2016 IEEE European Symposium on Security and Privacy

HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving

Stefano Calzavara
Università Ca' Foscari Venezia
calzavara@dais.unive.it

Ilya Grishchenko
CISPA, Saarland University
grishchenko@cs.uni-saarland.de

Matteo Maffei
CISPA, Saarland University
maffei@cs.uni-saarland.de

***Abstract*—We present HornDroid, a new tool for the static analysis of information flow properties in Android applications. The core idea underlying HornDroid is to use Horn clauses for soundly abstracting the semantics of Android applications and to express security properties as a set of proof obligations that are automatically discharged by an off-the-shelf SMT solver. This approach makes it possible to fine-tune the analysis in order to achieve a high degree of precision while still using off-the-shelf verification tools, thereby leveraging the recent advances in this field. As a matter of fact, HornDroid outperforms state-of-the-art Android static analysis tools on benchmarks proposed by the community. Moreover, HornDroid is the first static analysis tool for Android to come with a formal proof of soundness, which covers the core of the analysis technique: besides yielding correctness assurances, this proof allowed us to identify some critical corner-cases that affect the soundness guarantees provided by some of the previous static analysis tools for Android.**

1. Introduction

The Android platform is by far the most popular choice for mobile devices nowadays, with billions of applications routinely installed on a massive number of different phones and tablets. Given this increasing popularity, personal information and other sensitive data stored on Android devices constitute an attractive target for breaching users' privacy at scale by malicious application developers. Information flow control frameworks for Android have thus emerged as a prominent research direction, with several different proposals spanning from dynamic analysis [11], [19], [34], [17] to static analysis [39], [38], [24], [14], [20], [23], [2], [15], [21]. Static analysis is particularly appealing for information flow control, given its ability to provide full coverage of all the possible execution paths and the possibility to be employed in the vetting phase, i.e., before the application is uploaded onto the Google Play store.

The most recent works in this area [2], [15], [21], [36] are impressive in their efforts to support a significant fragment of the Android platform. Most of them leverage existing static analysers by encoding Android applications in a suitable format, e.g., FlowDroid [2], DroidSafe [15], and IccTA [21] use Soot [35], while CHEX [23] uses Wala [13]. Observing that existing static analysers come with intrinsic limitations that limit the precision of the analysis (e.g., Soot and Wala do not calculate all objects' points-to information in a both flow- and context-sensitive way), Amandroid [36] relies on a dedicated data-flow analysis algorithm.

Despite all this progress and sophisticated machinery, none of these tools achieves a satisfactory degree of soundness: even on benchmarks written by the community and consisting of simple programs (i.e., Droidbench [2]), for which the ground truth is known, all existing tools miss several malicious leaks (false negatives). This, along with the fact that none of these tools comes with a formal model or soundness proof, makes one wonder how accurately these analyses capture all the subtleties of the Android execution model, which is far from being trivial [26], and to which extent their results are reliable on real-life applications, for which the ground truth is not known.

Furthermore, the lack of precise and fully documented analysis definitions complicates the comparison between different approaches: for instance, there is no universal agreement on a single notion of object-sensitivity [28], though object-sensitivity has been recognized as crucial to support a precise analysis of real-world Android applications [2]. Hence, at the time of writing, the only way to grasp the relative strengths and weaknesses of different static analysis tools for Android applications relies on an hands-on testing on some common benchmark and a source code inspection of their implementation.

Our contributions. We present a fresh approach to the static analysis of Android applications, i.e., a data-flow analysis based on *Horn clause resolution* [5]. The core idea is to soundly abstract the semantics of Android applications into a set of Horn clauses and to formulate security properties as a set of proof obligations, which can be automatically discharged by off-the-shelf SMT solvers. In particular:

- We prove the soundness of our analysis against a rigorous formal model of a large fragment of the Android ecosystem, covering Dalvik bytecode, the event-driven nature of the activity lifecycle, and inter-component communication. While elaborating the proof, we identified a few critical corner-cases that affect the soundness guarantees provided by some of previous static analysis tools for Android.

978-1-5090-1752-2/16 $31.00 © 2016 IEEE
DOI 10.1109/EuroSP.2016.16

We believe that this formal model may constitute a foundational framework, serving as a starting and comparison point for future work in the field;

- We fine-tune the Horn clause generation in order to optimize precision and efficiency, while retaining soundness. Being a data-flow analysis rather than a pure taint analysis, our solution statically approximates run-time values, in contrast to most of the previous works in the field [2], [15], [21]. This boosts the precision of the analysis: for instance, it makes it possible to statically determine whether a conditional branch will never be taken at runtime and ignore it. A salient feature of our approach is the usage of SMT solving to discharge proof obligations. From an engineering point of view, this allows one to fine-tune the analysis while still building on off-the-shelf verification tools, thereby leveraging the continuous advances in this field.
- We develop a tool, called HornDroid, which implements the analysis described in the formal model and complements it in order to support additional Android features, such as reflection, exceptions, and threading. HornDroid automatically generates Horn clauses from the application bytecode and relies on the state-of-the-art SMT solver Z3 [9] for discharging proof obligations[1].
- We conduct a performance evaluation on DroidBench, a collection of 120 programs written by the community, comparing HornDroid with IccTA [21] (an extension of FlowDroid [2] to inter-component communication), Amandroid [36] and DroidSafe [15]. HornDroid outperforms the competitors in terms of sensitivity (i.e., soundness) and performance, while retaining a high specificity (i.e., precision): HornDroid is the only tool that identifies all the explicit information flows, it exhibits just one more false positive than Amandroid (the most accurate tool), and it is one order of magnitude faster than IccTA and AmanDroid, and two orders of magnitude faster than DroidSafe. Furthermore, we show that HornDroid scales well to real-life applications from Google Play by a comparative evaluation on the two largest applications from the Google Play Top 30, i.e., Candy Crash Soda Saga and Facebook, which pose significant problems to existing tools.

The tool as well as an extended version of this paper with a complete formalisation and proofs are available online [7].

2. Design and Motivations

Static information flow control for Android applications is a mature research area nowadays [39], [38], [24], [14], [20], [23], with IccTA [21] (an extension of FlowDroid [2] to inter-component communication), AmanDroid [36] and DroidSafe [15] representing the state-of-the-art in this field. Although all these proposals are impressive projects, which significantly advanced the area of information flow control for Android applications, they all have important limitations, motivating the need for novel research proposals.

We make this need apparent by focussing on two important design choices where these tools differ: *value-sensitivity* and *flow-sensitivity*. It is instructive to highlight the import of these choices in terms of both the soundness and the precision of the resulting static analysis. Table 1 summarizes the design choices of the tools we consider, including ours.

TABLE 1 Design Choices for Static Analysis Tools

	IccTA	*AD*	*DS*	*HD*
Value-sensitivity	no	yes	no	yes
Flow-sensitivity	yes	yes	no	partial

2.1. Value-sensitivity

Value-sensitivity is the ability of a static analysis to approximate runtime values and use this information to improve precision, e.g., by skipping unreachable program branches [25]. Concretely, consider the following code:

```
int x = 0;
for (int y = 0; y <= 10; y++) { x++; }
TelephonyManager tm = ...
String imei = tm.getDeviceId();
if (x == 0) { leak(imei); }
```

Though this code is perfectly safe, all the existing tools (IccTA, AmanDroid and DroidSafe) will identify it as leaky. IccTA and DroidSafe conservatively assume all the program points to be potentially reachable. Even AmanDroid raises a false alarm for this code, though it internally implements a dedicated data-flow analysis [36].

Besides this simple example, there are many reasons why real-world static analysis tools for Android applications should be value-sensitive to be practically useful. First, several features of Java and the Android APIs, most notably *reflection* and *dictionary-like* containers, e.g., intents and bundles, need value-sensitivity to be analysed precisely. Second, the loss of precision entailed by value-insensitivity may creep and interact badly with other desirable features of the static analysis, e.g., *context-sensitivity*, which has been deemed as crucial by previous studies [2], [15].

Context-sensitivity is the ability of the analysis to compute different static approximations upon different method calls. To understand why the benefits of context-sensitivity can be voided by value-insensitivity, consider the following method, where we assume to know a valid upper bound for the GPS location values:

```
void m (double x, double y) {
  if (x <= MAX_X && y <= MAX_Y)
    ...
```

1. It would be possible to discharge proof obligations by using at the same time different SMT solvers, since each of them might perform best on a certain class of queries. We did not find it necessary in our current experiments, but we plan to implement this feature in the future.

```
  else
    leak("Invalid location:" + x + y);
}
```

Context-insensitive static analyses would detect a dangerous information flow whenever the method `m` is invoked at two different program points and one of these invocations provides the location of the device in the actual parameters, while the other one provides an invalid location. The reason is that the method `m` would be analysed only once, hence the static analysis would detect that both public and confidential values may reach a sink. Conversely, a context-sensitive analysis potentially has the ability to discriminate between the two methods invocations and be precise, but the lack of value-sensitivity would necessarily lead to the detection of a non-existent information flow.

Finally, it is worth noticing that value-sensitivity is crucial to support security-relevant, value-dependent security queries (e.g., "Is the credit card number sent on HTTP rather than on HTTPS?" or "Is the picture actually uploaded on Facebook, as opposed to some other untrusted website?").

2.2. Flow-sensitivity

Flow-sensitivity is the ability of a static analysis to take the order of statements into account and compute different approximations at different program points [25]. To understand its importance, consider the following code:

```
TelephonyManager tm = ...
String imei = tm.getDeviceId();
imei = new String("empty");
leak(imei);
```

Though the code above is safe, the flow-insensitive analysis implemented in DroidSafe will identify it as leaky, since the variable `imei` does contain a secret information at some program point. Conversely, both FlowDroid and AmanDroid will correctly deem the program as safe.

Clearly, it is tempting to target a flow-sensitive information flow analysis tool to achieve a higher level of precision, but, as pointed out by the authors of DroidSafe [15], flow-sensitivity is very hard to get right for Android applications, due to their massive use of asynchronous callbacks. Both FlowDroid and AmanDroid suggest to tackle this problem by introducing a *dummy main method* emulating each possible interleaving of the callbacks defining the application lifecycle. Unfortunately, it is difficult to ensure that the dummy main method construction is accurate and comprehensive, which leads to missing malicious information flows [15].

