entry activity and stores a reference to `Context` in the user-specified instrumentation class. Most UI actions in Android, such as popping up a dialog box, require a pointer to a UI `Context`. The second step illustrates the use of codepoint set selection based on permissions capabilities and the ability of SIF to *replace* codepoints. In this step, the SIFScript replaces all API invokes that need INTERNET permission in the `flurry` module with another method that has the same signature as the replaced method, but displays a dialog box (Figure 7) that asks for the user's choice. Subsequent invocations to any method in `flurry` will result in network traffic only if the user has granted access.

We have instrumented the `aroundme` app with this capability. Our SIFScript is 64 lines of code, and can be easily extended to enforce fine-grained permissions for modules other than `flurry` or permissions other than network access. This would require a different codepoint set definition and additional code to track different types of permissions.

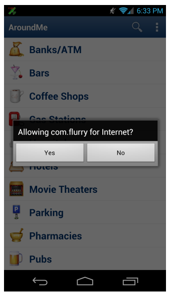
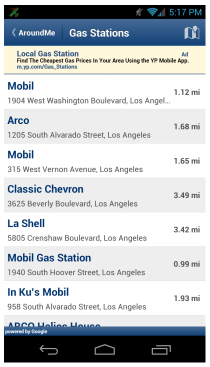


Figure 7: Dialog asking for user's choice

AdCleaner. Free versions of many mobile apps come with advertisement displays. In fact, this feature is so pervasive that there exist third-party libraries that app developers can use to include ad displays. Ads can consume Internet bandwidth, affect energy usage, and take up valuable screen real estate. Analogous to Web-based ad blockers, mobile app users can add a blacklist of domain names for ad hosting servers in their local name resolvers, but this requires root access. Recent research [24] has explored using library interposition techniques to block ads.

This capability is simple to implement in SIF, and requires only



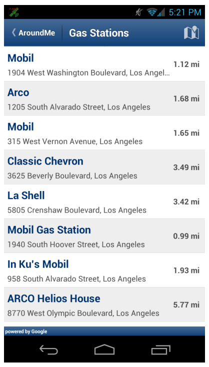
5:17 PM
AroundMe Gas Stations

Local Gas Station Ad
Find The Cheapest Gas Prices In Your Area Using the YP Mobile App.
m.yp.com/Gas_Stations

Mobil 1904 West Washington Boulevard, Los Angel...	1.12 mi
Arco 1205 South Alvarado Street, Los Angeles	1.68 mi
Mobil 315 West Vernon Avenue, Los Angeles	1.65 mi
Classic Chevron 3625 Beverly Boulevard, Los Angeles	3.49 mi
La Shell 5805 Crenshaw Boulevard, Los Angeles	3.42 mi
Mobil Gas Station 1940 South Hoover Street, Los Angeles	0.99 mi
In Ku's Mobil 958 South Alvarado Street, Los Angeles	1.93 mi

powered by Google

(a) Before



5:21 PM
AroundMe Gas Stations

Mobil 1904 West Washington Boulevard, Los Angel...	1.12 mi
Arco 1205 South Alvarado Street, Los Angeles	1.68 mi
Mobil 315 West Vernon Avenue, Los Angeles	1.65 mi
Classic Chevron 3625 Beverly Boulevard, Los Angeles	3.49 mi
La Shell 5805 Crenshaw Boulevard, Los Angeles	3.42 mi
Mobil Gas Station 1940 South Hoover Street, Los Angeles	0.99 mi
In Ku's Mobil 958 South Alvarado Street, Los Angeles	1.93 mi
ARCO Helios House 8770 West Olympic Boulevard, Los Angeles	5.77 mi

powered by Google

(b) After

Figure 8: Screenshot before and after AdCleaner

14 lines of code and a single instrumentation step. Our implementation simply replaces the `loadAd()` method invocation of a popular ad library with a null method. Figure 8 shows the screenshots of `aroundme` before and after our instrumentation is applied.

Privacy Leakage. As with many tools that have significant expressive power, SIF can also be used for malicious purposes. With this instrumentation task, we illustrate how easy it is in SIF to innocuously insert a significant privacy leak. In this example, this capability is achieved as a result of SIF's support for arbitrary user-specified instrumentation.

We have been able to instrument the `Skype` app, which is permitted to use both the camera and the network, to periodically take a picture with the camera and upload it to a website. This can, of course, significantly leak privacy by exposing the physical context of a given user. The SIFScript for this (Figure 6(c)) is 56 lines of code and requires only a single step, which instruments the `onCreate` method of the entry activity and stores a reference to the `Context` object. The inserted code *starts a background task* that periodically takes a photo and uploads it to a user server.

