Mining Explicit Information Flow Specifications from Concrete Executions

Lazaro Clapp
Stanford University
lazaro@stanford.edu

Saswat Anand
Stanford University
saswat@cs.stanford.edu

Alex Aiken
Stanford University
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 Java and the Android platform. When compared to a set of manually written specifications for 309 methods across 51 classes, our technique is able to recover 96.38% of these manual specifications and produces many more correct annotations that our manual models missed (97.12% vs 79.50% precision). 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

Keywords

Dynamic analysis; specification mining; Android

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 existing frameworks (e.g., Android, Apache Struts, and Spring) that are large and complex. 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 known to be hard 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 themselves usually build on abstractions written in lower-level languages for which a comprehensive static analysis may not be available (e.g., Java’s native methods). Such foreign functions appear simply 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 their 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 specifications manually for a large framework is still laborious and susceptible to human error. To address the limitations of these two approaches for explicit information flow analysis, in this paper we present an approach in which the specifications are mined from concrete executions of a program.

Mining specifications from traces is not a novel idea. However, to the best of our knowledge, all existing approaches produce control-flow specifications (e.g., [3, 24, 22]) or general pre- and post-conditions on methods (e.g., Daikon[9]). Mining information flow specifications from program executions has not been previously explored.

We mine explicit 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, particularly memory operations. Using this trace, we reconstruct the view the method has of the structures in the heap reachable from the method’s arguments and perform a specialized form of dynamic taint tracking for values.
in those structures. We then lift the dynamic information flow to a static signature summarizing the dynamic flows between a method’s arguments or between an argument and the return value. Finally, we combine the flows mined from different executions of the same method to produce an overall specification for the method. As we will see, this requires careful handling of each operation in the trace and is further complicated by holes in the traces (portions of the trace that are missing because the corresponding code is uninstrumented; these are mostly due to native methods).

Because our static specifications are generated based on dynamic behavior they are underapproximations, which means our specifications can be unsound (i.e., our approach can miss some flows). To determine the quality of our results, we evaluate our approach by using it to automatically generate specifications of 309 methods across 51 classes of the Android platform. We compare these specifications to a set of manually written specifications produced over a period of two years. Our approach is able to independently discover 96.38% of the manual models (96.38% recall) and find many additional correct specifications missed by human model writers with high domain expertise, while maintaining a false positive rate below 0.36%. In addition, our inferred specifications uncovered 2 errors in the original manually written specifications. While our approach cannot guarantee soundness, it performs at a level comparable to that of a human writing annotations, while greatly reducing manual effort.

We also propose a measure of the difficulty of using a dynamic analysis to mine static specifications. Intuitively, if few executions of a method are needed to converge to a sound specification of the method’s behavior, then that specification mining problem is easier than one that requires many observations of method executions. In our experiments, methods require only 1.38 executions on average to infer a sound specification.

In this paper, we make the following contributions:

- A dynamic analysis technique to mine explicit information flow specifications of platform methods to be used by a static taint analysis tool.
- A practical implementation of our technique, applied to the Android platform.
- A study comparing the precision and recall of the specifications generated by our tool to that of a set of manually written models.
- A metric for measuring how difficult it is to mine static specifications from dynamic executions, based on the average number of method executions needed to obtain a correct static model. For explicit information flow we find the required number of tests is 1.38; since 1 is the minimum possible score we believe explicit information flow typically requires few observations to produce useful specifications.

We begin by giving a motivating example for the value of our technique (Section 2) and then describe the overall architecture of our implementation (Section 3). Our specification mining technique is then presented in detail (Section 4). Next, we describe an empirical evaluation of our approach, comparing our automatically generated specifications with a preexisting set of manual models (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, a hybrid static/dynamic program analysis tool for Android applications: The main 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.

STAMP performs whole-program analysis of the Android application code and any libraries bundled into its installer (.apk file). However, because of the challenges involved in analyzing large application frameworks, STAMP does not directly analyze the Android platform’s libraries. Unfortunately, this leaves a large portion of the classes and methods used by any Android application outside of the scope of STAMP’s taint analysis. 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 in a configuration file. These specifications can then be loaded by the static analysis and assumed to be an accurate representation of the corresponding methods with respect to the properties the analysis cares about. 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. Note that if $a$ is a reference, information accessible from $a$ includes all objects transitively reachable through other object references in fields.