2.3. HornDroid

Our tool, HornDroid, targets a *sound* and *practical* information flow analysis for Android applications. We report on the design choices we made to hit the sweet spot between these two potentially conflicting requirements.

HornDroid implements a *value-sensitive* information flow analysis. As anticipated, value-sensitivity is crucial to support a practically useful analysis of real-world applications. The analysis implemented in HornDroid is reminiscent of *abstract interpretation*, whereby the operational semantics of a program is over-approximated by a computable abstract semantics. As it is customary for abstract interpretation, the design of the analysis is parametric with respect to the choice of a set of *abstract domains*, defining how runtime values are statically approximated: one can then fine-tune the precision of the analysis by testing different abstract domains. To ensure the scalability of our value-sensitive analysis, the abstract semantics implemented in HornDroid is based on *Horn clauses*, whose efficient resolution is supported by state-of-the-art SMT solvers [5].

HornDroid performs a *flow-sensitive* information flow analysis on the registers employed by the Dalvik Virtual Machine, while implementing a *flow-insensitive* analysis for callback methods and heap locations. This is crucial to preserve the precision of the analysis, without sacrificing soundness. We already mentioned that previous studies highlighted that flow-sensitive analyses may easily produce unsound results, due to the challenges of predicting all the possible orderings of the Android callbacks [15]. Moreover, while carrying out the soundness proof for HornDroid, we realized that *static fields* are particularly delicate to treat in a flow-sensitive fashion. The reason is that static fields provide a way to implement a shared memory between otherwise memory-isolated components running in the same application. Given that the execution order of different Android components is extremely hard to predict, due to their callback-driven nature, it turns out that flow-insensitivity for static fields is in practice needed for soundness. Indeed, since static fields can be used to exchange pointers to heap locations, a sound flow-sensitive analysis for heap locations is in general hard to achieve. Our soundness proof, instead, confirms that flow-sensitivity can be implemented for the registers employed by the Dalvik Virtual Machine without missing any malicious information flow.

3. Operational Semantics

We base our technical development on μ-Dalvik$_A$, a formal model of the Android semantics obtained by extending the μ-Dalvik calculus [18] with a complete characterisation of the activity-specific aspects of the Android platform [26].

3.1. Background and Scope

Android applications are developed in Java and then compiled to a custom bytecode format called *Dalvik*, which is run by the Dalvik Virtual Machine (DVM). Unlike Java VMs, which are stack machines, the DVM adopts a register-based architecture. Android applications are different from standard Java programs, since they are structured in *components* of four different types: activities, services, content providers and broadcast receivers [31]. These components represent distinct entry points of the Android framework into the application. Hence, the operational behaviour of an Android application does not simply amount to the

sequential execution of its bytecode implementation, but it heavily relies on callbacks from the Android framework, as a reaction to user inputs, system events, or inter-component communication. Different Android components, either in the same application or from different applications, can communicate by exchanging *intents*, i.e., dictionary-like messaging objects. Intents may be sent either to a specific component (*explicit* intents) or to any component which declares the will of providing a given functionality (*implicit* intents).

In our formal model we consider Android applications consisting of activities only. We focus on activities, since a tested semantics is available for them and because they exhibit the most complicated life-cycle among all the component types [26]. Also, we only model intra-application communication based on explicit intents: implicit intents are mostly, if not only, used for inter-application messages. As we discuss in Section 5, μ-Dalvik$_A$ does not cover all the Android features supported by HornDroid: the purpose of μ-Dalvik$_A$ is ensuring that the design principles at the core of HornDroid are sound and that most of the Android-specific subtleties have been taken into due account.

3.2. Syntax

We write $(r_i)^{i \leq n}$ for the sequence $r_1, \ldots, r_n$. If the length of the sequence is immaterial, we just write r^* and we still let r_j stand for its j-th element. We represent the empty sequence with a dot $(\cdot)$. We let $r^*[j \mapsto r']$ be the sequence obtained from r^* by replacing its j-th element with r'. A *partial map* is a sequence of key-value bindings $(k_i \mapsto v_i)^*$, where all the keys k_i are pairwise distinct. Given a partial map M, let $dom(M)$ stand for the set of its keys and let $M(k) = v$ whenever the binding $k \mapsto v$ occurs in M. We identify partial maps which are identical up to the order of their key-value bindings.

Table 2 provides the syntax of μ-Dalvik$_A$ programs. It is an extension of the original μ-Dalvik syntax [18] with a few additional statements modelling method calls to Android APIs used for inter-component communication.

A μ-Dalvik$_A$ program P is a sequence of classes cls^*, which in turn are defined by a class name c, a direct superclass c', some implemented interfaces c^*, and a number of fields fld^* and methods mtd^*. Field declarations $f : \tau$ include the field name f and its type τ, while method declarations $m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$ include the method name m, the argument types τ^*, the return type τ, and the method body st^*. The annotation n on top of the arrow tracks the number of local registers used by the method, which is statically known in Dalvik.

We briefly discuss below the statements of the language. An unconditional branch `goto` pc sets the program counter to pc. The statement `move` lhs rhs moves the right-hand side rhs into the left-hand side lhs: here, lhs may be a register r, an array cell $r_1[r_2]$, an object field $r.f$, or a static field $c.f$; rhs may be any of these elements or a constant. A conditional branch $\texttt{if}_{\lessgtr}\ r_1\ r_2$ `then` pc compares the content of two registers r_1 and r_2 using the comparison operator $\lessgtr$ and sets the program counter to pc if the check is successful, otherwise it moves to the next instruction. We then have unary and binary operations, represented by $\texttt{unop}_{\odot}\ r_d\ r_s$ and $\texttt{binop}_{\oplus}\ r_d\ r_1\ r_2$ respectively, where r_d is the destination register where the result of the operation must be stored and the other registers contain the operands. Object creation is modelled by `new` $r_d\ c$, which creates an object of class c and stores a pointer to it in r_d; array creation is similarly handled by `newarray` $r_d\ r_l\ \tau$, where r_d is the destination register where the pointer to the new array must be stored, r_l contains the array length and τ specifies the type of the array cells. The type cast statement `checkcast` $r_s\ \tau$ checks whether the register r_s contains a pointer to an object of type τ and it moves to the next instruction if this is the case, otherwise it stops the execution[2]. The statement `instof` $r_d\ r_s\ \tau$ stores `true` in

TABLE 2 μ-Dalvik$_A$ Syntax

P	::=	cls^*
cls	::=	`cls` $c \leq c'$ `imp` c^* $\{fld^*; mtd^*\}$
τ_{prim}	::=	`bool` | `int` | ...
τ	::=	$c \mid \tau_{prim} \mid$ `array`$[\tau]$
fld	::=	$f : \tau$
mtd	::=	$m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$
st	::=	`goto` pc
	|	`move` $lhs\ rhs$
	|	$\texttt{if}_{\lessgtr}\ r_1\ r_2$ `then` pc
	|	$\texttt{unop}_{\odot}\ r_d\ r_s$
	|	$\texttt{binop}_{\oplus}\ r_d\ r_1\ r_2$
	|	`new` $r_d\ c$
	|	`newarray` $r_d\ r_l\ \tau$
	|	`checkcast` $r_s\ \tau$
	|	`instof` $r_d\ r_s\ \tau$
	|	`invoke` $r_o\ m\ r^*$
	|	`sinvoke` $c\ m\ r^*$
	|	`return`
	|	`newintent` $r_i\ c$
	|	`put-extra` $r_i\ r_k\ r_v$
	|	`get-extra` $r_i\ r_k\ \tau$
	|	`start-activity` r_i
r	$\in$	*Registers*
pc	$\in$	$\mathbb{N}$
$\oplus$	::=	$+ \mid - \mid \ldots$
$\odot$	::=	$- \mid \neg \mid \ldots$
$\lessgtr$	::=	$< \mid > \mid \ldots$
$prim$	::=	`true` | `false` | ...
lhs	::=	r
	|	$r[r]$
	|	$r.f$
	|	$c.f$
rhs	::=	lhs
	|	$prim$

2. The corresponding Dalvik opcodes would raise an exception, but we do not model exceptions in our formalism.

r_d if r_s points to an object of type τ, otherwise it stores `false`. A method invocation `invoke` r_o m r^* calls the method m on the receiver object pointed by r_o, passing the values in the registers r^* as actual arguments. The invocation of static methods is modelled by `sinvoke` c m r^*. The `return` statement has no argument, rather there is a special register r_{ret} for holding return values: the return value must be moved to r_{ret} by the callee before calling `return`.

The last four statements are used to model inter-component communication. Intent creation is modelled by `newintent` r_i c, which creates an intent for the activity c and stores a pointer to it in r_i. The statement `put-extra` r_i r_k r_v adds to the intent pointed by r_i a new key-value binding $k \mapsto v$, where k and v are the contents of r_k and r_v respectively. The statement `get-extra` r_i r_k τ retrieves from the intent pointed by r_i the value bound to key k, where k is the content of r_k, provided that this value has type τ. Finally, `start-activity` r_i sends the intent pointed by r_i, thus starting a new activity. Throughout the paper, we only consider *well-formed* programs.

Definition 1. A program P is *well-formed* iff: (1) all its class names are pairwise distinct, (2) for each of its classes, all the field names are pairwise distinct, and (3) for each of its classes, all the method names are pairwise distinct.

Notice that the last condition of the definition above is not restrictive, since overloading resolution is performed at compile time in Java [1] and Dalvik bytecode thus identifies methods through their signature, rather than their name. In our formalism, we then suppose that method names are tagged with some distinctive information drawn from their signature, so that we can identify each method of a given class just by its name. Notice that two different classes can still define two methods with the same name, which is important to model dynamic dispatching.

From now on, we focus our attention on some well-formed program $P = cls^*$. Most of the definitions we present in the paper depend on P, but we do not make this dependence explicit in the notation to keep it lighter.

3.3. Dalvik Semantics

Table 3 defines the semantic domains employed by the operational semantics of μ-Dalvik$_A$. Values include primitive values and *locations*, i.e., pointers to heap elements extended with an annotation λ. Annotations have no semantic import and are only needed for our static analysis: we will discuss their role in Section 4.

