ModelGen: Mining Explicit Information Flow Specifications from Concrete Executions

Lazaro Clapp
Stanford University, USA
lazaro@stanford.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

ABSTRACT

We present a technique to mine explicit information flow specifications from concrete executions. These specifications can be consumed by a static taint analysis, enabling static analysis to work even when method definitions are missing or portions of the program are too difficult to analyze statically (e.g., due to dynamic features such as reflection). We present an implementation of our technique for the Android platform. When compared to a set of manually written specifications for 309 methods across 51 classes, our technique is able to recover 96.36% of these manual specifications and produces many more correct annotations that our manual models missed. We incorporate the generated specifications into an existing static taint analysis system, and show that they enable it to find additional true flows. Although our implementation is Android-specific, our approach is applicable to other application frameworks.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*

General Terms

Experimentation, Algorithms, Verification

Keywords

Dynamic analysis; specification mining; information flow

1. INTRODUCTION

Scaling a precise and sound static analysis to real-world software is challenging, especially for software written in modern object-oriented languages such as Java. Typically such software builds upon large and complex frameworks (e.g., Android, Apache Struts, and Spring). For soundness and precision, any analysis of such software entails analysis of the framework. However, there are at least four problems that make the analysis of framework code challenging. First, a very precise analysis of a framework may not scale because most frameworks are very large. Second, framework code may use dynamic language features, such as reflection in Java, which are difficult to analyze statically. Third, frameworks typically use non-code artifacts (e.g., configuration files) that have special semantics that must be modeled for accurate results. Fourth, frameworks usually build on abstractions written in lower-level languages for which a comprehensive static analysis may be unavailable (e.g., Java's native methods). Such foreign functions appear as missing code to the static analysis of the higher-level language.

One approach to address these problems is to use specifications (also called *models*) for framework classes and methods. From a high-level, a specification reflects those effects of the framework code on the program state that are relevant to the analysis. The analysis can then use these specifications instead of analyzing the framework. Use of specifications can improve the scalability of an analysis dramatically because specifications are usually much smaller than the code they specify. In addition to scalability, use of specifications can also improve the precision of the analysis because specifications are also simpler (e.g., no dynamic language features or non-code artifacts) than the corresponding code.

Although use of specifications can improve both scalability and precision of an analysis, obtaining specifications is a challenging problem in itself. If specifications are computed by static analysis of the framework code, the aforementioned problems arise. An alternative approach is to manually write specifications. This approach is not impractical because once the specifications for a framework are written, those specifications can be used to analyze any piece of software that uses that framework. However, writing and maintaining specifications manually for a large framework is still laborious and susceptible to human error. Dynamic analysis, which observes concrete executions of a program and generalizes to produce specifications, represents an attractive third alternative. Mining specifications from execution traces, to be consumed by a static analysis, is not a novel idea. For example, some techniques produce control-flow specifications (e.g., [2, 50, 34, 20, 36]), while others discover general pre- and post-conditions on methods (e.g., Daikon [15]). However, we are interested in using information-flow specifications computed through dynamic analysis as models to be consumed by a static analysis. This is a problem that, to our knowledge, has not been previously explored.

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771810

```
// Set-up objects
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder =
    Charset.forName("UTF-8").newEncoder();
TelephonyManager tMgr = ...;
// Leak phone number:
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

Figure 1: Leak phone number to Internet

Dynamic analysis has been used in the past to compute dependence summaries of methods to speed up a whole-program dynamic dependence analysis [41]. Such summaries are related, but not identical, to the information flow specifications described in this paper (see Section 6). Like previous techniques, we mine explicit[1] information flow specifications by executing each method for which we wish to construct a model and recording a trace of all operations performed by the method. Using this trace, we reconstruct the view the method has of the structures in the heap reachable from the method's arguments. We apply a specialized form of dynamic taint tracking to capture the information flows between locations inside those structures. We then lift these dynamic information flows to a static signature summarizing the flows between a method's arguments or between an argument and the return value. This lifting adds an abstraction step that is not standard in previous work, but is important to our technique. We merge the flows mined from different executions of a method to produce its overall specification.

We evaluate our generated specifications in three ways. First, we compare them to a set of specifications which were manually written over a period of two years. Our technique independently discovers 96.36% of the manual models and finds many additional correct specifications missed by human model writers. Second, we give our specifications as models for a static taint analysis. The specifications allow the analysis to discover over 31% additional flows, many of which we found to be true positives, while preserving a 98.12% recall compared to the same tool using only manual models. Third, we show that we are able to mine specifications from only a few executions of a method (average: 1.38) that are as good as those mined from large sets of traces.

We begin by giving a motivating example for the value of our technique (Section 2) and describe the overall architecture of our implementation (Section 3). We then present our specification mining technique in detail (Section 4). Next, we describe our empirical evaluation and present our results (Section 5). Finally, we summarize related work (Section 6) and conclude (Section 7).

2. MOTIVATION

As part of a long term research project to improve malware detection techniques for mobile platforms, our group has developed STAMP. STAMP is a hybrid static/dynamic program analysis tool for Android applications: The core analysis performed by STAMP is a static taint analysis that aims to detect privacy leaks. Given the code fragment in Figure 1, STAMP should infer that the device's phone number (retrieved by `getLine1Number()`) is sent to the Internet (using `socket`) and flag it as a potential leak.

[1]Explicit information flow considers only data-flow dependencies, unlike control-flow based *implicit* information flow.

Table 1: Specifications for platform methods

TelephonyManager.getLine1Number()	$PHONE_NUM → return
CharBuffer.put(String,int,int)	arg#1 → this this → return arg#1 → return
CharsetEncoder.encode(CharBuffer)	arg#1 → return
SocketChannel.write(ByteBuffer)	arg#1 → !INTERNET

STAMP performs whole-program analysis of the Android application code and any libraries bundled into its installer (.apk file). However, because of the challenges discussed in Section 1, STAMP does not directly analyze the Android platform's libraries. In Figure 1, STAMP's static analysis component has no way of inspecting the behavior of `tMgr.getLine1Number()`, `buffer.put()`, `encoder.encode()` or `socket.write()`. The simplest solution to this problem is to manually write a specification of the information flow properties of each platform method. These specifications can then be loaded by the static analysis and assumed to be an accurate representation of the corresponding methods. This is the approach we adopted for early versions of STAMP. Table 1 shows the specifications for the methods in Figure 1. The notation is as follows:

$a \rightarrow b$ indicates that there is a possible flow from a to b. Whatever information was accessible from a before the call is now potentially accessible from b after the call. If a is a reference, the information accessible from a includes all objects transitively reachable through other object references in fields.

this is the instance object for the modeled method.

return is the return value of the method.

arg#i is the i-th positional argument of the method. For a static method, argument indices begin at 0. For instance methods, *arg#0* is an alias for *this* and positional arguments begin with *arg#1*.

$SOURCE is a source of information flow and represents a resource, external to the program, from which the API method reads some sensitive information (e.g. `$CONTACTS`, `$LOCATION`, `$FILE`).

!SINK is an information sink and represents a location outside of the program to which the information flows (e.g. `!INTERNET`, `!FILE`).

Given the specifications in Table 1, STAMP can track the flow of sensitive information from `$PHONE_NUM`—through parameters and return values—to `!INTERNET`, via static analysis of the code in Figure 1.

Over a period of two years, we produced a large set of manually-written models. Generating these models was a non-trivial task, as it required running STAMP on various Android applications, discovering that it failed to find some flows, figuring out the platform methods involved in breaking the static flow path and reading the Android documentation before finally writing a model for each missing method.

In the rest of this paper we describe a technique for automatically mining explicit information flow specifications between the parameters (*this*, *arg#i*) and return values of arbitrary platform methods.

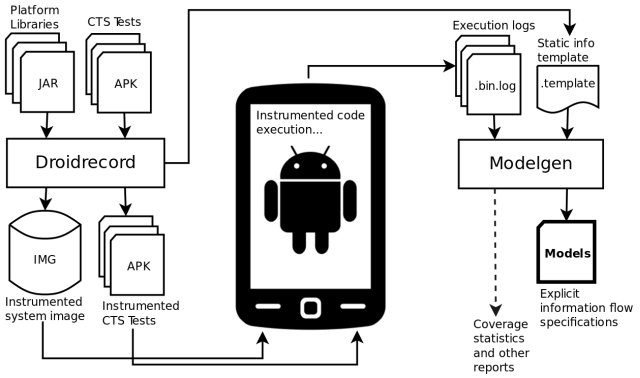


Figure 2: Architecture of Droidrecord/Modelgen

3. ARCHITECTURAL OVERVIEW

Figure 2 shows an architecture diagram of our system[2]. The first component, Droidrecord, takes binary libraries (.jar) and application archives (.apk) in their compiled form as Android's DEX bytecode. Droidrecord inserts a special logger class into every executable. Using the Soot Java Optimization Framework [49] with its Dexpler [7] DEX frontend, Droidrecord modifies each method to use this logger to record the results of every bytecode operation performed. We call each such operation an *event* and the sequence of all events in the execution of a program is a *trace*.

Once instrumented, the modified Android libraries are put together into a full Android system image that can be loaded into any standard Android emulator. For specification mining, we capture the traces generated by running the test suite for the platform methods we wish to model. In particular, we use the Android Compatibility Test Suite (CTS) [22]. We consider this kind of test suite as a type of informal specification that is available for many cases of real world systems. Good test suites include examples of method calls which capture the desired structure of the arguments and exercise edge-cases, in a way that, say, executing the method with randomly selected arguments, does not.

Running the instrumented tests over the instrumented system image produces a collection of traces. Modelgen is the component of our system that analyzes these traces offline and generates explicit information flow specifications.

Droidrecord Instrumentation. Since events are recorded to the device file system during the instrumented code's execution, it is important that their representation be compact. A compact representation reduces the slowdown resulting from performing many additional disk writes as the instrumented code executes. Controlling this slowdown is important, since the Android platform monitors processes for responsiveness and automatically terminates those which take too long to respond [23].

Droidrecord generates a template file (.template) containing all the information for each event that can be determined statically. The instrumented code stores only a number identifying the event in the template file and those values of the event that are only known at runtime. As an example, consider a single method call operation, shown below in Soot's internal bytecode format (slightly edited for brevity):

[2]The source code for our implementation and all related artifacts can be found at https://bitbucket.org/lazaro_clapp/droidrecord