This is the instance object for the modeled method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Flow Type</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>TelephonyManager.getLine1Number()</td>
<td>Null Path</td>
<td>return</td>
</tr>
<tr>
<td>CharBuffer.put(String,int)</td>
<td>null arg</td>
<td>arg#1 \rightarrow socket.write()</td>
</tr>
<tr>
<td>CharsetEncoder.encode(CharBuffer)</td>
<td>null arg</td>
<td>arg#1 \rightarrow return</td>
</tr>
<tr>
<td>SocketChannel.write(ByteBuffer)</td>
<td>null arg</td>
<td>arg#1 \rightarrow INTERNET</td>
</tr>
</tbody>
</table>

Figure 1: Leak phone number to Internet

<table>
<thead>
<tr>
<th>Method</th>
<th>Flow Type</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>TelephonyManager.getLine1Number()</td>
<td>Null Path</td>
<td>return</td>
</tr>
<tr>
<td>CharBuffer.put(String,int)</td>
<td>null arg</td>
<td>arg#1 \rightarrow socket.write()</td>
</tr>
<tr>
<td>CharsetEncoder.encode(CharBuffer)</td>
<td>null arg</td>
<td>arg#1 \rightarrow return</td>
</tr>
<tr>
<td>SocketChannel.write(ByteBuffer)</td>
<td>null arg</td>
<td>arg#1 \rightarrow INTERNET</td>
</tr>
</tbody>
</table>

Table 1: Specifications for platform methods
The bytecode is then instrumented to record the runtime values of the method’s parameters. In the case of reference types, Droidrecord uses the unique identifier returned by System.identityHashCode(Object) as the value.

When reading the trace, these values are plugged into the placeholder positions (denoted by $\_\_\_\_\_\_\_\_\$) above of the corresponding event template.

Note that 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 require no dynamic recording overhead whatsoever.

### 3.2 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 invocation of $m$; we write $(m : i)$ for the $i$th invocation of method $m$. Calls made by $(m : i)$

---

1. This is a custom form of Java bytecode that is executed by Android’s Dalvik virtual machine and is different from standard JVM bytecode.
2. Slightly edited for brevity.
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 inside the new thread. This separation may break some flows that involve operations of multiple threads and is a limitation of our current 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 \((m : i)\). As per the previous section, \(T_{(m, i)}\) 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, which we represent as 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 symbol \(\circ\) 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 the grammar of a three-address bytecode representing Java operations. However, it represents not static program structure, but the sequence of operations occurring during a concrete program run. As such, it has 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 operation events, but the notion of a conditional jump based on their value is meaningless for our event trace.

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 \(a.i\), which is consistent with field names being constants because we know the value of \(i\) for each such event).

3. For each method call event \(x = m_1(y)\) in \(T_{(m, i)}\) there is a unique invocation subtrace of the form \(T_{(m_1, j)} = \text{fun}(\gamma)\{\text{var } \tau; e_f\} \gamma\) where \(e_f\) is a return or throw event and \(\tau\) 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.\(^3\)

Finally, we avoid modeling static class fields explicitly by representing them as fields of a singleton object associated with each class.

4.2 Natural Semantics of a Subtrace

\(^3\)An invocation subtrace might contain any number of additional throw events, each immediately followed by a catch event in the same subtrace if the exception is caught in the same method that raises it.
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 quite identical to the semantics of Java bytecode, with differences arising from the fact that subtrace semantics represent a single execution path through the code.

During subtrace evaluation, an environment $\rho$ maps variable names to values, which is analogous to registers and the stack holding local variables during concrete execution. 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 $\rho$, 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. The operator $\downarrow$ 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 $\downarrow$ for different events and their necessary pre-conditions. 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 $x.f$:

\[
\begin{align*}
    t & := y = x.f; z = 1; w = y + z; x.f = w \\
    \text{Assuming } x & \text{ contains the address } a \text{ (i.e., } \rho(x) = a) \text{ of heap record } r = \{ f : 0 \} \text{ (i.e., } h(a) = r), \text{ LOAD gives us:} \\
    \langle h, \rho, y = x.f \rangle & \downarrow \langle h, \rho[y \rightarrow 0] \rangle \\
\end{align*}
\]