A *local configuration* $\Sigma = \alpha \cdot \pi \cdot H \cdot S$ represents the state of a specific activity. It includes a call stack α, a pending activity stack π, a heap H, and a static heap S. A call stack α is a list of *local states*, which is populated upon method invocation. Each local state includes: (1) a program point $pp = c, m, pc$, where c and m identify the invoked method, while pc points to the next instruction to execute; (2) a list of statements st^*, modelling the method body; and (3) a map R binding local registers to their current value.

A pending activity stack π is a list of intents, which are treated as (untyped) dictionaries in our formalism. As anticipated, for the sake of simplicity, we only consider explicit intents in the formalization, i.e., intents which are meant to be delivered to an activity of a given class c: this class is specified after the 'at' symbol (@) in the intent syntax[3]. We use π to keep track of which activities have been started by the activity modelled by the local configuration.

Finally, a heap H is a mapping between locations and memory blocks, where each block is either an object, an array or an intent. Object fields are annotated with their static type, though we typically omit this annotation when it is unimportant. The static heap S simply binds static fields to their corresponding value.

The small-step operational semantics of μ-Dalvik$_A$ is defined by a reduction relation $\Sigma \rightsquigarrow \Sigma'$. Reduction takes place by fetching the next statement to execute, based on the program counter of the top-most local state of the call stack in Σ, and by running it to produce Σ'. The definition of the reduction relation is lengthy, but unsurprising, and it is given in the full version [7]. The only point worth noticing here is that, when a new memory block is created, e.g., by `new`, the corresponding pointer to the heap is annotated with the program point c, m, pc where creation takes place.

3.4. Activity Semantics

The operational behaviour of an activity does not depend only on its bytecode implementation, but also on external events, like user inputs and system callbacks. The event-driven nature of Android applications gives rise to highly non-deterministic executions, which are not trivial to approximate correctly by static analysis.

3.4.1. Formalizing Activities. We start by introducing a formal notion of activity.

Definition 2. A class cls is an *activity class* if and only if $cls = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*\}$ for some $c' \leq \texttt{Activity}$. An *activity* is an instance of an activity class. We stipulate that each activity has the following fields: (1) *finished*: a boolean flag stating whether the activity has finished or not; (2) *intent*: a pointer to the intent which started the activity; (3) *result*: a pointer to an intent storing the result of the activity computation; and (4) *parent*: a pointer to the parent activity, i.e., the activity which started the present one.

We require that each activity has a (possibly empty) set of *event handlers* for user inputs: given an activity class c, we let $handlers(c) = \{m_1, \ldots, m_n\}$ be the set of the names of the methods of c which may be dispatched when some user input event occurs. We assume a set of activity states *ActStates* and a relation *Lifecycle* $\subseteq$ *ActStates* $\times$ *ActStates* defining the state transitions admitted by the activity lifecycle [26]. We assume that each activity class c has a set

3. Extending the formalism to include implicit intents would not be difficult, but this would introduce non-determinism on the choice of the receiving activity, thus making the presentation harder to follow.

TABLE 3 μ-Dalvik$_A$ Semantic Domains

Pointers	p	$\in$	*Pointers*				
Program points	pp	$::=$	c, m, pc	Objects	o	$::=$	$\{\!\lvert c; (f_\tau \mapsto v)^* \rvert\!\}$
Annotations	λ	$::=$	$pp \mid c \mid in(c)$	Arrays	a	$::=$	$\tau[v^*]$
Locations	ℓ	$::=$	p_λ	Intents	i	$::=$	$\{\!\lvert @c; (k \mapsto v)^* \rvert\!\}$
Values	u, v	$::=$	$prim \mid \ell$	Memory blocks	b	$::=$	$o \mid a \mid i$
Registers	R	$::=$	$(r \mapsto v)^*$	Heaps	H	$::=$	$(\ell \mapsto b)^*$
Local states	L	$::=$	$\langle pp \cdot st^* \cdot R \rangle$	Static heaps	S	$::=$	$(c.f \mapsto v)^*$
Call stacks	α	$::=$	$\varepsilon \mid L :: \alpha$	Local configurations	Σ	$::=$	$\alpha \cdot \pi \cdot H \cdot S$
Pending activity stacks	π	$::=$	$\varepsilon \mid i :: \pi$				

of callbacks for each activity state s, whose names are returned by a function $cb(c, s)$; for the *running* state we let $cb(c, \textit{running}) = \textit{handlers}(c)$, i.e., when an activity is running, any callback set for user inputs may be dispatched.

We then extend the syntax of μ-Dalvik$_A$ with the elements in Table 4. A *frame* φ includes a location ℓ pointing to an activity, a corresponding activity state s, a pending activity stack π and a call stack α. Frames are organized in an *activity stack* Ω, modelling different activities executing in the same application: a single frame in Ω has priority of execution and is underlined. A *configuration* Ψ includes an activity stack Ω, a heap H and a static heap S.

TABLE 4 Extensions to the Syntax of μ-Dalvik$_A$

Activity states	s	$\in$	*ActStates*
Frames	φ	$::=$	$\langle \ell, s, \pi, \alpha \rangle \mid \underline{\langle \ell, s, \pi, \alpha \rangle}$
Activity stacks	Ω	$::=$	$\varphi \mid \varphi :: \Omega$
Configurations	Ψ	$::=$	$\Omega \cdot H \cdot S$

Convention: each activity stack Ω contains at most one active (underlined) frame.

3.4.2. Reduction Rules. Before presenting the formal semantics, we need to introduce some additional definitions. We start with the notion of *callback stack*, identifying the admissible format of a call stack for new frames pushed on the activity stack upon the invocation of a callback from the Android system. Let $\textit{sign}(c, m) = \tau^* \xrightarrow{n} \tau$ iff there exists a class cls_i such that $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\textit{fld}^*; \textit{mtd}^*, m : \tau^* \xrightarrow{n} \tau\ \{st^*\}\}$. Let then *lookup* stand for a *method lookup* function such that $\textit{lookup}(c, m) = (c', st^*)$ iff: (1) c' is the class defining the method which is dispatched when m is invoked on an object of type c, and (2) st^* is the method body.

Definition 3. Given a location ℓ pointing to an activity of class c, we let $\alpha_{\ell.s}$ stand for an arbitrary *callback stack* for state s, i.e., any call stack $\langle c', m, 0 \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c', st^*) = \textit{lookup}(c, m)$ for some $m \in cb(c, s)$, $\textit{sign}(c', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+j} \mapsto v_j)^{j \leq n}),$$

for some values $v_1, \ldots, v_n$ of the correct type $\tau_1, \ldots, \tau_n$.

In the definition, we let $\mathbf{0}$ be the default value for local registers. There is just one default value for registers in the model, since registers are untyped in Dalvik. In the following, it is also convenient to presuppose for each type τ the existence of a a default value $\mathbf{0}_\tau$, used to initialize fields of type τ upon object creation.

A tricky aspect of the operational semantics of activities, which has never been formalized before, is the *serialization* of objects upon inter-component communication. Different activities may exchange objects using intents, but these objects are never passed by reference: rather, they are serialized at the sender side and a copy of them is created at the receiver side. The intent itself is serialized upon communication. We formalize this serialization routine by two mutually recursive functions $\textit{ser}^H_{\textit{Val}}(v) = (v', H')$ and $\textit{ser}^H_{\textit{Blk}}(b) = (b', H')$, returning a serialized copy of their argument and a new heap where all the pointers created in the serialization process have been instantiated correctly. We refer to Table 5 below for the definition of the two functions. Their definition uses a set of pointers Γ to keep track of which pointers have already been followed in the serialization process, so as to allow the serialization of memory blocks including self-references.

Finally, the operational semantics requires the next definition of *successful* call stack. A successful call stack is the call stack of an activity which has completed its computation.

Definition 4. A call stack α is *successful* if and only if $\alpha = \langle pp \cdot \texttt{return} \cdot R \rangle :: \varepsilon$ for some pp and R. We let $\overline{\alpha}$ range over successful call stacks.

Now we have all the ingredients to define the formal semantics of activities, which is given by the reduction rules in Table 5. As anticipated, the rules closely follow previous work by Payet and Spoto [26], which we extend to provide a more accurate account of inter-component communication by modelling value-passing based on a serialization routine. We give a short explanation of all the rules, we refer to [26] for a longer description.

Rule (A-ACTIVE) allows the execution of the statements in the active frame, using the reduction relation for local configurations described in Section 3.3. Rule (A-DEACTIVATE) models the situation where the active frame has run up to completion: the frame loses priority and one of the other rules can be applied. Rule (A-STEP) models

TABLE 5 Reduction Relation for Configurations ($\Omega \cdot H \cdot S \Rightarrow \Omega' \cdot H' \cdot S'$)

(A-ACTIVE)
$$\frac{\alpha \cdot \pi \cdot H \cdot S \rightsquigarrow \alpha' \cdot \pi' \cdot H' \cdot S'}{\Omega :: \underline{\langle \ell, s, \pi, \alpha \rangle} :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \underline{\langle \ell, s, \pi', \alpha' \rangle} :: \Omega' \cdot H' \cdot S'}$$

(A-DEACTIVATE)
$$\Omega :: \underline{\langle \ell, s, \pi, \overline{\alpha} \rangle} :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega' \cdot H \cdot S$$

(A-STEP)
$$\frac{(s, s') \in \mathit{Lifecycle} \qquad \pi \neq \varepsilon \Rightarrow (s, s') = (\mathit{running}, \mathit{onPause}) \qquad H(\ell).\mathit{finished} = \texttt{true} \Rightarrow (s, s') \in \{(\mathit{running}, \mathit{onPause}), (\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}}{\langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \alpha_{\ell.s'} \rangle} :: \Omega \cdot H \cdot S}$$

(A-DESTROY)
$$\frac{H(\ell).\mathit{finished} = \texttt{true}}{\Omega :: \langle \ell, \mathit{onDestroy}, \pi, \overline{\alpha} \rangle :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \Omega' \cdot H \cdot S}$$