Figure 9 shows an experiment in which a user first uses a phone outdoors, puts the phone in his pocket, then removes the phone indoors. The photos clearly reveal these place transitions in addition to several details about the locations.

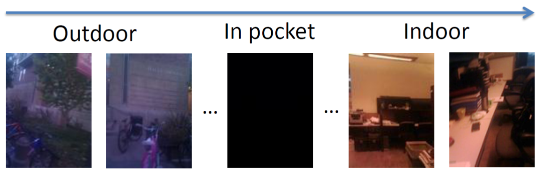
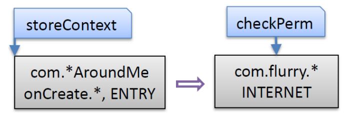


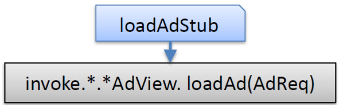
Figure 9: Photos taken and uploaded by instrumented app

FreeMarket. A recent study [26] has explored vulnerabilities in the process of in-app Android purchases. Specifically, the work proposes an attack on Google's In-App Billing service protocol that allows for users to pay for purchases from within an app. This attack instruments an app to modify its behavior to (a) bypass access to Google's billing servers, (b) redirect these calls to a local Android service (instantiated by instrumentation code) that always returns successfully, and (c) bypassing a key verification step. These instrumentation steps have been documented in [26] and we have been able to devise SIF code that replicates these instrumentation steps.

The SIFScript for this task consists of 22 lines of codepoint set specifications. The first step finds invokes to Android's `bindService` API in `Context` class and replace that binding with one to



(a) Fine-grained permission control



(b) AdCleaner

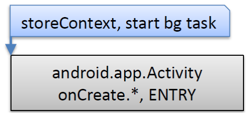
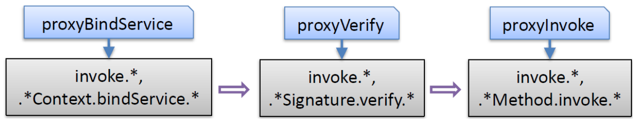

storeContext, start bg task
android.app.Activity onCreate.*, ENTRY
(c) Privacy leakage study

Figure 6: SIFScript descriptions for some tasks


proxyBindService
proxyVerify
proxyInvoke
invoke.*, .*Context.bindService.*
invoke.*, .*Signature.verify.*
invoke.*, .*Method.invoke.*
Figure 10: FreeMarket attacker

a local service. The second step replaces invocations to the signature verification Java API call with a null method that is always successful. A final step adds some instrumentation to deal with the case where in-app billing is invoked through Java reflection.

We have re-created the instrumentation steps necessary to mount the attack described in [26], but have not validated that attack because (a) the actions of the local service are not described in detail in the paper, and (b) the attack is successful only on some applications whose identities are not revealed in the paper (so we would have to search for a vulnerable app).

AlgoProf. A recent work [29] has explored methods to estimate an application's asymptotic complexity as a function of input size. Their prototype, AlgoProf, instruments Java bytecode in order to collect extensive profiling information, and then post-processes the output to produce a complexity estimate. Specifically, they instrument every method entry and exit, array access, field access, object allocation, and loop entry and exit. Although this work did not target mobile apps (its focus is on Java apps in general), it nicely demonstrates features of SIF that might be exploited in future instrumentation-based studies of mobile apps.

We have written a SIFScript to replicate the instrumentation required by AlgoProf. For space reasons, we omit a pictorial description of the code. The SIFScript has *ten* steps which require 61 lines of code (not including the user-specified instrumentation), and uses a wide range of SIF's codepoint set capabilities. Unlike previous examples that have only instrumented method entry and exit or invocations, this SIFScript also instruments Java bytecodes (*e.g.,* for field accesses) and is the only one of our examples that also instruments loops. Although our SIFScript easily duplicated the instrumentation part of the approach in only 61 lines of code, we could not verify the results of executing the instrumented apps without also duplicating AlgoProf's complexity inference algorithms, which was beyond the scope of this paper.

AppInsight. By far the most complex instrumentation task that we have applied SIF to is AppInsight [25], which analyzes latency-critical paths in mobile apps. Based on the observation that many UI actions require multiple concurrent operations, this work identifies *upcalls* and then adds instrumentation code to match upcalls with asynchronous callers.