Then, applying LIT, BINOP and STORE, respectively, we get:

\[
\begin{align*}
    \langle h, \rho[y \rightarrow 0], z = 1 \rangle & \downarrow \langle h, \rho[y \rightarrow 0]; z \rightarrow 1 \rangle \\
    \langle h, \rho[y \rightarrow 0]; z \rightarrow 1, w = y + z \rangle & \downarrow \langle h, \rho[y \rightarrow 0]; z \rightarrow 1, w \rightarrow 1 \rangle \\
    \langle h, \rho[\ldots; w \rightarrow 1], x.f = w \rangle & \downarrow \langle h[a \rightarrow \{ f : 1 \}], \rho[\ldots; w \rightarrow 1] \rangle \\
\end{align*}
\]

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

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

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

The modeling semantics augment the natural semantics by associating colors with every heap location and primitive value. For subtrace $T_{m \rightarrow i}$, each argument to $m$ is initially assigned a single unique color. The execution of $T_{m \rightarrow i}$ 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 $a$.

### Figure 5: Modeling semantics

**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 label (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 $\rho$ 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 \( \mathcal{L} \) 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 \( C : VLoc \rightarrow 2^{\text{Color}} \) from virtual locations to sets of colors. The modeling semantics also uses \( G : \{\{\text{Color, Color}\}\} \), which is a relation on colors or, equivalently, a directed graph in which nodes are colors, and \( D : (\text{Address, Field}) \rightarrow \text{Boolean} \), 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 currently executed subtrace. We explain the use of \( G \) and \( D \) 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 \text{mLoad} and \text{mStore} preserve Invariants II and III, respectively.

Rule \text{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 \text{mNew}, which models new object creation, is similar. Rule \text{mAssign} models an assignment \( x = y \) where \( x \) and \( y \) are both variables in \( \rho \) and does not create a new location, but instead updates \( \mathcal{L}(x) \) to be the location of \( y \), indicating that they are the same value, again preserving Invariant I.

Rule \text{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 \( C(l_3) \) becomes the union of \( C(l_1) \) and \( C(l_2) \).

Rules \text{mLoad} and \text{mStore} deal with field locations. The virtual location of field \( a.f \) (denoted \( \mathcal{L}(a.f) \)) 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 \( \rho \) and the name of \( f \), if \( f \) is of primitive type.

Rule \text{mLoad} models load events of the form \( x = y.f \) by assigning the location \( l_2 = \mathcal{L}(y.f) \) 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 \( x.f \) has been written previously, then \( C(l_2) \) becomes the union of \( C(l_1) \) and \( C(l_2) \), indicating that \( l_2 \) now has the colors of all of its previous accesses plus a possibly new set of colors \( C(l_1) \). 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 \( D(\rho(y), f) = \text{True} \). In this case it is no longer true that \( \mathcal{L}(y.f) \) was reachable from \( \mathcal{L}(y) \) on method entry and so it is not necessary to propagate the color of \( \mathcal{L}(y) \) to \( \mathcal{L}(y.f) \) 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 6 shows the effect of a single load operation from an argument to \( m \), while Figure 7 depicts the coloring of a set of the heap locations after multiple load events.

Rule \text{mStore} models store events of the form \( x.f = y \). The rule updates \( D(\rho(x), f) = \text{True} \) since it writes to \( x.f \). We could satisfy Invariant III by implementing \text{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 \( G \) to record an edge from each color \( c_1 \) of \( \mathcal{L}(y) \) to each color \( c_2 \) of \( \mathcal{L}(x.f) \) with the following meaning: \( c_1 \rightarrow c_2 \in G \) means every virtual location with color \( c_2 \) has color \( c_1 \) as well. Figure 8 depicts the results of a store operation, while figure 9 depicts how \( G \) serves to associate two colored heap subgraphs.

Rule \text{mSVN} implements standard method call semantics, mapping the virtual locations of arguments and the return value between caller and callee. Rule \text{mSEQ} is the same as \text{SEQ} in the natural semantics but adds \( \mathcal{L}, C, G \) and \( D \) where needed.

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 \( G \)). 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 \( a_k \rightarrow r \) \((a_k \rightarrow a_j)\) to our model.