(A-BACK)
$$\frac{H' = H[\ell \mapsto H(\ell)[\mathit{finished} \mapsto \texttt{true}]]}{\langle \ell, \mathit{running}, \varepsilon, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \langle \ell, \mathit{running}, \varepsilon, \overline{\alpha} \rangle :: \Omega \cdot H' \cdot S}$$

(A-REPLACE)
$$\frac{H(\ell) = \{\!|c; (f_\tau \mapsto v)^*, \mathit{finished} \mapsto u|\!\} \qquad o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \texttt{false}|\!\} \qquad H' = H, p_c \mapsto o}{\langle \ell, \mathit{onDestroy}, \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle p_c, \mathit{constructor}, \pi, \alpha_{p_c.\mathit{constructor}} \rangle} :: \Omega \cdot H' \cdot S}$$

(A-HIDDEN)
$$\frac{\varphi = \langle \ell, s, \pi, \overline{\alpha} \rangle \qquad s \in \{\mathit{onResume}, \mathit{onPause}\} \qquad (s', s'') \in \{(\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}}{\varphi :: \Omega :: \langle \ell', s', \pi', \overline{\alpha}' \rangle :: \Omega' \cdot H \cdot S \Rightarrow \varphi :: \Omega :: \underline{\langle \ell', s'', \pi', \alpha_{\ell'.s''} \rangle} :: \Omega' \cdot H \cdot S}$$

(A-START)
$$\frac{s \in \{\mathit{onPause}, \mathit{onStop}\} \qquad i = \{\!|@c; (k \mapsto v)^*|\!\} \qquad \emptyset \vdash \mathit{ser}^H_{\mathit{Blk}}(i) = (i', H') \qquad p_c, p'_{in(c)} \notin \mathit{dom}(H, H') \qquad o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \texttt{false}, \mathit{intent} \mapsto p'_{in(c)}, \mathit{parent} \mapsto \ell|\!\} \qquad H'' = H, H', p_c \mapsto o, p'_{in(c)} \mapsto i'}{\langle \ell, s, i :: \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle p_c, \mathit{constructor}, \varepsilon, \alpha_{p_c.\mathit{constructor}} \rangle} :: \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega \cdot H'' \cdot S}$$

(A-SWAP)
$$\frac{\varphi' = \langle \ell', \mathit{onPause}, \varepsilon, \overline{\alpha}' \rangle \qquad H(\ell').\mathit{finished} = \texttt{true} \qquad \varphi = \langle \ell, s, i :: \pi, \overline{\alpha} \rangle \qquad s \in \{\mathit{onPause}, \mathit{onStop}\} \qquad H(\ell').\mathit{parent} = \ell}{\varphi' :: \varphi :: \Omega \cdot H \cdot S \Rightarrow \varphi :: \varphi' :: \Omega \cdot H \cdot S}$$

(A-RESULT)
$$\frac{\varphi' = \langle \ell', \mathit{onPause}, \varepsilon, \overline{\alpha}' \rangle \qquad H(\ell').\mathit{finished} = \texttt{true} \qquad \varphi = \langle \ell, s, \varepsilon, \overline{\alpha} \rangle \qquad s \in \{\mathit{onPause}, \mathit{onStop}\} \qquad H(\ell').\mathit{parent} = \ell \qquad \emptyset \vdash \mathit{ser}^H_{\mathit{Val}}(H(\ell').\mathit{result}) = (\ell'', H') \qquad H'' = (H, H')[\ell \mapsto H(\ell)[\mathit{result} \mapsto \ell'']]}{\varphi' :: \varphi :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle \ell, s, \varepsilon, \alpha_{\ell.\mathit{onActivityResult}} \rangle} :: \varphi' :: \Omega \cdot H'' \cdot S}$$

where:

$$\Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(\mathit{prim}) = (\mathit{prim}, \cdot) \qquad \frac{p_\lambda \in \Gamma}{\Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(p_\lambda) = (\nu(p_\lambda), \cdot)}$$

$$\frac{p_\lambda \notin \Gamma \qquad \Gamma \cup \{p_\lambda\} \vdash \mathit{ser}^H_{\mathit{Blk}}(H(p_\lambda)) = (b, H'') \qquad H' = H'', \nu(p_\lambda) \mapsto b}{\Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(p_\lambda) = (\nu(p_\lambda), H')}$$

$$\frac{\forall i \in [1, n] : \Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(v_i) = (u_i, H_i) \qquad H' = H_1, \ldots, H_n}{\Gamma \vdash \mathit{ser}^H_{\mathit{Blk}}(\tau[(v_i)^{i \leq n}]) = (\tau[(u_i)^{i \leq n}], H')} \qquad \frac{\forall i \in [1, n] : \Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(v_i) = (u_i, H_i) \qquad H' = H_1, \ldots, H_n}{\Gamma \vdash \mathit{ser}^H_{\mathit{Blk}}(\{\!|c'; (f_i \mapsto v_i)^{i \leq n}|\!\}) = (\{\!|c'; (f_i \mapsto u_i)^{i \leq n}|\!\}, H')}$$

$$\frac{\forall i \in [1, n] : \Gamma \vdash \mathit{ser}^H_{\mathit{Val}}(v_i) = (u_i, H_i) \qquad H' = H_1, \ldots, H_n}{\Gamma \vdash \mathit{ser}^H_{\mathit{Blk}}(\{\!|@c'; (k_i \mapsto v_i)^{i \leq n}|\!\}) = (\{\!|@c'; (k_i \mapsto u_i)^{i \leq n}|\!\}, H')}$$

Conventions: the activity stack on the left-hand side does not contain underlined frames, but for the first two rules. In the serialization rules we assume the existence of a function $\nu(\_)$ assigning to each pointer a fresh pointer with the same annotation, used to store the result of the serialization.

the transition of the top-level activity from state s to one of its successors s' in the activity lifecycle: correspondingly, a new callback method is executed. Two side-conditions constrain the possible state transitions, based on the presence of pending activities to start and on whether the activity has finished or not.

Rule (A-DESTROY) models the removal of a finished activity from the activity stack. Rule (A-BACK) models the scenario where the user hits the back button on the Android device and the top-most activity gets finished by the system. Rule (A-REPLACE) corresponds to screen orientation changes: the foreground activity is destroyed and gets replaced by a fresh activity instance; notice that the new pointer to the heap is annotated with the class of the activity. Rule (A-HIDDEN) models the scenario where a new activity (the frame φ) has come to the foreground and hides a previously running activity, which gets stopped or destroyed by the system.

The starting of a new activity is modelled by rule (A-START). The top-most activity is paused or stopped and there is some intent i to be sent to c: the intent is serialized and a new instance of c is pushed on the activity stack, setting its *intent* field to a pointer to the serialized copy of i and setting its *parent* field to a pointer to the activity which sent the intent. The pointer to the new activity is annotated with the class c, while the pointer to the serialized copy of the intent gets the annotation $in(c)$: again, this is needed just for the static analysis and will be discussed later. Notice that, if multiple activities need to be started, rule (A-SWAP) allows a parent activity to substitute itself to a child activity on the top of the activity stack, so that rule (A-START) can be applied again to fire the remaining intents. Finally, rule (A-RESULT) allows a finished activity in the foreground to return the result of its computation to the parent activity: the parent activity gets a serialized copy of the result and becomes active by executing a corresponding callback, bound to the *onActivityResult* state.

3.5. Examples

One reason why it is useful to have a formal semantics before devising a static analysis technique is to pinpoint corner cases which may potentially lead to unsound analysis results. We discuss two examples below.

3.5.1. Static Fields. Even though inter-component communication does not allow for the exchange of references, activities in the same application can still share memory by using static fields. This is apparent in the formal semantics, since the syntax of configurations Ψ contains a global static heap S, which can be accessed by using publicly known names of static fields. We then observe that the order of execution of different activities, or even different callbacks inside the same activity, is very hard to predict: for instance, the rules in Table 5 highlight that even activities which are not on the top of the activity stack may become active and execute callbacks by rule (A-HIDDEN). Also, the same callback may be executed multiple times, since an activity may be routinely recreated by the Android system due to user activities (e.g., screen orientation changes), which cannot be known statically, as modelled by rule (A-REPLACE).

The implication on static analysis is that it is extremely challenging to implement flow-sensitivity on accesses to static fields without producing unsound results. Furthermore, given that static fields may be used to share pointers to heap locations, flow-sensitivity for heap accesses is also hard to achieve. Since we target soundness in this work, the static analysis we devise in the next section is flow-insensitive on both static fields and heap locations.

3.5.2. Serialization. Rule (A-START) of the operational semantics highlights that intents are serialized upon inter-component communication. This means that, when a parent activity starts a child activity, the latter operates on a copy of the intent sent by the former and not on the same intent.

The implication on static analysis is that, although the callback bound to the *onActivityResult* state of the parent activity is always executed after the construction of the child activity, no change to the intent done by the child activity should overwrite the original over-approximation of the intent computed for the parent activity when a result is returned to it. This applies to any object which is serialized with the intent. The static analysis in the next section provides a conservative over-approximation of this behaviour.

4. Static Analysis

The static analysis we propose works by translating an input program P into a corresponding *abstract* program Δ, i.e., a set of Horn clauses modelling an over-approximation of its semantics. By feeding these clauses to an automated theorem prover and by showing the unsatisfiability of an appropriate logical formula, we can prove that some set of undesired configurations is never reached by P.

4.1. Overview

The analysis is based on the syntactic categories in Table 6. We start by discussing how values are approximated. We presuppose the existence of an arbitrary set of abstract domains used to approximate primitive values: for each primitive value $prim$, we assume that there exists a corresponding abstraction $\widehat{prim}$, e.g., integer numbers could be approximated by their sign. Locations of the form $\ell = p_\lambda$, instead, are abstracted into their annotation λ. An *abstract value* $\hat{v}$ is a set of elements drawn from either the abstract domains or the set of annotations.