The SIFScript for AppInsight requires *twelve* instrumentation steps requiring 91 lines of code (not including user-specified instrumentation). These steps run the gamut of location specifications and actions, instrumenting individual bytecodes and invocations, replacing invocations with user-defined instrumentation, and inserting new handlers for specified events. It was not possible to compare the output of our implementation against the original AppInsight because SIF works for Android platforms and AppInsight leverages certain opcodes and functions that are only present in Silverlight (Windows Phone). Nonetheless, our SIFScript implementation demonstrates that even complex state of the art instrumentation based approaches can be easily implemented in SIF.

Location Auditor. Finally, we have previously described another instrumentation task, location auditing (Listing 2) on the `aroundme` app. This task demonstrated the use of the pathset abstractions. An experiment involving this instrumented app is discussed below.

4.2 Efficiency of SIF

The second major aspect of SIF's design is its efficiency, which we quantify in four distinct ways. First, SIF uses program analysis to minimize the instrumentation of path sets, and we quantify these savings. Second, we demonstrate that SIF's user-perceived time to instrument a binary is moderate. Third, we quantify the runtime overhead due to SIF. Finally, we quantify the accuracy of our feedback estimator. With an accurate estimator, users can iteratively refine their instrumentation to achieve acceptable overhead.

Program Analysis for Path Sets. SIF analyzes the program binary in order to minimize the instrumentation for the `sequence` operation on path sets. To evaluate its performance, we instrumented the `aroundme` app with the location auditor task (Listing 2). `CPFinder` was able to find 10 codepoints with LOCATION permission and 12 with INTERNET permissions. Using program analysis, SIF determined that only 25 methods (out of about 560 methods in the app) needed to be instrumented, or fewer than 5% of the total number of methods. This demonstrates the benefit of sophisticated program analysis for path sets.

We also ran a simple use case that searched for nearby gas stations and restaurants. SIF reported three suspicious paths: two from the `aroundme` package and one from `ads`. As expected, `aroundme` read the location and sent it over the network. Unexpectedly, we found that the `ads` package also appeared to send a user's location over the network, presumably for location-targeted advertising.

No path was reported for the `flurry` package, indicating that, at least for this use case, there were no paths that read the GPS sensor then accessed the network. However, by analyzing network traffic, we found that `flurry` *did* leak location information. An analysis of the binary revealed that it read and stored the location in memory or storage, and later transmitted that location over the network. To detect such leaks, taint analysis or other forms of information flow analysis are necessary; SIF's path inspection capabilities can help narrow the search scope of leakage.

Time to Instrument Apps. In this section, we quantify the CPU time taken to instrument binaries for seven of the tasks presented above; in each case, we instrument the corresponding apps used to demonstrate the instrumentation tasks. Our measurements are performed on a ThinkPad T400 laptop with 3GB RAM. As shown in Table 5, the search for relevant codepoints is fast; `CPFinder` takes at most 4.6s to finish. The time to apply the instrumentation is at most 6s. Notice that the cost depends on the app binary size as well as the complexity of the relevant SIFScript. Instrumentation time is dominated by the the cost of packing and unpacking the

	Finding Codepoint	Placing Instrumentation	Unpacking + Packing	Total
Timing Profiler	2.236	3.458	18.491	24.185
Call Graph Profiler	4.532	5.972	19.371	29.875
Flurry-like Analytics	0.975	1.269	17.452	19.696
Fine-grained Permission	1.261	1.597	8.873	11.731
AdCleaner	1.093	1.325	8.755	11.173
Privacy Leakage	0.864	1.421	23.92	26.205
Location Auditor	1.569	2.093	9.014	12.676

Table 5: Time to instrument SIF tasks

app. All tasks can finish within half a minute; we consider this to be reasonable, especially since SIF itself only contributes to a small fraction of these times.

Runtime Overhead. There are two sources of overhead that affect the performance of an instrumented app: user-defined instrumentation, and instructions inserted by SIF. We now quantify the latter by replacing all user-defined code with empty stub so that the functionality of the original app is unchanged; this allows us to measure the overhead attributable to SIF. We run the same workload on both original and instrumented app and compare their running time. Table 6 shows the duration of original app and the overhead introduced by SIF. Overall, SIF introduces less than 2% overhead except for the call graph profiler, where the overhead is 4.41%. This is because SIF overhead depends on the number of codepoints to be instrumented and the call graph profiler has to instrument many more codepoints than the rest.