```
$r5 = v_invoke $r4.<StringBuilder.append(int)>(i0);
```

When encountering this instruction, Droidrecord outputs the following event template into its .template file, including a unique template identifier:

```
17533:[MethodCallRecord{Thread: _,
       Name: <java.lang.StringBuilder.append(int)>,
       At: [...],
       Parameters: [obj:StringBuilder:_,int:_],
       ParameterLocals: [$r4,i0],
       Height: int:_}]
```

The bytecode is then instrumented to record the identifier, followed by the runtime values of the method's parameters:

```
staticinvoke <TraceRecorder.recordEvent(long)>(17533L);
staticinvoke <TraceRecorder.writeThreadId()>();
staticinvoke <TraceRecorder.writeObjectId(Object)>(r4);
staticinvoke <TraceRecorder.write(int)>(i0);
$r5 = v_invoke $r4.<StringBuilder.append(int)>(i0);
```

When reading the trace, these values are plugged into the placeholder positions ('_' above) of the event template. For some events (simple assignments, arithmetic operations, etc) all the values can be inferred statically as a simple function of the values of previous events. These events generate event templates but incur no dynamic recording overhead.

Modelgen Trace Extraction. After tests are run and traces extracted from the emulator, they are first pre-processed and combined with the static information in the .template file. The result is a sequential stream of events for each method invocation; we write $(m : i)$ for the ith invocation of method m. Calls made by $(m : i)$ to other methods are included in this stream, together with all the events and corresponding calls within those other method invocations. Spawning a new thread is an exception: events happening in a different thread are absent from the stream for $(m : i)$, but appear in the streams for enclosing method invocations in the new thread. This separation may break flows that involve operations of multiple threads and is a limitation of our implementation. We did not find any cases where a more precise tracking of explicit information flow across threads would have made a difference in our experimental results.

4. SPECIFICATION MINING

To explain Modelgen's core model generation algorithm, we describe its behavior on a single *invocation subtrace* $T_{(m:i)}$, which is the sequence of events in the trace corresponding to method invocation $(m : i)$. Recall $T_{(m:i)}$ includes the invocation subtraces for all method invocations called from m during invocation $(m : i)$, including any recursive calls to m. We now describe a simplified representation of $T_{(m:i)}$ (Section 4.1) and give its natural semantics (Section 4.2), that is, the meaning of each event in the subtrace with respect to the original program execution. Modelgen analyzes an invocation subtrace by processing each event in order and updating its own bookkeeping structures. We represent this process with a non-standard semantics: the modeling semantics of the subtrace (Section 4.3). After Modelgen finishes scanning $T_{(m:i)}$, interpreting it under the modeling semantics, it saves the resulting specification which can then be combined with the specifications for other invocations of m (Section 4.4).

4.1 Structure of a Trace

Figure 3 gives a grammar for the structure of traces, consisting of a sequence of events. Events refer to constant primitive values, field or method labels, and variables. The

$$
\begin{array}{rll}
T ::= & e^* & \text{(trace)} \\
e \in event ::= & x = pv & \text{(literal load)} \\
| & x = newObj & \text{(new object)} \\
| & x = y & \text{(variable copy)} \\
| & x = y \diamond z & \text{(binary op)} \\
| & x = y.f & \text{(load)} \\
| & x.f = y & \text{(store)} \\
| & x = m(\overline{y}) & \text{(call)} \\
| & return\ x & \text{(return)} \\
| & throw\ x & \text{(throw exception)} \\
| & catch\ x = a & \text{(caught exception)}
\end{array}
$$

$pv \in Primitive\ Value$ $\quad \diamond \in BinOp$
$a \in Address$ $\quad r \in Rec = \{\overline{f : v}\}$
$x, y, z \in Var$ $\quad \rho \in Env : Var \rightharpoonup Value$
$f \in Field$ $\quad h \in Heap : Address \rightharpoonup Rec$
$m \in Method$

Figure 3: Structure of a trace

symbol $\diamond$ stands for binary operations between primitive values. Objects are represented as records mapping field names to values, which might be either addresses or primitive values. This grammar is similar to that of a 3-address bytecode representing Java operations. However, it represents not static program structure, but the sequence of operations occurring during a concrete program run, leading to the following characteristics:

1. Conditional (`if`, `switch`) and loop (`for`, `while`) operations are omitted and unnecessary; the events in T represent a single path through the program. The predicates inside conditionals are still evaluated, usually as binary operations.

2. The values of array indices in recorded array accesses are concrete, which allows us to treat array accesses as we would object field loads and stores (e.g., $a[i]$ becomes $a.i$, and note i is a concrete value).

3. For each method call event $x = m_1(\overline{y})$ in $T_{(m:i)}$ there is a unique invocation subtrace of the form $T_{(m_1:j)} = fun(\overline{z})\{var\ \overline{x}; \overline{e}; e_f\}$ where e_f is a return or throw event and $\overline{x}$ is a list of all variable names used locally within the invocation. Again, since we cover only one path through m for each invocation, invocation subtraces may have at most one return event and must end with a return or throw event.

We avoid modeling static fields explicitly by representing them as fields of a singleton object for each class.

4.2 Natural Semantics of a Subtrace

Figure 4 gives a natural semantics for executing the program path represented by an invocation subtrace. Understanding these standard semantics makes it easier to understand the custom semantics used by Modelgen to mine specifications, which extend the natural semantics. The natural semantics of a subtrace are similar but not identical to those of Java bytecode. The differences arise from the fact that subtrace semantics represent a single execution path.

During subtrace evaluation, an environment ρ maps variable names to values. A heap h maps memory addresses to object records. Given a tuple $\langle h, \rho, e\rangle$ representing event e under heap h and environment ρ, the operator $\downarrow$ represents the evaluation of e in the given context and produces a new tuple $\langle h', \rho'\rangle$ containing a new heap and a new environment.

$$\frac{}{\langle h, \rho, x = pv\rangle \downarrow \langle h, \rho[x \to pv]\rangle} \quad \text{(LIT)}$$

$$\frac{a \notin dom(h)}{\langle h, \rho, x = newObj\rangle \downarrow \langle h[a \to \{\}], \rho[x \to a]\rangle} \quad \text{(NEW)}$$

$$\frac{\rho(y) = v}{\langle h, \rho, x = y\rangle \downarrow \langle h, \rho[x \to v]\rangle} \quad \text{(ASSIGN)}$$

$$\frac{\rho(y) = pv_1 \qquad \rho(z) = pv_2 \qquad pv_1 \diamond pv_2 = pv_3}{\langle h, \rho, x = y \diamond z\rangle \downarrow \langle h, \rho[x \to pv_3]\rangle} \quad \text{(BINOP)}$$

$$\frac{\rho(y) = a \qquad h(a) = r \qquad r(f) = v}{\langle h, \rho, x = y.f\rangle \downarrow \langle h, \rho[x \to v]\rangle} \quad \text{(LOAD)}$$

$$\frac{\rho(x) = a \qquad h(a) = r \qquad \rho(y) = v \qquad r' = r[f \to v]}{\langle h, \rho, x.f = y\rangle \downarrow \langle h[a \to r'], \rho\rangle} \quad \text{(STORE)}$$