The different forms of annotations λ provide insight on different aspects of the static analysis. Program point annotations $pp = c, m, pc$ are used to represent pointers to memory blocks instantiated using the statements `new`, `newarray` and `newintent`: by abstracting these elements with the program point where they are created, we implement a *plain-object-sensitive* static analysis [29]. We chose it because it is well-understood and convenient to both formalize and

TABLE 6 Abstract Domains and Analysis Facts

Facts	f	::=		Abs. values	$\hat{u}, \hat{v}$	::=	$\emptyset \mid \{\widehat{prim}\} \mid \{\lambda\} \mid \hat{v} \cup \hat{v}$
Abs. registers		|	$\mathsf{R}_{\mathsf{pp}}(t^*\,;\,t^*)$	Abs. objects	$\hat{o}$	::=	$\{\!\lvert c; (f_\tau \mapsto \hat{v})^* \rvert\!\}$
Abs. heap entries		|	$\mathsf{H}(t, t')$	Abs. arrays	$\hat{a}$	::=	$\tau[\hat{v}]$
Abs. static fields		|	$\mathsf{S}_{\mathsf{c},\mathsf{f}}(t)$	Abs. intents	$\hat{i}$	::=	$\{\!\lvert @c; \hat{v} \rvert\!\}$
Abs. right-hand sides		|	$\mathsf{RHS}_{\mathsf{pp}}(t)$	Abs. mem. blocks	$\hat{b}$	::=	$\hat{o} \mid \hat{a} \mid \hat{i}$
Abs. results		|	$\mathsf{Res}_{\mathsf{c},\mathsf{m}}(t^*\,;\,t)$				
Abs. pending activities		|	$\mathsf{I}(t, t')$	Variables	x, y	$\in$	*Vars*
Set membership		|	$t \in t'$	Constants	k	::=	$\hat{v} \mid \hat{b} \mid \tau \mid \lambda$
Subtyping		|	$t \leq t'$	Terms	t	::=	$\mathsf{k} \mid x \mid in(t)$
Horn clauses		|	$\forall x^*. \bigwedge_i \mathsf{f}_i \implies \mathsf{f}$				
Abs. programs	Δ	::=	$\{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$				

present: we plan to integrate more advanced analyses like 2full+1H in future releases. Class name annotations c, instead, are used to represent activities in an object-insensitive way: different activities of the same class c are all abstracted by the annotation c, since it is generally hard to statically discriminate between different activity instances. Finally, we use the annotation $in(c)$ to abstract all the intents which are used to start an activity of class c.

Coming to memory blocks, our analysis is field-sensitive on objects, but field-insensitive on both arrays and intents. It is easier to implement field-sensitivity for objects, since field names are statically known in Java. Implementing field-sensitivity for arrays would require precise information on array bounds and indexes; intents, instead, would need an accurate string analysis, to deal with their dictionary-like programming patterns. It would be possible to leverage existing proposals [10] to implement a more precise analysis in terms of field-sensitivity, but we propose a simpler framework here to focus on the Android-specific aspects of the analysis. Notice that, just like the objects they approximate, abstract objects $\hat{o}$ feature type annotations on their fields, which are omitted when unimportant.

Abstract values and abstract memory blocks, plus all the types available in the analysed program and the annotations, determine a universe of *constants*, ranged over by k. A *term* t is either a constant k, a variable x drawn from a denumerable set *Vars* disjoint from the set of constants, or an expression of the form $in(t')$ for some term t'. The set of terms is used to define the syntax of *facts* f, logical formulas built on selected predicate symbols used by the analysis.

The fact $\mathsf{R}_{\mathsf{c},\mathsf{m},\mathsf{pc}}(\hat{u}^*\,;\,\hat{v}^*)$ states that, whenever the method m of class c is invoked with some arguments over-approximated by $\hat{u}^*$, the state of the local registers at the pc-th statement is over-approximated by $\hat{v}^*$. The syntax of the fact highlights that: (1) the analysis is flow-sensitive for register values, since it computes different static approximations at different program points, and (2) method invocations are handled in a context-sensitive way, where the notion of context coincides with the (abstraction of) the actual arguments supplied to the method upon invocation. The fact $\mathsf{H}(\lambda, \hat{b})$ states that some location p_λ refers to a heap element storing a memory block over-approximated by $\hat{b}$ at some point of the program execution. Notice that the fact does not contain any program point information, i.e., the analysis is flow-insensitive for heap locations, which is important for soundness (see Section 3.5). Similarly, the fact $\mathsf{S}_{\mathsf{c},\mathsf{f}}(\hat{v})$ states that the static field f of class c contains a value which is over-approximated by $\hat{v}$ at some point of the program execution. The fact $\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ states that the right-hand side of the `move` statement at program point pp evaluates to a value over-approximated by $\hat{v}$. The fact $\mathsf{Res}_{\mathsf{c},\mathsf{m}}(\hat{u}^*\,;\,\hat{v})$ states that, whenever the method m of class c is invoked with some arguments over-approximated by $\hat{u}^*$, its return value is over-approximated by $\hat{v}$. The fact $\mathsf{I}(c, \hat{i})$ tracks that an activity of class c has sent an intent which is over-approximated by $\hat{i}$. We then have set membership facts $t \in t'$ and subtyping facts $\tau \leq \tau'$ with the obvious meaning.

Finally, Horn clauses define the abstract semantics of programs. A Horn clause has the form:

$$\forall x_1, \ldots, \forall x_m. \mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f},$$

where all the variables of $\mathsf{f}_1, \ldots, \mathsf{f}_n, \mathsf{f}$ belong to $\{x_1, \ldots, x_m\}$ and each variable of f occurs among the variables of $\mathsf{f}_1, \ldots, \mathsf{f}_n$. Since most of the Horn clauses we present do not make use of constants, to improve readability we omit the universal quantifiers in front of Horn clauses and we just represent each variable occurring therein with a constant of the expected type. The few exceptions where constants are actually used are disambiguated using a sans serif font, e.g., we use c to denote the constant corresponding to the activity class c specifically, rather than some universally quantified variable standing for an arbitrary activity class. We let an underscore (_) stand for any syntactic element occurring in a Horn clause which is not significant to understanding.

4.2. Analysis Specification

4.2.1. Abstract Semantics of Dalvik. We start by presenting the abstract evaluation rules for right-hand sides, which are simple and provide a good intuition on how the static analysis works. These rules are given in Table 7.

To abstract a primitive value $prim$ at any program point pp, we just pick the corresponding element $\widehat{prim}$ from the

TABLE 7 Abstract Evaluation of Right-hand Sides

$$\langle\!\langle prim \rangle\!\rangle_{pp} = \{\mathsf{RHS}_{\mathsf{pp}}(\{\widehat{prim}\})\} \qquad \langle\!\langle r_i \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_;\, \hat{v}^*) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v}_i)\} \qquad \langle\!\langle c.f \rangle\!\rangle_{pp} = \{\mathsf{S}_{\mathsf{c,f}}(\hat{v}) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v})\}$$

$$\langle\!\langle r_i.f \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_;\, \hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{\!|c; (f' \mapsto \hat{v}')^*, f \mapsto \hat{u}|\!\}) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

$$\langle\!\langle r_i[r_j] \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_;\, \hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \tau[\hat{u}]) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

underlying abstract domain. To abstract the content of the register r_i at program point pp, we take the fact $\mathsf{R}_{\mathsf{pp}}(\_;\, \hat{v}^*)$ and we return the i-th abstract value $\hat{v}_i$. To abstract the content of a static field $c.f$ at any program point, we take any fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ and we return the abstract value $\hat{v}$. Abstracting the content of the field f of an object at program point pp is slightly more complicated: if the pointer to the object is stored in the register r_i, we pick the i-th abstract value $\hat{v}_i$ from the fact $\mathsf{R}_{\mathsf{pp}}(\_;\, \hat{v}^*)$ modelling the state of the registers at pp; then, if $\hat{v}_i$ contains any pointer abstraction λ, we use it to match a corresponding abstract heap entry $\mathsf{H}(\lambda, \hat{o})$ and we return the value of the field f of the abstract object $\hat{o}$ contained therein. We similarly abstract the content of array cells: just notice that, since the representation of arrays is field-insensitive, the index of the cell does not play any role in the static analysis.

The rules for abstracting a right-hand side are useful to define the abstract semantics of the `move` statement. Other statements require some additional definitions. First, for each comparison operator $\lessgtr$ and each primitive operation $\odot, \oplus$ of the concrete semantics, we presuppose the existence of a corresponding abstract operation $\hat{\lessgtr}$, $\hat{\odot}$ and $\hat{\oplus}$ defined over the elements of the appropriate abstract domain. Then, given an abstract memory block $\hat{b}$, we define a function $\widehat{\textit{get-type}}(\hat{b})$ as follows:

$$\widehat{\textit{get-type}}(\hat{b}) = \begin{cases} c & \text{if } \hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\} \\ \texttt{array}[\tau] & \text{if } \hat{b} = \tau[\hat{v}] \\ \texttt{Intent} & \text{if } \hat{b} = \{\!|@c; \hat{v}|\!\} \end{cases}$$

Finally, we assume a function $\widehat{\textit{lookup}}(m)$, which returns the set of classes which define (or inherit) a method called m.

With these definitions, we are ready to introduce the abstract semantics of statements. The idea is to define, for each possible form of statement st, a translation $(\!|st|\!)_{pp}$ into a set of Horn clauses, which over-approximate the semantics of st at program point pp. The full formal semantics of the translation is given in Table 8 and explained below.

The rule for `goto` pc' propagates the state of the registers at the current program counter pc to pc'. The rule for $\texttt{if}_{\lessgtr}\ r_i\ r_j\ \texttt{then}\ pc'$ propagates the state of the registers at the current program counter pc either to pc' or to $pc + 1$, based on the outcome of a comparison $\hat{\lessgtr}$ between the abstract values $\hat{v}_i$ and $\hat{v}_j$ approximating the content of registers r_i and r_j respectively: both branches may be enabled, as the result of an over-approximation of the contents of the registers. The two rules for unary and binary operations just employ the appropriate abstract operation to update the approximation of the content of the destination register r_d. The four rules for the `move` statement rely on the auxiliary rules for abstracting a right-hand side we introduced before: these rules store their result in a RHS fact, which occurs in the premises of the Horn clause used to update the abstraction of the left-hand side. The most interesting point to notice here is that field-sensitivity or its absence has an import on how fields are updated: for objects, we replace the old value of the field with the new one; for arrays and intents, instead, we add the new value to the old approximation, since their abstraction over-approximates the content of the entire data structure, rather than just the single element which is updated. The rules for `instof` and `checkcast` use the $\widehat{\textit{get-type}}$ function previously defined.