	Original app (sec)	Overhead by SIF (sec)
Timing Profiler	59.473	0.452 (0.76%)
Call Graph Profiler	59.729	2.637 (4.41%)
Flurry-like Analytics	115.384	0.679 (0.59%)
Fine-grained Permission	11.862	0.153 (1.29%)
AdCleaner	11.721	0.114 (0.97%)
Privacy Leakage	35.230	0.137 (0.39%)

Table 6: Runtime overhead of SIF

Accuracy of Overhead Feedback. We evaluate the accuracy of SIF's overhead estimates by comparing them with measured ground truth values. Our experiments measure execution times with and without instrumentation. The difference between these two numbers is the ground truth cost of the instrumentation. The experiments were averaged over ten runs and controlled for most types of non-deterministic behavior seen between successive runs [3].

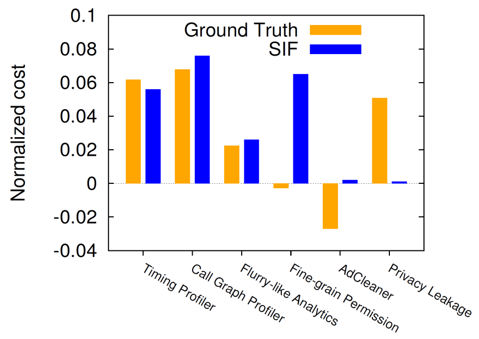


Figure 11: Accuracy of SIF's overhead estimates

Figure 11 plots the SIF estimates and measured ground truth for the six tasks we have tested completely that also involve user-specified instrumentation. The y-axis represents the instrumentation overhead measured as a fraction of the total execution time. SIF's overhead estimate is very close to ground truth, within 15%, for all tasks but privacy leakage. For fine-grained permission control and AdCleaner, the measured ground-truth has *lower* execution time after the instrumentation; that is because, in both cases, SIF replaces invocations with null methods. Our overhead estimator currently does not account for *replaced* invocations, so it over-estimates overhead. However the estimate in these situations will always provide a conservative upper bound estimate, which is appropriate, since the goal is to give the user an approximate indication of potential overhead. The one case that requires significant future work is the privacy leakage study. Its estimate is significantly off because its user-defined instrumentation is complex (the code fires a periodic timer which uploads a photo) and we have not developed analysis methods for such code.

[3] For example, we modified the `Angry Birds` binary to set its random number seed to a fixed value (by default, it uses the current local time.

5. RELATED WORK

Instrumentation frameworks have been widely used for traditional software. Adaptive Programming [18] provides a language to systematically alter classes, but does not support instrumentation of particular methods or paths. Aspect-Oriented Programming (AOP) [16] allows for instrumentation of user-defined programming points that match certain conditions, and has spawned many derivative pieces of work [19, 14, 21]. However, these pieces of work focus on specific problems and do not provide a general purpose framework for instrumentation. More general frameworks that are based on AOP include AspectJ [17], AspectC++ [28], and LMP [10]. Compared to SIF, these approaches are limited by the underlying representation of codepoints; they do not include loops [16] and are limited to method invocation, entry, and exit points. In comparison, SIF is able to arbitrarily instrument any codepoint or path-based location.

More broadly, SIF differs from AOP based instrumentation frameworks in four ways. First, it provides mechanisms for identifying and utilizing path-based information. Second, although AOP allows for loop structures to be used as codepoints, this is not adequate for extracting complete path information; SIF's instrumentation mechanisms allow for a more complete handling of language constructs such as loops and exceptions. Third, unlike AOP, SIF provides inter-procedural instrumentation mechanisms in addition to intra-procedural instrumentation. Finally, SIF provides support for multi-threaded path information reporting; in AOP it is difficult to distinguish thread information at local codepoints, but SIF can distinguish paths that belong to different threads by taking advantage of global path variables.

An earlier binary instrumentation framework, Metric Description Language (MDL) [12], allows users to dynamically record information for x86 instructions. MDL is tightly coupled with x86, restricts the instrumentation that can be inserted to either counter or timers, and does not support path-based structures, such as loops and branches. In contrast, SIF permits arbitrary user-specified instrumentation, and supports path-based structures like loops and branches.