### 4.4 Combining Specifications

After the invocation subtrace \( T_{(m,i)} \) has been interpreted according to modeling semantics, we use the criteria just outlined to produce a series of argument-to-argument and argument-to-return flows for \((m:i)\). Because our analysis for each invocation produces an underapproximation based on a single execution, we combine the results from different invocations of \( m \) by taking an upper bound on the set of flows discovered for every execution, which is simply the union of the results of \((m:i)\) for every \( i \).

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

### 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, even a dynamic analysis faces challenges recording the execution of every method in the Android platform. In particular, any technique based on Java bytecode instrumentation cannot capture the behavior of native methods and system calls. Furthermore, because our inserted recorder class is itself written in Java, we must exclude from instrumentation some Java classes it depends upon to avoid an infinite regress.

Thus, traces are not always full traces but represent only a part of a program’s execution. We need to deal with two separate problems during event interpretation: (1) How can we detect that a trace has called an uninstrumented method? (2) How are the uninstrumented calls handled by Modelgen?

For the second 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 \text{System.arraycopy} and \text{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.

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 can (and often do) 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 the issue of telling whether the method we see start after a call instruction is the instruction’s callee or if the callee itself is uninstrumented and we are seeing a method called from inside of code we cannot observe.

Our solution for detecting holes in the trace due to invoking uninstrumented code is to record the Java stack height 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 the stack height to determine when we have called into uninstrumented code.

### 5. Evaluation

We evaluate Modelgen by using it to generate information flow specifications for a significant portion of the Android platform, including 309 methods across 51 classes, for which we have manually written specifications. These specifications were originally written over a period of two years as part of building the STAMP static analysis tool for Android. Generating these models was a non-trivial task, as it required running the STAMP tool 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 fi-
nally writing a model for each missing method. Modelgen greatly reduces this manual effort.

Generated models are only as good as the traces the tool can analyze for each method being modeled. For our evaluation, we obtained traces by running tests from the Android Compatibility Test Suite (CTS)[14] for each class for which we had manual models. The Android CTS is designed to ensure compatibility between multiple implementations and variations of the Android platform and our positive results are due at least in part to the fact that CTS is a high quality set of 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., \( \text{arg#}X \to \text{return} \)) included either in our manual specifications or generated by Modelgen. We then list separately the flows discovered by Modelgen and those in our manual specifications. Note that we consider only those flows in methods for which we have manual specifications, even though in the process of obtaining those, Modelgen constructs specifications for 1,815 methods in total, including those methods called from the methods being modeled or the tests used to exercise them.

We evaluate Modelgen using two metrics: precision and recall. Precision is a measure of soundness: the percentage of all possible flows through the method are discovered by Modelgen. Given that neither Modelgen nor our manual models are fully precise, we approximate precision by comparing the flows discovered by each approach to the union of the flows discovered by both approaches. Formally, let \( \text{FModelgen} \) be the set of flows discovered by Modelgen and \( \text{FManual} \) the set of flows in our manual models. The precision of Modelgen is:

\[
P_{\text{Modelgen}} = \frac{|\text{FModelgen}|}{|\text{FModelgen} \cup \text{FManual}|}
\]

And the precision of the manual models is:

\[
P_{\text{Manual}} = \frac{|\text{FManual}|}{|\text{FModelgen} \cup \text{FManual}|}
\]

Table 2 lists the precision of each approach for each package. Overall, Modelgen’s precision is above 97%, whereas the manual models are about 80% precise by comparison. Again, a caveat is that these numbers represent the relative precision of both approaches, as it is possible that there exist flows through platform methods that are absent in both our manual models and in Modelgen’s generated specifications.

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

\[
\text{Recall} = 1 - \frac{|\text{FModelgen} \cup \text{FManual}| - |\text{FModelgen}|}{|\text{FManual}|}
\]

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.

When collecting our results and contrasting the manual models to Modelgen’s specifications, we found two false positives in Modelgen, both in the same method and due to an unexpected hole in the trace resulting from a bug in how our current implementation handles inner classes. Modelgen detected the hole in the trace and processed it under worst-case assumptions, resulting in two spurious flows. Fixing the bug will remove these flows.

Notably, we also found two errors in the manual models: one was a typo (\( \text{arg#}2 \to \text{arg#}2 \) instead of \( \text{arg#}2 \to \text{return} \)) and the other was a reversed annotation (\( \text{arg#}1 \to \text{this} \) instead of \( \text{this} \to \text{arg#}1 \)).