$$\frac{\begin{array}{c} m = fun(z_1, ..., z_n)\{var\ \overline{x'}; \overline{e}; return\ y'\} \\ \forall i\ \rho(y_i) = v_i \qquad \rho'(y') = v' \\ \langle h, [z_1 \to v_1, ..., z_n \to v_n, \overline{x'} \to undef\,], \overline{e}\rangle \overline{\downarrow} \langle h', \rho'\rangle \end{array}}{\langle h, \rho, x = m(y_1, ..., y_n)\rangle \downarrow \langle h', \rho[x \to v']\rangle} \quad \text{(INV)}$$

$$\frac{\langle h_i, \rho_i, e_i\rangle \downarrow \langle h_{i+1}, \rho_{i+1}\rangle}{\langle h_0, \rho_0, e_0; \ldots; e_{n-1}\rangle \overline{\downarrow} \langle h_n, \rho_n\rangle} \quad \text{(SEQ)}$$

Figure 4: Natural semantics

The operator $\overline{\downarrow}$ represents the evaluation of a sequence of events which consists of evaluating each event ($\downarrow$) under the heap and environment resulting from the evaluation of the previous event. The rules in Figure 4 describe the behavior of $\downarrow$ and $\overline{\downarrow}$ for different events and their necessary preconditions. We omit the rules for handling exceptions since they do not add significant new ideas with respect to our specification mining technique and exception propagation complicates both the natural and modeling semantics.

We now consider how the natural semantics represent the evaluation of the following example subtrace fragment which increments a counter at $x.f$:

$$t ::= y = x.f; z = 1; w = y + z; x.f = w$$

Assuming x contains the address a (i.e., $\rho(x) = a$) of heap record $r = \{f : 0\}$ (i.e., $h(a) = r$), LOAD gives us:

$$\langle h, \rho, y = x.f\rangle \downarrow \langle h, \rho[y \to 0]\rangle$$

Applying LIT, BINOP and STORE, respectively, we get:

$$\langle h, \rho[y \to 0], z = 1\rangle \downarrow \langle h, \rho[y \to 0; z \to 1]\rangle$$

$$\langle h, \rho[y \to 0; z \to 1], w = y + z\rangle \downarrow \langle h, \rho[y \to 0; z \to 1; w \to 1]\rangle$$

$$\langle h, \rho[...; w \to 1], x.f = w\rangle \downarrow \langle h[a \to \{f : 1\}], \rho[...; w \to 1]\rangle$$

Using those evaluations for each expression, SEQ gives the full evaluation of the fragment as

$$\langle h, \rho, t\rangle \overline{\downarrow} \langle h[a \to \{f : 1\}], \rho[y \to 0; z \to 1; w \to 1]\rangle$$

where, in addition to some changes to the environment, field f of record r in the heap has been incremented by one.

4.3 Modeling Semantics of a Subtrace

To obtain the information flow facts required to construct our specifications, not only are we interested in tracking information flow through the portion of the heap reachable from the arguments and return value of m, but we also want to "lift" these flows so that they refer exclusively to

$$\frac{l = new\_loc() \qquad c = new\_color()}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = pv\rangle \downarrow \langle h, \rho[x \to pv], \mathscr{L}[x \to l], \mathbb{C}[l \to \{c\}], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mLit})$$

$$\frac{a \notin dom(h) \qquad l = new\_loc() \qquad c = new\_color()}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = newObj\rangle \downarrow \langle h[a \to \{\}], \rho[x \to a], \mathscr{L}[x \to l], \mathbb{C}[l \to \{c\}], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mNew})$$

$$\frac{\rho(y) = v \qquad \mathscr{L}(y) = l}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y\rangle \downarrow \langle h, \rho[x \to v], \mathscr{L}[x \to l], \mathbb{C}, \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mAssign})$$

$$\frac{\begin{array}{c}\rho(y) = pv_1 \qquad \rho(z) = pv_2 \qquad pv_1 \diamond pv_2 = pv_3 \\ \mathscr{L}(y) = l_1 \qquad \mathscr{L}(z) = l_2 \qquad l_3 = new\_loc() \\ C = \mathbb{C}(l_1) \cup \mathbb{C}(l_2)\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y \diamond z\rangle \downarrow \langle h, \rho[x \to pv_3], \mathscr{L}[x \to l_3], \mathbb{C}[l_3 \to C], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mBinop})$$

$$\frac{\begin{array}{c}\rho(y) = a \qquad h(a) = r \qquad r(f) = v \\ \mathscr{L}(a) = l_1 \qquad \mathscr{L}(y, f) = l_2 \\ C = \mathbb{D}(a, f)?\mathbb{C}(l_2) : \mathbb{C}(l_1) \cup \mathbb{C}(l_2)\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y.f\rangle \downarrow \langle h, \rho[x \to v], \mathscr{L}[x \to l_2], \mathbb{C}[l_2 \to C], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mLoad})$$

$$\frac{\begin{array}{c}\rho(x) = a \qquad h(a) = r \qquad \rho(y) = v \qquad r' = r[f \to v] \\ \mathscr{L}(y) = l_1 \qquad \mathscr{L}(a) = l_2 \\ \mathbb{G}' = \mathbb{G} + \{c_1 \to c_2 | \forall c_1 \in \mathbb{C}(l_1), c_2 \in \mathbb{C}(l_2)\}\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x.f = y\rangle \downarrow \langle h[a \to r'], \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}', \mathbb{D}[(a, f) \to True]\rangle} \quad (\textsc{mStore})$$

$$\frac{\begin{array}{c}m = fun(z_1, ..., z_n)\{var\ \overline{x'}; \overline{e}; return\ y'\} \\ \rho_m = [z_1 \to v_1, ..., z_n \to v_n, \overline{x'} \to undef\ ] \\ \mathscr{L}_m = \mathscr{L}[z_1 \to \mathscr{L}(y_1), ..., z_n \to \mathscr{L}(y_n), \overline{x'} \to new\_loc()] \\ \langle h, \rho_m, \mathscr{L}_m, \mathbb{C}, \mathbb{G}, \mathbb{D}, \overline{e}\rangle \overline{\downarrow} \langle h', \rho', \mathscr{L}', \mathbb{C}', \mathbb{G}', \mathbb{D}', t\rangle \\ \mathscr{L}'' = \mathscr{L}'[z_1 \to \mathscr{L}(z_1), ..., z_n \to \mathscr{L}(z_n), \overline{x'} \to \mathscr{L}(\overline{x'})] \\ \forall i\ \rho(y_i) = v_i \qquad \rho'(y') = v' \qquad \mathscr{L}(y') = l\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = m(y_1, ..., y_n)\rangle \downarrow \langle h', \rho[x \to v'], \mathscr{L}''[x \to l], \mathbb{C}', \mathbb{G}', \mathbb{D}'\rangle} \quad (\textsc{mInv})$$

$$\frac{\begin{array}{c}\langle h_i, \rho_i, \mathscr{L}_i, \mathbb{C}_i, \mathbb{G}_i, \mathbb{D}_i, e_i\rangle \downarrow \\ \langle h_{i+1}, \rho_{i+1}, \mathscr{L}_{i+1}, \mathbb{C}_{i+1}, \mathbb{G}_{i+1}, \mathbb{D}_{i+1}\rangle\end{array}}{\langle h_0, \rho_0, \mathscr{L}_0, \mathbb{C}_0, \mathbb{G}_0, \mathbb{D}_0, e_0; \ldots; e_{n-1}\rangle \overline{\downarrow} \langle h_n, \rho_n, \mathscr{L}_n, \mathbb{C}_n, \mathbb{G}_n, \mathbb{D}_n\rangle} \quad (\textsc{mSeq})$$

Figure 5: Modeling semantics

the method arguments and return value rather than intermediate heap locations. We perform both this tasks simultaneously, through the modeling semantics of the subtrace.

The modeling semantics augment the natural semantics by associating *colors* with every heap location and primitive value. For subtrace $T_{(m:i)}$, each argument to m is initially assigned a single unique color. The execution of $T_{(m:i)}$ under the modeling semantics preserves the following invariants:

Invariant I: Computed values have all the colors of the argument values used to compute them.

Invariant II: At each point in the trace, if a heap location l is accessed from an argument a using a chain of dereferences that exists at method entry, then l has the color of a.

Invariant III: At each point in the trace, every argument and the return value have all the colors of heap locations reachable from that argument or return value.