The rule for `invoke` is the most complicated one, since it has to deal with dynamic dispatching. The challenge here is that the name of the invoked method is statically known from the syntax of the statement, but the method implementation is not, since it depends on the runtime type of the receiver object, an information which is only over-approximated when solving the Horn clauses, rather than when generating them. We then use the method name and the number of arguments passed upon invocation to narrow the set of possible classes of the receiver object, using the functions $\widehat{\textit{lookup}}$ and *sign*, and we generate one Horn clause for each of them. We then rely on subtyping to make the analysis precise, by imposing that a Horn clause generated for class c'' can only be fired if the class c' of (the abstraction of) the receiver object is a subtype of c''. Besides implementing a sound approximation of the dynamic dispatching mechanism, the rule for `invoke` generates additional Horn clauses used to propagate the abstraction of the method return value from the callee to the caller: this is done by using a Res fact, which is introduced by a `return` statement in the implementation of the callee, as we discuss below. The rule for static method invocation follows a similar logic, but it is significantly simpler, due to the lack of dynamic dispatching on static calls.

The rules for object and array creation create a new abstract heap entry $\mathsf{H}(\lambda, \hat{b})$, where λ is the current program point and $\hat{b}$ is the abstraction of a freshly initialized object/array. The rule for `return` introduces a Res fact, storing an over-approximation of the method return value; notice that the arguments $\hat{v}^*_{call}$ supplied upon method invocation are propagated in the Res fact, which is important to implement context-sensitivity, i.e., to propagate the result to the

TABLE 8 Abstract Semantics of μ-Dalvik$_A$ - Statements (let $pp = c, m, pc$)

$$
\begin{array}{lcl}
(\!|\texttt{goto } pc'|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc'}}(\_;\hat{v}^*)\} \\
(\!|\texttt{if}_{\otimes}\ r_i\ r_j \texttt{ then } pc'|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \hat{v}_i\ \hat{\otimes}\ \hat{v}_j \implies \mathsf{R}_{\mathsf{c,m,pc'}}(\_;\hat{v}^*)\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \neg(\hat{v}_i\ \hat{\otimes}\ \hat{v}_j) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \\
(\!|\texttt{binop}_{\oplus}\ r_d\ r_i\ r_j|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \hat{v}_i\ \hat{\oplus}\ \hat{v}_j])\} \\
(\!|\texttt{unop}_{\odot}\ r_d\ r_i|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \hat{\odot}\ \hat{v}_i])\} \\
(\!|\texttt{move } r_d\ rhs|\!)_{pp} & = & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \wedge \mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \hat{v}'])\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \\
(\!|\texttt{move } r_a[r_{idx}]\ rhs|\!)_{pp} & = & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_a \wedge \mathsf{H}(\lambda, \tau[\hat{v}']) \implies \mathsf{H}(\lambda, \tau[\hat{v}' \cup \hat{v}''])\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \\
(\!|\texttt{move } r_o.f\ rhs|\!)_{pp} & = & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\}) \implies \\
 & & \mathsf{H}(\lambda, \{|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\})\} \cup \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \\
(\!|\texttt{move } c'.f\ rhs|\!)_{pp} & = & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \implies \mathsf{S}_{c',f}(\hat{v}')\} \cup \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \\
(\!|\texttt{instof } r_d\ r_s\ \tau|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{get\text{-}type}(\hat{b}) \leq \tau \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \widehat{\texttt{true}}])\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{get\text{-}type}(\hat{b}) \not\leq \tau \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \widehat{\texttt{false}}])\} \\
(\!|\texttt{checkcast } r_s\ \tau|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{get\text{-}type}(\hat{b}) \leq \tau \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \\
(\!|\texttt{invoke } r_o\ m'\ (r_{i_j})^{j\leq n}|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{|c';(f \mapsto \hat{u})^*|\}) \wedge c' \leq \mathsf{c}'' \implies \\
 & & \mathsf{R}_{\mathsf{c}'',\mathsf{m}',0}((\hat{v}_{i_j})^{j\leq n};(\hat{\mathbf{0}}_k)^{k\leq loc},(\hat{v}_{i_j})^{j\leq n}) \mid c'' \in \widehat{lookup}(m') \wedge \widehat{sign}(c'',m') = (\tau_j)^{j\leq n} \xrightarrow{loc} \tau\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{|c';(f \mapsto \hat{u})^*|\}) \wedge c' \leq \mathsf{c}'' \wedge \mathsf{Res}_{\mathsf{c}'',\mathsf{m}'}((\hat{v}_{i_j})^{j\leq n};\hat{v}'_{ret}) \implies \\
 & & \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[ret \mapsto \hat{v}'_{ret}]) \mid c'' \in \widehat{lookup}(m')\} \\
(\!|\texttt{sinvoke } c'\ m'\ (r_{i_j})^{j\leq n}|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c}',\mathsf{m}',0}((\hat{v}_{i_j})^{j\leq n};(\hat{\mathbf{0}}_k)^{k\leq loc},(\hat{v}_{i_j})^{j\leq n}) \mid \widehat{sign}(c',m') = (\tau_j)^{j\leq n} \xrightarrow{loc} \tau\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \mathsf{Res}_{\mathsf{c}',\mathsf{m}'}((\hat{v}_{i_j})^{j\leq n};\hat{v}'_{ret}) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[ret \mapsto \hat{v}'_{ret}])\} \\
(\!|\texttt{new } r_d\ c'|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \{|c';(f \mapsto \hat{\mathbf{0}}_\tau)^*|\})\} \cup \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \mathsf{pp}])\} \\
(\!|\texttt{newarray } r_d\ r_l\ \tau|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \tau[\hat{\mathbf{0}}_\tau])\} \cup \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \mathsf{pp}])\} \\
(\!|\texttt{return}|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\hat{v}^*_{call};\hat{v}^*) \implies \mathsf{Res}_{\mathsf{c,m}}(\hat{v}^*_{call};\hat{v}_{ret})\} \\
(\!|\texttt{start-activity } r_i|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{|@c';\hat{u}|\}) \implies \mathsf{I}(\mathsf{c}, \{|@c';\hat{u}|\})\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \\
(\!|\texttt{newintent } r_d\ c'|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \{|@c';\emptyset|\})\} \cup \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d \mapsto \mathsf{pp}])\} \\
(\!|\texttt{put-extra } r_i\ r_k\ r_j|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{|@c';\hat{v}'|\}) \implies \mathsf{H}(\lambda, \{|@c';\hat{v}' \cup \hat{v}_j|\})\} \cup \\
 & & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*)\} \\
(\!|\texttt{get-extra } r_i\ r_k\ \tau|\!)_{pp} & = & \{\mathsf{R}_{\mathsf{pp}}(\_;\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{|@c';\hat{v}'|\}) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[ret \mapsto \hat{v}'])\}
\end{array}
$$

right caller. The rule for `start-activity` tracks that the present activity c has sent an intent: an over-approximation of the intent is propagated from the corresponding abstract heap entry into the I fact modelling the presence of a pending activity which is about to start. The last rules for managing intents should be easy to understand, based on the intuitions given for the other rules.

4.2.2. Abstract Semantics of Activities. We can finally introduce the abstract semantics of activities. Intuitively, it is defined by: (1) the Horn clauses produced by translating each statement in the bytecode, and (2) a small set of bytecode-independent Horn clauses, abstracting the event-driven behaviour of activities. This is formalized next.

Definition 5. Let $P = (cls_i)^{i\leq n}$ be a program where $cls_i =$ `cls` $c_i \leq c'$ `imp` c^* $\{fld^*; (mtd_j)^{j\leq h_i}\}$ and $mtd_j = m_j : \tau^* \xrightarrow{loc} \tau\ \{(st_k)^{k\leq s_{ij}}\}$, we let $(\!|P|\!)$ be defined as follows:

$$(\!|P|\!) = \bigcup_{i\leq n, j\leq h_i, k\leq s_{ij}} (\!|st_k|\!)_{c_i,m_j,k} \cup \mathcal{R},$$

where $\mathcal{R}$ stands for the union of all the rules in Table 9.

We explain the rules from Table 9. Rule *Cbk* simulates the invocation of a callback: since we do not approximate the activity state in the abstract semantics, any callback method bound to a state s of the activity lifecycle may be non-deterministically dispatched; the statically unknown arguments supplied to the callback are abstracted by the top element ($\top$) of the abstract domain associated to their type, which is a sound over-approximation of any value of that type. Rule *Fin* tracks updates to the *finished* field of an activity in the abstract semantics: since it is hard to statically track whether an activity has finished or not, the rule sets the field to the top element of the abstract domain used to represent boolean value ($\top_{\texttt{bool}}$). Rule *Rep* approximates the behaviour of rule (A-REPLACE) of the concrete semantics: the activity fields may be reset to their default abstract value as the result of a screen orientation change.