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)[15]—because our instrumentation results in a significant slowdown, these timers are triggered more often than they would be in uninstrumented runs.

Given these results, we are confident that Modelgen can be used to replace almost all manual information flow models as it managed to reproduce almost all our manual flow annotations (96.38% recall) and produced many more correct annotations that our manual models missed (97.12% vs 79.50% precision), while significantly reducing manual effort. Although our evaluation focuses on Java and the Android platform, the results should generalize to any platform for which good test suites are available.

5.1 Controlling for Testsuite Quality

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

We measure how strongly a particular specification mining technique depends on the available tests by the number of method executions it needs to observe before it converges to a sound specification. Intuitively, if few executions of a method are needed to converge to a sound specification of the method’s behavior, then that specification mining problem is easier than one that requires many executions, and therefore many test cases.

We take all methods from the previous experiment for which we are able to record traces and Modelgen produces non-empty specifications, which are 264 methods in total. For each such method \( m \), we consider the final specification produced by modelgen \( (S_m) \) as well as the set \( S \) of specifications for each invocation subtrace of \( m \). Starting with the empty specification we repeatedly add a random specification chosen from \( S \) until the model matches \( S_m \), recording how many such subtrace specifications are used to recover \( S_m \).

We repeat this procedure for all methods. Figure 10 shows a log scale plot of the number of methods (y axis) that required \( n \) traces (x 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 gives 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. We conclude that explicit information flow typically requires few observations to produce useful specifications.

6. RELATED WORK

Modelgen represents a solution to a common limitation of static program analysis, utilizing techniques related to dynamic taint tracking to provide specifications of API methods. In addition, the Droidrecord toolchain is related to previous tools for tracing dynamic executions. In this section, we discuss some of the most closely-related work in each of these areas.

There has also been some previous work on identifying sources and sinks for sensitive information in the Android platform, based on the information implicitly provided by permission checks inside API code[11]. This work could be combined with our method for inferring explicit information flow specifications to enable fully automatic explicit information flow analysis (i.e., with no manual annotations).

6.1 Static taint analysis

A number of static techniques and tools have been and continue to be developed for analyzing whole applications, including static taint analyses (e.g., [7, 17, 27, 23, 36], Sabelfeld and Myers[32] provide a survey of work before 2002). For applications that run inside complex application frameworks these analyses often must include some knowledge of the framework itself (TAJ[36], for example, includes knowledge about the Struts and EJB web frameworks).

Sridharan et al. propose F4F[34], a general scheme for encoding framework-specific knowledge in a way that can be processed by a general static analysis tool. They develop generators that, for a particular framework and application, produce a description of the framework code as used by the application, decoupling framework-modeling from analysis implementation. Their scheme can encode more about the modeled framework than information flow specifications, but any models for framework methods must be written manually. One could imagine a combined approach in which F4F generators use Modelgen to automatically produce specifications of most framework methods when the client is an static taint analysis.

6.2 Dynamic taint tracking

Dynamic taint tracking uses instrumentation and runtime monitoring to observe and possibly also confine the information flow of an application. Many schemes have been proposed for dynamic taint tracking [16, 5, 8, 1]. Taint tracking schemes differ in the following dimensions: the type and expressiveness of taint annotations used; how sources, sinks and sanitizers are chosen; how taint is propagated by each program operation; and how to deal with implicit taint propagation through program control dependencies. Schwartz et al.[33] provide an exploration of this design space as well as an overview of previous work, while Clause et al.[5] propose a generic framework (Dytan) capable of expressing many different types of dynamic taint analyses.

Our technique for modeling API methods is similar to dynamic taint tracking. We could redefine Modelgen as a dynamic explicit taint tracking system that assigns different taint labels to each argument of the method being modeled and propagates taint from an object to its fields on every
load operation (a non-standard taint propagation function). After each modeled method execution, the system would then perform dynamic heap-reachability to determine if the taint of any argument is reachable from any other argument or the return value by following field references. This approach could be implemented on top of 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 would exacerbate timing dependent issues with the Android platform (recall the discussion in Section 5). We would also still need to aggregate data over multiple runs to gain confidence in the generality of the models.