These invariants are easily motivated. Invariant I is the standard notion of taint flow: the result of an operation has the taint of the operands. Invariant II captures the granularity of our specifications on entry to a method: all the locations reachable from an argument are part of the taint class associated with that argument (recall the semantics of our specifications described in Section 2). Similarly, Invariant III captures reachability on method exit. For example, if part of the structure of *arg#1* is inserted into the structure reachable from *arg#2* by the execution of the trace, then *arg#2* will have the color of *arg#1* on exit. At every step of the modeling semantics these invariants are preserved for every computed value and heap location seen so far; the invariants need not hold for heap locations and values that have not yet been referenced by any event in the examined portion of the subtrace. In addition, reachability in Invariants II and III applies only to the paths through the heap actually accessed during subtrace execution.

The natural semantics differentiate between primitive values or addresses stored in variables of ρ and objects stored in the heap h. Although this distinction is useful in representing the subtrace's execution, for specification mining we want to associate colors with both heap and primitive values. For uniformity, we introduce a mapping $\mathscr{L}$ which assigns a "virtual location" ($VLoc$) to every variable, object and field based on origin (i.e., where the value was first created) rather than the kind of value. Because virtual locations may be tainted with more than one color (recall Invariant I), we introduce a map $\mathbb{C} : VLoc \rightharpoonup 2^{Color}$ from virtual locations to sets of colors. The modeling semantics also use $\mathbb{G} : \{(Color, Color)\}$, which is a relation on colors or, equivalently, a directed graph in which nodes are colors, and $\mathbb{D} : (Address, Field) \rightharpoonup Boolean$, which stands for "destructively updated" and maps object fields to a boolean value indicating that the field of that location has been written in the current subtrace. We explain the use of $\mathbb{G}$ and $\mathbb{D}$ below.

Figure 5 lists the modeling semantics corresponding to the natural semantics in Figure 4. We now explain how the first 4 rules preserve Invariant I, as well as how mLoad and mStore preserve Invariants II and III, respectively.

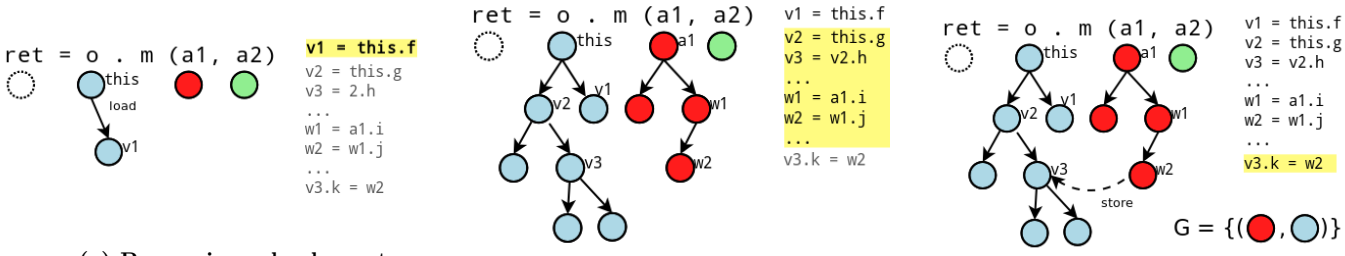
Rule mLit models the assignment of literals to variables. A new literal value is essentially a new information source within the subtrace and is assigned a new location with a new color. The location is associated with the variable now holding the value, preserving Invariant I. Rule mNew, which models new object creation, is similar. Rule mAssign models an assignment $x = y$ where x and y are both variables in ρ and does not create a new location, but instead updates $\mathscr{L}(x)$ to be the location of y, indicating that they are the same value, again preserving Invariant I.

Rule mBinop gives the modeling semantics for binary operations. Assuming locations l_1 and l_2 for the operands, the rule adds a new location l_3 to represent the result. Because of Invariant I, l_3 must be assigned all the colors of l_1 and all the colors of l_2, thus $\mathbb{C}(l_3)$ becomes $\mathbb{C}(l_1) \cup \mathbb{C}(l_2)$.

Rules mLoad and mStore deal with field locations. The virtual location of field $a.f$ (denoted $\mathscr{L}(a, f)$) is defined as either the location of the object stored at $a.f$, if the field is of reference type, or as an identifier which depends on $\mathscr{L}(a)$ and the name of f, if f is of primitive type.

Rule mLoad models load events of the form $x = y.f$ by assigning the location $l_2 = \mathscr{L}(y, f)$ to x and computing the color set for this location (which will be the colors for both x and $y.f$). There are three cases to consider:

- If this is the first time the location $\mathscr{L}(y, f)$ has been referenced within the subtrace $T_{(m:i)}$, then $y.f$ has no



(a) Processing a load event

(b) Loads reconstruct the argument's structure

(c) Processing a store event

Figure 6: Effects of loads and stores in Modelgen's Modeling Semantics

color (all heap locations except the arguments start with the empty set of colors in $\mathbb{C}$). Furthermore, since this is the first access, $y.f$ has not been previously written in the subtrace, so $\mathbb{D}(\rho(y), f) = \mathit{False}$. Therefore, l_2 is assigned the colors $\mathbb{C}(l_1) \cup \mathbb{C}(l_2)$ where $l_1 = \mathscr{L}(y)$. Since $\mathbb{C}(l_2) = \emptyset$ before the load event, we end up with $\mathbb{C}(l_2) = \mathbb{C}(l_1)$. If $y.f$ is reachable from a method argument through y, this establishes Invariant II for $y.f$ on its first access.

- If l_2 has been loaded previously in the trace but not previously overwritten, then $\mathbb{C}(l_2) = \mathbb{C}(l_1) \cup \mathbb{C}(l_2)$, indicating that l_2 now has the colors of all of its previous accesses plus a possibly new set of colors $\mathbb{C}(l_1)$. This handles the case where a location is reachable from multiple method arguments and preserves Invariant II.
- If $y.f$ has been written previously then $\mathbb{D}(\rho(y), f) = \mathit{True}$. In this case it is no longer true that $\mathscr{L}(y, f)$ was reachable from $\mathscr{L}(y)$ on method entry and so it is not necessary to propagate the color of $\mathscr{L}(y)$ to $\mathscr{L}(y, f)$ to preserve Invariant II and we omit it. Also, note that if $y.f$ has been written, that implies the value stored in $y.f$ was loaded before the write and so $y.f$ will already have at least one color.

Figure 6a shows the effect of a single load operation from an argument to m, while Figure 6b depicts the coloring of a set of the heap locations after multiple load events.

Rule mStore models store events of the form $x.f = y$. The rule updates $\mathbb{D}(\rho(x), f) = \mathit{True}$ since it writes to $x.f$. We could satisfy Invariant III by implementing mStore in a way that traverses the heap backwards from x to every argument of m that might reach x and associates every color of y with those arguments (and possibly intermediate heap locations). As an optimization, we instead use $\mathbb{G}$ to record an edge from each color c_1 of $\mathscr{L}(y)$ to each color c_2 of $\mathscr{L}(x.f)$

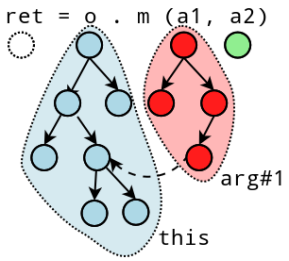


Figure 7: Stores connect argument structures

with the following meaning: $c_1 \to c_2 \in \mathbb{G}$ means every virtual location with color c_2 has color c_1 as well. Figure 6c depicts the results of a store operation, while Figure 7 depicts how $\mathbb{G}$ serves to associate two colored heap subgraphs.

Rule mInv implements standard method call semantics, mapping the virtual locations of arguments and the return value between caller and callee. Rule mSeq is the same as Seq in the natural semantics, adding $\mathscr{L}$, $\mathbb{C}$, $\mathbb{G}$ and $\mathbb{D}$.

As a consequence of Invariants I and II, the modeling semantics associate the color of each argument to every value and heap location that depends on the argument values on entry to m. Then, because of Invariant III, when the execution reaches the end of subtrace $T_{(m:i)}$ every argument and the return value have all the colors of heap locations reachable from that argument or return value (as represented by $\mathbb{G}$). We construct our specifications by examining the colors of each argument a_j and the return value r after executing the subtrace: for every color of r (or a_j) that corresponds to the initial color of a different argument a_k, we add $a_k \to r$ ($a_k \to a_j$) to our model.

4.4 Combining Specifications

For each invocation subtrace $T_{(m:i)}$, the process just outlined produces an underapproximation of the specification for m, based on a single execution $(m : i)$. We combine the results from different invocations of m by taking an upper bound on the set of argument-to-argument and argument-to-return flows discovered for every execution, which is simply the union of the results of $(m : i)$ for every i.