Rule *Act* represents the starting of a new activity. If an intent has been sent by an activity of class c' to start an activity of class c, we introduce: (1) a new abstract heap entry to bind an abstraction of the intent to $in(c)$, and (2) a new abstract heap entry to bind an abstraction of the started activity to c. No serialization happens in the abstract semantics: if an intent is used to send an object in the concrete semantics, a reference to the corresponding abstract object is sent in our abstraction. This is sound, since our analysis is flow-insensitive on heap values, hence no over-approximation of the original object is ever lost as

TABLE 9 Abstract Semantics of μ-Dalvik$_A$ - Activity Rules

$$
\begin{array}{lcl}
\textit{Cbk} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^*|\!\}) \wedge c \leq \mathsf{c}' \implies \mathsf{R}_{\mathsf{c}',\mathsf{m},0}((\top_{\tau_j})^{j\leq n}; (\hat{\mathbf{0}}_k)^{k\leq loc}, c, (\top_{\tau_j})^{j\leq n}) \mid \\
 & & c' \text{ is an activity class} \wedge \exists s : m \in cb(c', s) \wedge \textit{sign}(c', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau\} \\
\textit{Fin} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \textit{finished} \mapsto \_|\!\}) \implies \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \textit{finished} \mapsto \top_{\texttt{bool}}|\!\})\} \\
\textit{Rep} & = & \{\mathsf{H}(c, \{\!|c; (f_\tau \mapsto \_)^*|\!\}) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^*|\!\})\} \\
\textit{Act} & = & \{\mathsf{I}(c', \{\!|@c; \hat{v}|\!\})) \implies \mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\})\} \cup \\
 & & \{\mathsf{I}(c', \{\!|@c; \hat{v}|\!\})) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^*, \textit{finished} \mapsto \widehat{\texttt{false}}, \textit{parent} \mapsto c', \textit{intent} \mapsto in(c)|\!\})\} \\
\textit{Res} & = & \{\mathsf{H}(c', \{\!|c'; (f' \mapsto \_)^*, \textit{parent} \mapsto c, \textit{result} \mapsto \lambda|\!\}) \wedge \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \textit{result} \mapsto \_|\!\} \implies \\
 & & \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \textit{result} \mapsto \lambda|\!\})\} \\
\textit{Sub} & = & \{\tau \leq \tau' \mid \tau \leq \tau' \text{ is a valid subtyping judgement}\}
\end{array}
$$

the result of an update to the heap at the receiver side. We then have rule *Res*, which is used to communicate a result from a child activity to its parent, thus simulating the behaviour of rule (A-RESULT) in the concrete semantics; again, no serialization happens in the process, rather a pointer to the result is passed. Finally, rule *Sub* corresponds to an axiomatization of the subtyping relationships for the analysed program.

4.3. Formal Results

The soundness of the analysis is proved using *representation functions*, a standard approach in program analysis [25]. The representation function β_{Cnf} maps an arbitrary configuration Ψ into a corresponding set of facts Δ, modelling an over-approximation of Ψ. Its definition is lengthy, but unsurprising, e.g., each element $\ell \mapsto b$ of the heap is converted into an abstract heap entry $\mathsf{H}(\lambda, \hat{b})$, where λ is the annotation on ℓ and $\hat{b}$ is an abstraction of b. After defining β_{Cnf}, we introduce a partial order $\sqsubseteq$ on analysis facts, with the intuitive understanding that $\mathsf{f} \sqsubseteq \mathsf{f}'$ whenever f is a more precise abstraction than f'. The partial order is then lifted to abstract programs by having $\Delta <: \Delta'$ if and only if $\forall \mathsf{f} \in \Delta : \exists \mathsf{f}' \in \Delta' : \mathsf{f} \sqsubseteq \mathsf{f}'$.

Our main theorem states that any reachable configuration in the concrete semantics is over-approximated by some set of facts which is provable using the abstract semantics of the program and an abstraction of the initial configuration. The proof is parametric with respect to the choice of the abstract domains/operations used for primitive values, provided they offer some minimal soundness guarantees. This allows for choosing different trade-off between efficiency and precision of the analysis.

Theorem 1 (Preservation). If $\Psi \Rightarrow^* \Psi'$ under a program P, there exists $\Delta :> \beta_{Cnf}(\Psi')$ such that:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta.$$

By providing an over-approximation of any reachable configuration of the concrete semantics in terms of a corresponding set of facts, the theorem can be used to prove the absence of undesired information flows of sensitive data into local registers of selected sink methods. In particular, we leverage the theorem to develop a provably sound taint analysis, based on standard ideas. Due to space constraints, we refer to the online version [7] for full details.

5. Experiments

We developed HornDroid, a static analysis tool for Android applications based on our theory. HornDroid implements a sound, fully automatic taint analysis aimed at detecting malicious information flows in Android applications. The analysis is based on a publicly available database of sources and sinks specific to the Android platform [27].

Figure 1. HornDroid Architecture

The architecture of HornDroid is shown in Figure 1. Given an Android application as an input, HornDroid generates Horn clauses defining an over-approximation of the application semantics, following the formal specification in Section 4; the choice of the underlying abstract domains and operations implements a simple taint propagation logic. The Horn clauses are encoded in the SMT-LIB format supported by many popular SMT solvers, including our choice Z3 [9]. HornDroid automatically generates analysis queries based on its database of sources and sinks[4] and the unsatisfiability of the queries is verified using the Property-Directed Reachability (PDR) engine implemented in Z3 [16]. If no query is satisfiable, no information leak from a source to a sink may occur in the analysed application.

5.1. Evaluation on DroidBench

DroidBench [2] is a set of small applications which has been proposed by the research community as a testing ground for static information flow analysis tools for

4. We use the latest and largest database available in the literature, i.e. the one used in DroidSafe [15].

Android. The current version of the benchmark (2.0) includes 120 test cases, featuring both leaky (positive) and benign (negative) examples. We tested IccTA, AmanDroid, DroidSafe and HornDroid on this benchmark, the results are summarized in the confusion matrix in Table 10, reporting the number of true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn) produced by the tools.

TABLE 10 Confusion Matrix on DroidBench

	Output	
	leaky	*benign*
	IccTA/AD/DS/HD	*IccTA/AD/DS/HD*
leaky	tp : 64 / 70 / 89 / 96	fn : 36 / 30 / 11 / 4
benign	fp : 8 / 5 / 10 / 6	tn : 11 / 14 / 9 / 13

IccTA does not detect 36 out of 100 leaky applications, AmanDroid misses 30 and DroidSafe still misses 11. Most of the leaks missed by IccTA and AmanDroid are due to flow-sensitivity and some callbacks which are not correctly detected by the analysis; as to DroidSafe, we do not have definite answers on the unsound results, given the sheer size of the project and the lack of complete documentation. HornDroid performs much better than all its competitors on DroidBench, since it only misses 4 leaky applications: all these cases are related to *implicit flows*, which are not covered by standard taint analyses (and our formal proof).

But even better, despite the strong security guarantees it provides, the analysis performed by HornDroid is not overly conservative, since it detects as potentially leaky only 6 out of 19 benign applications. We notice that 3 of these false alarms are due to flow insensitivity of the heap abstraction, one to an over-approximation of exceptions, and 2 to an over-approximated treatment of inter-app communication. Only AmanDroid is more precise, since it produces one less false positive; on the other hand, it misses many more malicious information flows than HornDroid (30 vs 4). For the sake of completeness, we report in [7] a full breakdown of the experiments on DroidBench.

The experimental results on DroidBench are summarized by a few standard statistical measures in Table 11, which highlight that soundness in HornDroid does not come at the cost of precision.

TABLE 11 Performance Measures on DroidBench

	IccTA	*AD*	*DS*	*HD*
Sensitivity	0.64	0.70	0.89	0.96
Specificity	0.58	0.74	0.47	0.68
F-Measure	0.61	0.72	0.62	0.80

Sensitivity = $tp/(tp + fn)$ $\sim$ Soundness
Specificity = $tn/(tn + fp)$ $\sim$ Precision
F-Measure = $2 * (sens * spec)/(sens + spec)$ $\sim$ Aggregate

Besides the quality of the results, also performances are important. Table 12 reports the mean and the median of the analysis times for the applications in DroidBench. As it turns out, HornDroid is one order of magnitude faster than both IccTA and AmanDroid, which in turn are one order of magnitude faster than DroidSafe. The extremely good performances of HornDroid are due to both design choices, like flow insensitivity on the activity life-cycle, and excellent support by Z3 in Horn clauses resolution.

TABLE 12 Analysis Time for DroidBench (Seconds)

	IccTA	*AD*	*DS*	*HD*
Average Analysis Time	19	11	176	1
Median Analysis Time	15	10	186	1

5.2. Evaluation on Real Applications

In order to evaluate the practicality of our analysis, we performed a test on the two largest applications available in the Google Play Top 30: the game Candy Crash Soda Saga (51.7 Mb) and the Facebook application (46.5 Mb). We ran the experiments on a server with 64 multi-thread cores and 758 Gb of memory, however the highest memory consumption by HornDroid was around 10 Gb, so it is possible to reproduce our results even on a modern commercial machine.

HornDroid found an information leak in Facebook, while Candy Crash Soda Saga appears to be secure. The analysis took around 30 minutes and 60 minutes respectively. We tested all the existing competitors on both applications, to check whether they could confirm the analysis results. Unfortunately, AmanDroid crashed just after the beginning of the analysis of Facebook, while both DroidSafe and IccTA failed to terminate within the timeout we set (2 hours). We were able instead to analyse Candy Crash Soda Saga using AmanDroid in around 50 minutes, getting an information flow. After a manual inspection, we realized this is a false positive due to the incorrect inclusion of the `onHandleIntent` method of the class `IntentService` among the possible sources of sensitive information: this is not included in more recent proposals [15], [21]. Both IccTA and DroidSafe were not able to analyse the application within 2 hours. Due to space constraints, we refer to [7] for a more comprehensive experimental evaluation on real applications.

5.3. Features and Limitations

As anticipated, the formalization in the previous sections only captures the *core* of the analysis implemented in HornDroid and establishes the soundness of its principles. The tool, however, supports more features which are needed to make the analysis scale to real applications. We detail here some important aspects of HornDroid which are not covered by our formal model and we comment on current limitations.

Android Components. Although the μ-Dalvik$_A$ model only represents activities and their life-cycle, HornDroid supports all the component types available on the Android

platform, including services, broadcast receivers and content providers [31]. The implementation of the analysis for these components does not significantly differ from the one for activities we presented in the paper, though it requires a correct modelling of their specific life-cycle.

Fragments. Fragments are used to separate the functionality of an activity among different independent subcomponents [32]. In order to support a sound analysis of fragments, HornDroid over-approximates their life-cycle by executing all the fragments along with the containing activity in a flow-insensitive way. This might lead to precision problems on real applications, but this is the simplest of the sound options, which follows the philosophy we adopted for activity analysis.

Arrays. Though the static analysis we formalized is field-insensitive on arrays, HornDroid supports a more precise treatment of array indexes. Being value-sensitive, HornDroid statically approximates which indexes of an array may be accessed at runtime: if a secret value is stored in the first position of the array, but only the second element of the array is leaked, the tool does not raise an alarm, contrarily to all the other existing tools (cf. the breakdown on the experiments in [7]).