6.3 Dynamic techniques for creating API specifications

Extracting specifications of API methods or classes from traces of concrete executions is not a new idea; different schemes have been proposed for generating different kinds of models. However, unlike our information flow specifications, most such specifications focus on describing control-flow related properties of the code being modeled.

An et al.\cite{10} show an approach to type inference in the Ruby programming language that makes use of observing run-time types over multiple execution traces and performing constraint-solving to infer general static types. Type models can be considered as a very coarse form of data flow models.

Multiple techniques have been proposed to generate Finite State Automata (FSA) for approximating programs (for example, \cite{3, 24, 22}). The general approach is to take multiple traces, each consisting of a sequence of method invocations and construct an FSA where state transitions are labeled by method names in such a way that each trace used to generate the model would be accepted by the FSA. In general, approaches start by making a tree from all traces and then progressively merging states based on different criteria. Precision depends on how states are merged. Usually, at least the kTail criterion\cite{3} must be met to merge two states of the tree: both states must be followed by invocations of the same k methods along at least one path through the graph of the partially-constructed automata. In addition to the kTail criterion, merging states can be restricted to cases where the merging respects inferred temporal properties\cite{24} or where both states have the same inferred constraints on program values\cite{22}, among others.

Daikon\cite{29, 9} is a system that infers program invariants from dynamic executions, including method pre- and post-conditions and object invariants. Daikon-generated invariants provide a model of the behavior of complex code and have been used for vulnerability detection\cite{10}. Other approaches to generating invariants include work by Nguyen et al. on polynomial and array invariants\cite{28} as well as work by Henkel et al. on algebraic “axioms”\cite{18} (particularly, equivalences between abstracted program terms). An empirical study of the reliability of invariant inference by Polikarpova et al.\cite{30} has found that automatic systems (Daikon) tend to produce many more correct invariants than are usually included in manual programmer annotations, even considering only programs where such annotations exist. On the flip side, the same study also shows such systems miss about 40% of manually written annotations and that up to one third of automatically generated invariants can be incorrect or irrelevant. Because Modelgen specifications are of a more constrained form that depends only on the flow of values rather than arbitrary relations we achieve a comparatively higher precision and recall.

Both FSA-based and invariant-based approaches generate models of the control properties of API classes and methods, but are not designed to express the data flow dependencies needed for static taint analysis.

Finally, Qi et al.\cite{31} use program synthesis techniques to construct simplified versions of API methods that agree with a set of given traces on their input and output pairs.

6.4 Tools for Tracing Dynamic Executions

The RoadRunner framework\cite{12} is perhaps the most similar to our approach. RoadRunner produces a stream of events that are processed by client dynamic analysis tools written against the framework’s API. Droidrecord follows a similar model, but currently Modelgen is our only client tool. Sofya\cite{21, 20} is another dynamic analysis framework that works by dispatching, filtering and consuming events. RoadRunner and Sofya analyses are online, while Droidrecord works offline. Offline analysis allows us to execute the instrumented program inside an Android emulator or device while performing our analysis on a larger machine, as well as more easily aggregating data over multiple traces.

PTQL\cite{13} is a relational query language over program traces. The associated PARTIQLE compiler instruments programs to evaluate PTQL queries online. Because the query is known before instrumentation, in-memory bookkeeping structures can be optimized for a particular query. PQL\cite{26} is another query language for analyzing programs, with an associated resolver that has both a static and a dynamic component. Query languages over program traces work at a higher-level of abstraction than stream-of-events processors, trading some generality for ease of writing client dynamic analyses.

Tools like ATOM\cite{35} or Pin\cite{25}, on the other hand, operate at a lower-level: analyses are written by specifying method calls to be replaced as well as a replacement method which is often either a logger method or part of an online analysis. Libraries like Soot\cite{37}, ASM\cite{4} or BCEL\cite{6} provide a general interface for inspecting and editing Java bytecode (or DEX), providing the basis for most dynamic tracing tools. Droidrecord is built directly on top of Soot.

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 have also presented an implementation of this technique and showed that, for 309 methods of the Android platform, we can achieve 96.38% recall of our manual models and also produce many additional correct flows for those same methods that two years of manual effort missed. We have also shown that explicit information flow models can be generated even with a modest set of execution traces, requiring 1.38 observations of a method’s execution on average in our experiments.

References

\cite{1} Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In POPL, pages


[33] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.