For example, consider the method `max(a,b)` designed to return the larger of two numbers, disregarding the smaller one. Suppose that we have two subtraces for this method: one for invocation `max(5,7)`, which returns 7 and produces the model $M_1 = \{\mathit{arg\#2} \to \mathit{return}\}$ and one for invocation `max(9,2)`, which returns 9 and produces the model $M_2 = \{\mathit{arg\#1} \to \mathit{return}\}$. Clearly the correct specification reflecting the potential explicit information flow of method `max(a,b)` is $M_1 \cup M_2 = \{\mathit{arg\#1} \to \mathit{return},\ \mathit{arg\#2} \to \mathit{return}\}$.

We should note that combining specifications in this way inherently introduces some imprecision with respect to the possible flows on a given execution of the method. The effects of this imprecision in our overall system depend on the characteristics of the static analysis that consumes the specifications. For example, the above specification for `max(a,b)` would be strictly less precise than analyzing the corresponding code (assuming the natural implementation) with an ideal path-sensitive analysis, since it merges two different control paths within the max function: one in which the first argument is greater and one in which the second argument is

greater. For context-sensitive but path-insensitive analysis such as STAMP (see Section 5.2), loss of precision due to combining specifications is less common, but still possible in theory. Consider a method `do(a,b) { a.doImpl(b) }` and two invocations of this method in which `a` has different types and each type has its own implementation of `a.doImpl(b)`. A context-sensitive analysis can tell which version of `doImpl` is executed, but Modelgen simply merges the flows observed for every version of `doImpl` seen in any trace of `do(a,b)`.

4.5 Calls to Uninstrumented Code

Our approach to specification mining is based on instrumenting and executing as much of the platform code as we can. Unfortunately recording the execution of every method in the Android platform is challenging. In particular, any technique based on Java bytecode instrumentation cannot capture the behavior of native methods and system calls. Since our inserted recorder class is itself written in Java, we must also exclude from instrumentation some Java classes it depends upon to avoid introducing an infinite recursion. Thus, traces are not always full traces but represent only a part of a program's execution. We need to deal with two problems during event interpretation: (1) How should Modelgen interpret calls to uninstrumented methods? (2) How can we detect that a trace has called uninstrumented code?

For the first problem, Modelgen offers two separate solutions. The user can provide manually written models for some methods in this smaller uninstrumented subset (as we do, for example, for `System.arraycopy` and `String.concat`). If a user-supplied model is missing for a method, Modelgen assumes a worst-case model in which information flows from every argument of the method to every other argument and to its return value. In many cases, this worst-case model, although imprecise, is good enough to allow us to synthesize precise specifications for its callers. Note that the need for a set of manual models for uninstrumented code does not negate the benefits of Modelgen, since this represents a significantly smaller set of methods. For example, to produce 660 specifications from a subset of the Android CTS (see Section 5.1) we needed only 70 base manual models.

The problem of detecting uninstrumented method calls inside traces is surprisingly subtle. Droidrecord writes an event at the beginning of each method and before and after each method call. In the simplest case we would observe these before-call and after-call markers adjacent to each other, allowing us to conclude that we called an uninstrumented method. However, because uninstrumented methods often call other methods which are instrumented, this simple approach is not enough. A call inside instrumented code could be followed by the start of another instrumented method, distinct from the one that is directly called. Dynamic dispatch and complex class hierarchies further complicate determining if the method we see start after a call instruction is the instruction's callee.

Our solution for detecting holes in the trace due to invoking uninstrumented code is to record the height of the call stack at the beginning of every method and before and after each call operation. Since the stack grows for every method call, whether instrumented or not, we use its height to determine when we have called into uninstrumented code. The usual pattern to get the stack height (using a StackTrace object) is expensive. As an optimization, we modify the Dalvik VM to add a shortcut method to get the stack height.

5. EVALUATION

We perform three studies to evaluate the specifications generated by Modelgen. First, we compare them directly against our existing manually-written models (Section 5.1). Second, we contrast the results of running the STAMP static information-flow analysis system using these specifications as input, against the results of the same system using the manual models (Section 5.2). Third, we study the effect of test suite quality on the mined specifications (Section 5.3).

5.1 Comparison Against Manual Models

To evaluate Modelgen's ability to replace the manual effort involved in writing models for STAMP (see Section 2), we compare the specifications mined by Modelgen against existing manual models for 309 Android platform methods.

We conducted all of our evaluations on the Android 4.0.3 platform, which has a total of 46,559 public and protected methods. STAMP includes manual models for 1,116 of those methods, of which 335 are inside the `java.lang.*` package which DroidRecord does not currently instrument (this is due partly to performance reasons and partly to our instrumentation code depending on classes in this package, this is not a limitation of the general technique), and 321 have only source or sink annotations, leaving 460 methods for which Modelgen could infer comparable specifications.

For our evaluation, we obtained traces by running tests from the Android Compatibility Test Suite (CTS) [22]. We restricted ourselves to a subset of the CTS purporting to test those classes in the `java.*` and `android.*` packages, but outside of `java.lang.*`, for which we have manual models (not counting simple source or sink annotations). For some packages for which we have manual models, such as `com.google.*`, the CTS contains no tests.

Table 2 summarizes our findings, organized by Java package. For each package we list the number of classes and methods for which we have manual specifications, as well as the total number of correct individual flow annotations (e.g. $arg\#X \rightarrow return$) either from our manual specifications or generated by Modelgen. We then list separately the flows discovered by Modelgen and those in our manual specifications. We consider only those flows in methods for which we have manual models and only those classes for which we ran any CTS tests, which gives us 309 methods to compare.