Exceptions. HornDroid implements a conservative solution to handle exceptions, i.e., exceptions are always assumed to be thrown. A similar coarse over-approximation is implemented in FlowDroid [2]. We leave a more precise treatment of exceptions to future work: we believe that the value-sensitivity of the analysis implemented in our tool will be crucial to limit the number of false alarms for exception handling. For instance, a value-sensitive analysis can ensure that a null pointer exception is never raised at runtime, since it over-approximates the set of the possible runtime values.

Inter-app Communication. HornDroid has limited support for inter-application communication, i.e., it conservatively detects an information leak whenever an intent storing secret data is sent to another application. More precise results could be achieved by analysing all the communicating applications simultaneously, but the current implementation of HornDroid only supports the analysis of a single application. We plan to leverage existing state-of-the-art solutions to overcome this limitation [21].

Threading. HornDroid handles multithreading by assuming that threads are executed in a sequential, but arbitrary order, much in the same spirit of the callbacks defining the activity life-cycle. This is the same strategy used in FlowDroid. We conjecture, but did not prove yet, that this strategy is sound in our case, since the analysis is flow insensitive on everything except for registers, which are not shared. For flow-sensitive analysis techniques (e.g., FlowDroid), instead, this strategy is in general unsound, since it may miss potential interleavings arising due to synchronization on shared memory (e.g., static heaps). The only aspect that should be added to our static analysis is a thread pool simulation. In Java, every time the method `execute` is called on a thread, this is placed in a pool and then executed by the system by calling the runnable method `run`. Our static analysis similarly binds each invocation of `execute` to a corresponding `run` method.

Reflection. Though supporting reflection soundly is an open research problem [30], HornDroid still covers a significant fraction of common reflection cases by implementing a simple string analysis. The solution we propose is in the same spirit of DroidSafe, i.e., reflective calls which can be statically resolved are replaced by direct calls to the appropriate method. Pragmatically, however, we observed that we are able to achieve much better results than DroidSafe for the reflection cases in DroidBench.

Limitations. A comprehensive implementation of analysis stubs for method calls to the Android APIs is still lacking: we only implemented some selected stubs for our experiments, to show that our approach is feasible and practical. When a stub to an external library is missing, the tool tries to be conservative: the return value of the call is over-approximated to the top element of the corresponding abstract domain, and it is tainted whenever at least one of the arguments is tainted. Other important limitations of HornDroid are shared with existing solutions [2], [15]. First, the analysis does not capture *implicit* information flows at present. Second, the analysis does not consider *native code*: this is a point we leave as a future work, observing that SMT solving has been successfully applied in the past to C code (see, e.g., the SLAM project [3]). Third, the analysis is oblivious to the *semantics* of the information flows, i.e., it lacks any built-in declassification mechanism to qualify legitimate data flows. Since our analysis approximates data information rather than just tracking taints, however, it is in principle possible to encode expressive data-dependent declassification policies, e.g., one could define the result of an encryption as untainted only if the encryption is performed with the right key.

6. Additional Related Work

Several papers have proposed an operational semantics for Android applications by now. The first attempt is due to Chaudhuri [8], who presented a core calculus to model Android applications. Later research proposed much more concrete models: Jeon *et al.* developed μ-Dalvik, a relatively simple formal language which thoroughly models a significant fraction of the Dalvik opcodes [18]. Wognsen *et al.* presented an even richer language, which also formalises exceptions and some common uses of reflection [37]. Recently, Payet and Spoto complemented existing research by defining the first operational semantics for Android activities [26]. The semantics takes into account the event-driven behaviour of the activity lifecycle and, to some extent, the inter-component communication mechanism. Unfortunately, though, it represents only a small subset of the opcodes

available in Dalvik and just models the control flow of activities, rather than the data flows enabled by inter-component communication. Our proposal integrates [18] and [26], while providing the first accurate description of how data flows between different components of an Android application.

Cassandra [22] is, to the best of our knowledge, the only tool implementing a provably sound information flow analysis for Android applications. The analysis is based on security types: well-typed programs ensure a termination-insensitive notion of non-interference, which proves the absence of both explicit and implicit information flows. By capturing implicit flows, Cassandra provides stronger security assurances than other static analysis tools, including ours. On the other hand, the analysis implemented in Cassandra is exclusively focused on the bytecode, and it does not track information leaks enabled by the application lifecycle. Moreover, the design of Cassandra is not very practical, since it requires application developers to write security certificates, giving a typing of all fields and methods in the application. Being type-based, Cassandra does not track any static approximation of runtime values, thus making it easy for malicious developers to force an overwhelming number of false alarms. We are not aware of any experimental evaluation of Cassandra so far.

Static analyses for improving the security of Android applications are not limited to information flow control: important applications include the detection of over-privileged apps [12] and of attack surfaces for privilege escalation [6]. Finally, it is worth mentioning that also dynamic analysis of Android applications is a popular research line [11], [19], [34], [17]. Dynamic analysis is largely complementary to static analysis, since it is typically more precise, but it hardly provides full coverage of all the possible execution paths and thus is not suitable to be employed in the vetting phase of an application.

7. Conclusion

We presented HornDroid, a tool for the static analysis of Android applications based on Horn clause resolution. HornDroid is the first static analysis tool for Android that comes with a formal proof of soundness covering a large fragment of the Android ecosystem. Based on an available benchmark proposed by the community, we experimentally showed that HornDroid is much more efficient than competitors, very precise, and it is the first tool to detect all the existing (explicit) information flows.

Our approach makes it easy to fine-tune the static analysis, since one has just to modify the Horn clause generation algorithm, while the resolution can be performed using off-the-shelf SMT solvers, thus leveraging the tremendous progress in this field. In order to facilitate future extensions by the community, we make our tool freely available, as source code as well as through a web interface [7].

We are currently working on the verification of CTL formulas, by using a recently developed encoding into Horn clauses [4]. Furthermore, we plan to extend our tool in order to check non-interference properties and prove the absence of implicit information flows. We would also like to identify sound solutions to implement flow-sensitivity for heap locations, thus making our static analysis even more precise. For further boosting the precision, we intend to integrate in HornDroid a recently developed string analysis engine for Z3 [33]. Finally, we intend to extend the formal model and the proof of soundness in order to cover the entire analysis.

References

[1] "The Java Language Specification," https://docs.oracle.com/javase/specs/jls/se7/html/, last accessed on February 2013.

[2] 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*. ACM, 2014, pp. 259–269.

[3] T. Ball, V. Levin, and S. K. Rajamani, "A Decade of Software Model Checking with SLAM," pp. 68–76, 2011.

[4] T. A. Beyene, M. Brockschmidt, and A. Rybalchenko, "CTL+FO Verification As Constraint Solving," in *SPIN*. ACM, 2014, pp. 101–104.

[5] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "Program Verification as Satisfiability Modulo Theories," in *SMT*. ACM, 2012, pp. 3–11.

[6] M. Bugliesi, S. Calzavara, and A. Spanò, "Lintent: Towards security type-checking of Android applications," in *FMOODS/FORTE*, 2013, pp. 289–304.

[7] S. Calzavara, I. Grishchenko, and M. Maffei, "Full version of the present submission and HornDroid implementation," available online at https://www.sps.cs.uni-saarland.de/horndroid/.

[8] A. Chaudhuri, "Language-Based Security on Android," in *PLAS*. ACM, 2009, pp. 1–7.

[9] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*. Springer-Verlag, 2008, pp. 337–340.

[10] I. Dillig, T. Dillig, and A. Aiken, "Precise Reasoning for Programs Using Containers," in *POPL*. ACM, 2011, pp. 187–200.

[11] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014.

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

[13] S. Fink and J. Dolby, "WALA – The TJ Watson Libraries for Analysis," 2012. [Online]. Available: http://wala.sf.net/

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in *TRUST*. Springer-Verlag, 2012, pp. 291–307.

[15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *NDSS*. IEEE, 2015.

[16] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT*. Springer-Verlag, 2012, pp. 157–171.

[17] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications," in *CCS*. ACM, 2011, pp. 639–652.

[18] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic Execution for Dalvik Bytecode," University of Maryland, Tech. Rep., 2012.

[19] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-Time Enforcement of Information-Flow Properties on Android - (Extended Abstract)," in *ESORICS*. ACM, 2013, pp. 775–792.

[20] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *MoST*, 2012.

[21] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *ICSE*. IEEE Press, 2015, pp. 280–291.

[22] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "Cassandra: Towards a Certifying App Store for Android," in *SPSM@CCS*. ACM, 2014, pp. 93–104.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *CCS*. ACM, 2012, pp. 229–240.

[24] C. Mann and A. Starostin, "A Framework for Static Detection of Privacy Leaks in Android Applications," in *SAC*. ACM, 2012, pp. 1457–1462.

[25] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, 1999.

[26] É. Payet and F. Spoto, "An Operational Semantics for Android Activities," in *PEPM*. ACM, 2014, pp. 121–132.

[27] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," in *NDSS*, 2014.

[28] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick Your Contexts Well: Understanding Object-Sensitivity," in *POPL*. ACM, 2011, pp. 17–30.

[29] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick Your Contexts Well: Understanding Object-sensitivity," in *POPL*. ACM, 2011, pp. 17–30.

[30] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "More Sound Static Handling of Java Reflection," Tech. Rep., 2014.

[31] The Android Developers Guide, "App Components," available online at http://developer.android.com/guide/components/index.html.

[32] ——, "Fragments," available online at http://developer.android.com/guide/components/fragments.html.

[33] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications," in *CCS*. ACM, 2014, pp. 1232–1243.

[34] O. Tripp and J. Rubin, "A Bayesian Approach to Privacy Enforcement in Smartphones," in *USENIX*. USENIX, 2014, pp. 175–190.

[35] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" in *CC*. Springer-Verlag, 2000, pp. 18–34.

[36] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *CCS*. ACM, 2014, pp. 1329–1341.

[37] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen, "Formalisation and Analysis of Dalvik Bytecode," *Sci. Comput. Program.*, vol. 92, pp. 25–55, 2014.

[38] Z. Yang and M. Yang, "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis," in *WCSE*. IEEE, 2012, pp. 101–104.

[39] Z. Zhao and F. C. C. Osorio, "TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the use of static analysis taint tracking," in *MALWARE*. IEEE, 2012, pp. 135–143.