DynamoRio[7] and Pin[20] are dynamic code manipulation frameworks. They run x86 instructions on interpreters using Just In Time Translation (JIT) and perform instrumentation while executing the programs. In contrast, SIF uses static instrumentation techniques, which avoids the runtime cost of interpretation and instrumentation of these two approaches. Also, SIF can analyze the structure of an entire application during instrumentation, so it can have a global view that allows for optimization and more sophisticated instrumentation. For example, it can more readily identify subsets of relevant paths and loop structures than these approaches. Similarly, Valgrind [23] provides a Shadow Value recording, a form of instrumentation, for assembly code. Unlike SIF, however, it does not support user-defined instrumentation.

There have also been instrumentation frameworks proposed for Java or Android applications. InsECTJ [27] can record runtime information for Java applications. It can trace bytecode execution at specified points, such as method entry, and record information, such as method arguments, at these points; unlike SIF, it does not support the insertion of arbitrary user-specified instrumentation at these points. Davis *et al.* [9] have built a framework to rewrite methods in Android applications. Their framework cannot, unlike SIF, exploit the path information for more sophisticated instrumentation.

Several other pieces of work have instrumented binaries to study various kinds of application behavior: dynamic memory allocation [11], data flow anomalies [13], app billing [26], workload characterization [29], dynamic permissions checking [22], and critical path latency [25]. Although these approaches make extensive use of instrumentation, they do not provide general code instrumentation capabilities that SIF does. We have shown that SIF is expressive enough to realize the instrumentation used in many of these papers.

6. CONCLUSION

In this paper, we have described the design and implementation of SIF, a binary instrumentation framework for mobile apps whose codepoint set abstractions are able to specify instrumentation location at different granularities and incorporate resource usage permissions. Its path set abstractions allow dynamic path inspection between arbitrary codepoints, and its program analysis techniques can reduce the overhead of instrumentation. SIF is expressive enough to incorporate a variety of instrumentation tasks previously proposed in the literature, and is quite efficient.

Much work remains, however, including validating SIF on other instrumentation tasks, porting SIF to other platforms and integrating their access permissions methods into SIF, supporting advanced language features such as reflection in its path set abstractions, evaluating the effectiveness of path sets for studying privacy leakage and comparing path-sensitive analysis with more expensive information-flow style analyses, studying the usability of overhead feedback, and improving the accuracy of feedback estimation for advanced forms of user-specified instrumentation.

Acknowledgements

We would like to thank our shepherd, Z. Morley Mao, and the anonymous reviewers, for their insightful suggestions for improving the technical content and presentation of the paper.

7. REFERENCES

[1] apktool. http://code.google.com/p/android-apktool.
[2] BCEL. http://commons.apache.org/bcel.
[3] dex2jar. http://code.google.com/p/dex2jar.
[4] flurry. http://www.flurry.com.
[5] traceview. http://developer.android.com/guide/developing/debugging/debugging-tracing.html.
[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. of ACM CCS*, 2012.
[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, 2000.
[8] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. of ACM/IEEE MICRO*, 1996.
[9] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *MoST*, 2012.
[10] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *Proc. of ACM Reflection*, 1999.
[11] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. *Electronic Notes in Theoretical Computer Science*, 2005.
[12] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. of IEEE SHPCC*, 1994.
[13] J. Huang. Detection of data flow anomaly through program instrumentation. *IEEE TOSE*, 1979.
[14] J. Irwin, J. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proc. of ISCOPE*, 1997.
[15] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proc. of CCS SPSM*, 2012.
[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *Proc. of ECOOP*, 1997.
[17] R. Laddad. *AspectJ in action: practical aspect-oriented programming*. Manning, 2003.
[18] K. Lieberherr. Adaptive object-oriented software the demeter method. *PWS Boston*, 1996.
[19] C. Lopes. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.
[20] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of ACM SIGPLAN Notices*, 2005.
[21] A. Mendhekar, G. Kiczales, and J. Lamping. Rg: A case-study for aspect-oriented programming. Technical report, SPL97-009, 1997.
[22] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. of CCS SPSM*, 2010.
[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM PLDI*, 2007.
[24] P. Pearce, P. A. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proc. of ACM ASIACCS*, 2011.
[25] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Proc. of ACM OSDI*, 2012.
[26] D. Reynaud, T. Song, E. Magrino, R. Wu, and Shin. Freemarket: Shopping for free in android applications. *NDSS*, 2012.
[27] A. Seesing and A. Orso. Insectj: a generic instrumentation framework for collecting dynamic information within eclipse. In *Proc. of OOPSLA on Eclipse Technology eXchange*, 2005.
[28] O. Spinczyk, A. Gal, and W. Schröder-Preikschat.

Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proc. of ACM TOOLS Pacific*, 2002.

[29] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proc. of ACM PLDI*, 2012.

[30] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proc. of TRUST*, 2011.

APPENDIX

A. MORE SIFSCRIPT PROGRAMS

```
class FineGrainPerm implements SIFTask {
  public void run() {
    CPFinder.init();
    CPFinder.setClass("com.tweakersoft.aroundme.
         AroundMe", "android.app.Activity");
    CPFinder.setMethod("onCreate:\\(Landroid\\/os\\/
         Bundle;\\)V");
    CPFinder.setBytecode(ENTRY);
    UserCode code;
    for (CP cp : CPFinder.apply()) {
      code = new UserCode("Logger", "storeContext",
           THIS);
      Instrumenter.place(code, BEFORE, cp);
    }
    CPFinder.init();
    CPFinder.setClass("com.flurry.*", null);
    CPFinder.setPermission(INTERNET);
    for (CP cp : CPFinder.apply()) {
      code = new UserCode("Logger", "checkPerm",
           ALL_ARGS);
      Instrumenter.place(code, AT, cp);
    }
  }
}
class Logger {
  private static Activity act;
  private static short allow = -1;
  public static void storeContext(Object obj) {
    act = (Activity) obj;
  }
  public static Object checkPerm(Object obj, String
       name, Object... args) {
    if (allow < 0) {
      create_dialog(act);
    }
    Object ret = null;
    if (allow > 0) {
      List<Class> params = new ArrayList<Class>();
      for (Object arg : args) {
        params.add(arg.getClass());
      }
      try {
        Method mthd = obj.getClass().getMethod(name,
             params);
        ret = mthd.invoke(obj, args);
      } catch (Exception e) {}
    }
    return ret;
  }
  private static void create_dialog(Context con) {
    AlertDialog.Builder builder = new AlertDialog.
         Builder(con);
    builder.setCancelable(true);
    builder.setTitle("Allowing com.flurry for
         Internet?");
    builder.setInverseBackgroundForced(true);
    builder.setPositiveButton("Yes", new
         DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int
           which) {
        allow = 1;
        dialog.dismiss();
      }
    });
    builder.setNegativeButton("No", new
         DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int
           which) {
        allow = 0;
        dialog.dismiss();
      }
    });
    AlertDialog alert = builder.create();
    alert.show();
  }
}
```

Listing 3: Fine-grained permission control

```
class PermLeakage implements SIFTask {
  public void run() {
    CPFinder.init();
    CPFinder.setClass(null, "android.app.Activity");
    CPFinder.setMethod("onCreate:\\(Landroid\\/os\\/
         Bundle;\\)V");
    CPFinder.setBytecode(ENTRY);
    for (CP cp : CPFinder.apply()) {
      UserCode code = new UserCode("Logger", "start",
           THIS);
      Instrumenter.place(code, BEFORE, cp);
    }
  }
}
class Logger {
  public static void start(Object obj) {
    Activity act = (Activity) obj;
    Context context = act.getApplicationContext();
    new Timer().schedule(new MyTask(context), 0,
         10000);
  }
}
class MyTask extends TimerTask {
  private SurfaceView view;
  private Camera cam;
  private PictureCallback jpegPictureCallback = new
       PictureCallback() {
    public void onPictureTaken(byte[] data, Camera
         camera) {
      FileOutputStream fos = null;
      String fname = String.format("/sdcard/%d.jpg",
           System.currentTimeMillis());
      try {
        fos = new FileOutputStream(fname);
        fos.write(data);
        fos.close();
      } catch (Exception e) {}
      HttpClient httpClient = new DefaultHttpClient()
           ;
      HttpPost httpPost = new HttpPost("http://enl.
           usc.edu/upload.php");
      MultipartEntity multiPart = new MultipartEntity
           ();
      multiPart.addPart("my_pic", new FileBody(new
           File(fname)));
      httpPost.setEntity(multiPart);
      try {
        httpClient.execute(httpPost);
      } catch (Exception e) {}
    }
  };
  public MyTask(Context context) {
    view = new SurfaceView(context);
    cam = Camera.open();
    try {
      cam.setPreviewDisplay(view.getHolder());
    } catch (IOException e) {}
  }
  public void run() {
    cam.startPreview();
    cam.takePicture(null, null, jpegPictureCallback);
    try {
      Thread.sleep(500);
    } catch (InterruptedException e) {}
  }
}
```

Listing 4: Privacy leakage