We evaluate Modelgen under two metrics: precision and recall. Precision relates to the true positive rate of Modelgen, listing the percentage of Modelgen flow annotations which represent actual possible information flows induced by the method. To determine which Modelgen annotations are correct, we compared them with our manual models and, when the specification for a given method differed between both approaches, we inspected the actual code of the method to see if the differing annotations represented true positives or false positives for either technique. Thus, if $FModelgen$ is the set of flows discovered by Modelgen and TP the set of all flows we verified as true positives, then Modelgen's precision is defined as:

$$PModelgen = |FModelgen \cap TP|/|FModelgen|$$

Similarly the precision of the manual models is:

$$PManual = |FManual \cap TP|/|FManual|$$

Table 2: Comparing Modelgen specifications and manual models

Package	Classes	Methods	Missing trace info.	Total correct flows	Modelgen correct flows	Manual correct flows	Modelgen false positives	Manual errors	Modelgen precision	Manual precision	Modelgen recall
java.nio.*	2	26	4	50	50	42	0	0	100%	100%	100%
java.io.*	28	146	23	280	275	234	2	0	99.28%	100%	97.86%
java.net.*	7	37	4	104	100	65	0	1	100%	98.48%	93.85%
java.util.*	4	28	0	36	36	31	0	1	100%	96.88%	100%
android.text.*	3	5	2	3	3	3	0	0	100%	100%	100%
android.util.*	2	8	1	11	4	7	0	0	100%	100%	0%
android.location.*	3	13	3	12	12	9	0	0	100%	100%	100%
android.os.*	2	46	3	60	60	49	0	0	100%	100%	100%
Total	51	309	40	556	540	440	2	2	99.63%	99.55%	96.36%

Table 2 lists the precision of each approach for each package. Both Modelgen specifications and manual models achieve a precision of over 99%.

Recall measures how many of our manual models are also discovered by Modelgen, and is calculated as:

$$Recall = |FModelgen \cap FManual|/|FManual|$$

As we can see from Table 2, Modelgen finds about 96% of our manual specifications. The specifications Modelgen misses were written to capture implicit flows, which is not surprising since Modelgen is designed to detect only explicit flows. A prime example of this limitation is the row corresponding to `android.util`, in which 7 of the 8 analyzed methods are part of the `android.util.Base64` class, which performs base64 encoding and decoding of byte buffers via table lookups, inducing implicit flows. Modelgen discovers four new correct flows for these methods, but misses all the implicit flows encoded in the manual models. The last remaining method in this package is a native method.

We can similarly calculate Modelgen's recall versus the total number of true positives from both techniques, as well as the analogous metric for Manual:

$$|FModelgen \cap TP|/|TP| = 97.12\%$$

$$|FManual \cap TP|/|TP| = 79.14\%$$

This shows that our technique discovers many additional correct specifications that our manual models missed.

We found two false positives in Modelgen, both in the same method. Two spurious flow annotations were generated, due to a hole in the trace which Modelgen processes under worst-case assumptions. Notably, we also found two errors in the manual models: one was a typo (*arg#2* $\rightarrow$ *arg#2* instead of *arg#2* $\rightarrow$ *return*) and the other was a reversed annotation (*arg#1* $\rightarrow$ *this* instead of *this* $\rightarrow$ *arg#1*).

Our current implementation of Modelgen failed to produce traces for a few methods that have manual annotations, listed under the column "Missing trace info." of Table 2. Reasons for missing traces include: the method for which we tried to generate a trace is a native method, the Android CTS lacks tests for the given method, or an error occurred while instrumenting the class under test or while running the tests. This last case often took the form of triggering internal responsiveness timers inside the Android OS, known as ANR (Application Not Responding) [23]—because our instrumentation results in a significant slowdown (about 20x), these timers are triggered more often than they would be in uninstrumented runs. Since capturing the traces is a one-time activity, this high overhead is otherwise acceptable.

These results suggest that Modelgen can be used to replace most of the effort involved in constructing manual models, since it reproduced almost all our manual flow annotations (96.38% recall) and produced many new correct annotations that our existing models lacked. Although our evaluation focuses on Java and Android, the results should generalize to any platform for which good test suites exist.

5.2 Whole-System Evaluation of STAMP and Modelgen

The STAMP static analysis component is a bounded context-sensitive, flow- and path-insensitive information flow analysis. A complete description of this system can be found in Section 4 of [18]. STAMP never analyzes platform code and treats platform methods for which it has no explicit model under best-case assumptions. That is, platform methods without models are assumed to induce no flows between their arguments or their return values[3].

To evaluate the usefulness of our specifications in a full static analysis, we ran STAMP under two configurations: base and augmented. In the base configuration, we used only the existing manually-written models. In the augmented configuration, we included (1) all source and sink annotations from the manual models (annotating sources and sinks is outside of the scope of Modelgen), (2) the Modelgen specifications generated in the experiment of Section 5.1, and (3) the existing manual models for those methods for which Modelgen did not construct any specifications (e.g. the `java.lang.*` classes). The base and augmented configurations included 1215 and 2274 flow annotations, respectively.

We compared the results of both configurations on 242 apps from the Google Play Store. These apps were randomly selected among those for which STAMP was able to run with a budget of 8GB of RAM and 1 hour time limit in both configurations. The average running time per app is around 7 minutes in either configuration.

STAMP finds a total of 746 (average 3.08 per app) and 986 (average 4.07) flows in the base and augmented configuration, respectively. The union of the flows discovered in both configurations is exactly 1000. In other words, STAMP finds 31% (254) new flows in the augmented configuration. Like most static analysis systems, STAMP can produce false positives, even when given sound models. Additionally, Modelgen may produce unsound models for some methods (recall the discussions in sections 4.4 and 4.5). Given this, we would like to know what proportion of these new flows are true positives. To estimate the true positive rate of the new flows, we took 10 random apps from the subset of our sample (109 of 242 apps) for which the augmented configuration finds any new flows. We manually inspected these apps and marked those flows for which we could find a feasible source-sink path, and for which control flow could reach such path, as true positives. Although this sort of inspection is always

[3]The alternative, analyzing under worst-case assumptions, produces an overwhelming number of false positives.

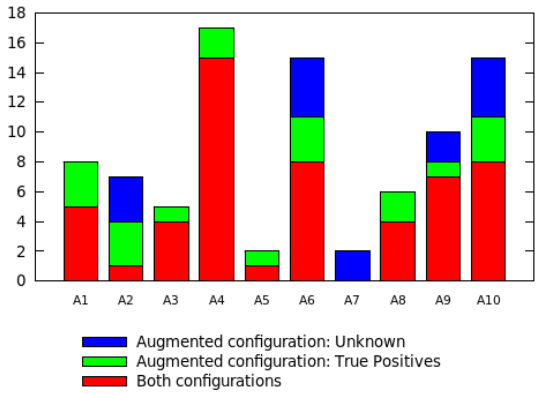


Figure 8: Flows found with and without Modelgen models

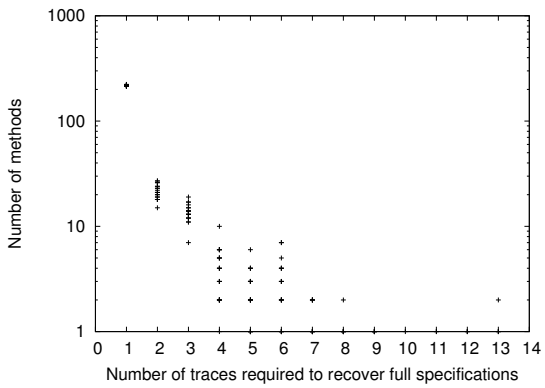


Figure 9: Specification Convergence

susceptible to human error, we tried to be conservative in declaring flows to be true positives. In most cases, the flows are contained in advertisement libraries and would trigger as soon as the app started or a particular view within the app was displayed to the user.

Figure 8 shows the results of our manual inspection. The flows labeled as "Augmented configuration: Unknown" are those for which we could not find a source-sink path, but are not necessarily false positives. The flows labeled "Augmented configuration: True Positives" represent a lower bound on the number of new true positives that STAMP finds in the augmented configuration. The lower portion of the bar corresponds to those flows found in both configurations, without attempting to distinguish whether they are false or true positives. For the 10 apps, the augmented configuration produces 64% more flows than the base configuration, and at least 55% of these new flows are true positives.

The recall of the augmented configuration, which is the percentage of all flows found in the base configuration that were also found in the augmented configuration, is 98.12%. A flow found in the base configuration could be missed in the augmented configuration if Modelgen infers a different specification for a method, which is relevant for the flow, than the manually-written model.

5.3 Controlling for Test Suite Quality

Specification mining based on concrete execution traces depends on having a representative set of tests for each method for which we want to infer specifications. One threat to the validity of our experiment is that it could be that our results are good only because the Android compatibility tests are unusually thorough. In this section we attempt to control for the quality of the test suite.

We measure how strongly our specification mining technique depends on the available tests by the number of method executions it needs to observe before it converges to the final specification. Intuitively, if few executions of a method are needed to converge to a suitable specification of the method's behavior, then our specification mining technique is more robust that if it requires many executions, and therefore many tests. Additionally, if a random small subset of the observed executions is enough for our technique to discover the same specification as the full set of executions, we can gain some confidence that observing additional executions won't dramatically alter the results of our specification mining.

We take all methods from Table 2 for which we are able to record traces and Modelgen produces non-empty specifications, which are 264 methods in total. We restrict ourselves to those methods, as opposed to the full set for which we have mined specifications, since we have examined them and found them to be correct during the comparison of Section 5.1. For each such method m, we consider the final specification produced by Modelgen (S_m) as well as the set \$ of specifications for each invocation subtrace of m. Starting with the empty specification we repeatedly add a random specification chosen from \$ until the model matches S_m, recording how many such subtrace specifications are used to recover S_m.

Figure 9 shows a log scale plot of the number of methods (vertical axis) that required n traces (horizontal axis) to recover the full specification over each of 20 trials. That is, we sampled the executions of each method to recover its specification and then counted the number of methods that needed one execution, the number that needed two, and so on, and then repeated this process 19 more times. The multiple points plotted for each number of executions give an idea of the variance due to the random choices of method executions to include in the specification.

It is also useful to consider aggregate statistics over all method specification inferences. In our experiment, 83.7% of the methods needed just one subtrace specification to recover the specification and no method required more than an average of 9 random subtrace specifications. The maximum number of subtraces needed to converge to a method specification (when taking the worst out of 20 iterations of the algorithm) was 13 for `java.util.Vector.setElementAt(Object, int)`. The average number of subtraces required to converge to a specification is 1.38. For comparison, the specifications evaluated in Section 5.1 were inferred using a median of 4 traces (the average, 207, is dominated by a few large outliers). We conclude that explicit information flow typically requires few observations to produce useful specifications.

6. RELATED WORK

Dynamic techniques for creating API specifications. Many schemes have been proposed for extracting different kinds of specifications of API methods or classes from traces of concrete executions. Closest to ours is work on pro-

ducing dynamic dependence summaries of methods as a way to improve the performance of whole-program dynamic dependence analysis [41]. Dependence analysis of some form is a prerequisite for explicit information flow analysis, since it involves determining which program values at which point in the execution are used to compute every new program value. Although our use case, and thus evaluation, is very different than that of [41], the specifications produced are somewhat related. In [41], a specification that a flows to b means literally that a location named by a is used to compute a value in a location named by b. In our framework, a specification that a flows to b means that some value reachable from a is used to compute some value reachable from b. Thus, the major difference is that we "lift" the heap-location level flows to abstract flows between method arguments and between arguments and the return value of the method, as described in 4.3. This lifting step requires additional infrastructure to maintain colors in the dynamic analysis, an issue that does not arise in dynamic dependence analysis. The added abstraction reduces the size of the summaries and allows us to generalize from fewer traces, though with a potential loss in precision, a trade-off which our results suggest is justified.

Using dynamic analysis to compute specifications consumed by static analysis has also been heavily explored. However, most such specifications focus on describing control-flow related properties of the code being modeled. A large body of work (e.g. [9, 2, 50, 34, 52, 51, 11, 20, 36, 35, 32]) constructs Finite State Automata encoding transitions between abstract program states. Other approaches focus on inferring program invariants from dynamic executions, such as method pre- and post-conditions (Daikon [40, 15, 16]), array invariants [39] and algebraic "axioms" [26]. Another relevant work infers static types for Ruby programs based on the observed run-time types over multiple executions [28]. Finally, program synthesis techniques have been used to construct simplified versions of API methods that agree with a set of given traces on their input and output pairs [42].

Dynamic taint tracking and related analyses. Dynamic taint tracking uses instrumentation and run-time monitoring to observe or confine the information flow of an application. Many schemes have been proposed for dynamic taint tracking [24, 10, 14, 5]. An exploration of the design space for such schemes appears in [45]. Dytan [10] is a generic framework capable of expressing various types of dynamic taint analyses. Our technique for modeling API methods is similar to dynamic taint tracking, and could in principle be reformulated to target Dytan or some similar general dynamic taint tracking framework. However, heap-reachability and all of our analysis would have to be performed online, as the program runs, which might exacerbate timing dependent issues with the Android platform.

As mentioned previously, dependence analysis is also related to information flow analysis, and the large body of work in dynamic dependence analysis is therefore also relevant to our own (e.g. [47, 27] and references therein).

Tools for Tracing Dynamic Executions. Query languages such as PTQL [21] and PQL [37] can be used to formulate questions about program executions in a high-level DSL, while tools like JavaMaC [30], Tracematches [1], Hawk [12] and JavaMOP [29] permit using automata and formal logics for the same purpose. Frameworks like RoadRunner [19] and Sofya [31] allow analyses to subscribe to a stream of events representing the program execution as it runs.

Static taint analysis. A number of static techniques and tools [13, 25, 38, 33, 48] have been developed for whole-application taint analysis. See [44] for a survey of work in this field. For applications that run inside complex application frameworks these analyses often must include some knowledge of the framework itself. F4F [46] is a scheme for encoding framework-specific knowledge in a way that can be processed by a general static analysis. In F4F, any models for framework methods must be written manually. Flowdroid [3] is a context-, flow- and object-sensitive static taint analysis system for Android applications, which can analyze Android platform code directly. By default, it uses models or 'shortcuts' for a few platform methods as a performance optimization and to deal with hard-to-analyze code. Flowdroid's shortcuts are also information-flow specifications of a slightly more restrictive form than that used by Modelgen. Thus, it seems likely the FlowDroid shortcuts could also be mined successfully from tests.

The technique presented in [8] uses static analysis to infer specifications of framework methods such that those specifications complete information flow paths from sources to sinks. Since this technique does not analyze the code of the framework methods, it often suggests spurious models, which must be filtered by a human analyst. This technique is complimentary to ours; DroidRecord can be used to validate models inferred by this technique. There has also been some previous work on identifying sources and sinks in the Android platform based on the information implicitly provided by permission checks inside API code [17, 4, 6] or by applying machine learning to some of the method's static features [43]. This work could be combined with our method for inferring specifications to enable fully automatic explicit information flow analysis (i.e., with no manual annotations).

7. CONCLUSIONS

We have described an effective technique for generating explicit information flow specifications for platform methods that outperforms manual flow annotations in practice. We presented Modelgen, an implementation of this technique for Java and the Android platform. Modelgen specifications are highly precise and provide high recall with respect to our existing manual models. They also allow our static analysis to find true flows it misses despite years of manual model construction effort. Furthermore, such specifications can be inferred from a relatively small set of execution traces.

In future work, we plan to explore ways in which our dynamic specification mining technique might be augmented with lightweight static analysis of the platform code or combined with specification mining techniques based on static analysis of the application code (e.g. [8]). We will also explore whether the small number of tests needed to converge to the correct specification can be generated automatically, such as by using dynamic symbolic execution or other input generation techniques.

8. ACKNOWLEDGMENTS

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

9. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

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

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

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

[6] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[7] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *SOAP*, pages 27–38, 2012.

[8] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[9] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 100(6):592–597, 1972.

[10] J. A. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[11] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, 2006.

[12] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

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

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[16] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

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

[18] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *FSE*, pages 576–587, 2014.

[19] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[20] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT*, pages 339–349, 2008.

[21] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402, 2005.

[22] Google. Compatibility test suite (Android) - https://source.Android.com/compatibility/cts-intro.html.

[23] Google. Keeping your app responsive - https://developer.Android.com/training/articles/perf-anr.html.

[24] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *ACSAC*, pages 303–311, 2005.

[25] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.

[26] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[27] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, pages 371–382, 2012.

[28] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *POPL*, pages 459–472, 2011.

[29] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE*, pages 1427–1430, 2012.

[30] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: A run-time assurance tool for Java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218–235, 2001.

[31] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *ICSE Companion*, pages 51–52, 2007.

[32] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *ICSE*, pages 179–182, 2010.

[33] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[34] D. Lo and S. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE*, pages 265–275, 2006.

[35] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE*, pages 345–354, 2009.

[36] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[37] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, pages 365–383, 2005.

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

[39] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693, 2012.

[40] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.

[41] V. K. Palepu, G. H. Xu, and J. A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *ASE*, pages 59–69, 2013.

[42] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury. Modeling software execution environment. In *WCRE*, pages 415–424, 2012.

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

[44] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SSP*, pages 317–331, 2010.

[46] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. In *OOPSLA*, pages 1053–1068, 2011.

[47] S. Tallam and R. Gupta. Unified control flow and data dependence traces. *ACM Transactions on Architecture and Code Optimization*, 4(3), 2007.

[48] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, pages 13–23, 1999.

[50] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS/ETAPS*, pages 461–476, 2005.

[51] T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE*, pages 835–838, 2006.

[52] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.