A Sound Flow-Sensitive Heap Abstraction for the Static Analysis of Android Applications

Stefano Calzavara*, Ilya Grishchenko†, Adrien Koutsos‡, Matteo Maffei†
* Università Ca' Foscari Venezia † TU Wien, ‡ LSV, CNRS, ENS Paris-Saclay

***Abstract*—The present paper proposes the first static analysis for Android applications which is both flow-sensitive on the heap abstraction and provably sound with respect to a rich formal model of the Android platform. We formulate the analysis as a set of Horn clauses defining a sound over-approximation of the semantics of the Android application to analyse, borrowing ideas from recency abstraction and extending them to our concurrent setting. Moreover, we implement the analysis in HornDroid, a state-of-the-art information flow analyser for Android applications. Our extension allows HornDroid to perform strong updates on heap-allocated data structures, thus significantly increasing its precision, without sacrificing its soundness guarantees. We test our implementation on DroidBench, a popular benchmark of Android applications developed by the research community, and we show that our changes to HornDroid lead to an improvement in the precision of the tool, while having only a moderate cost in terms of efficiency. Finally, we assess the scalability of our tool to the analysis of real applications.**

I. Introduction

Android is today the most popular operating system for mobile phones and tablets, and it boasts the largest application market among all its competitors. Though the huge number of available applications is arguably one of the main reasons for the success of Android, it also poses an important security challenge: there are way too many applications to ensure that they go through a timely and thorough security vetting before their publication on the market. Automated analysis tools thus play a critical role in ensuring that security verification does not fall behind with respect to the release of malicious (or buggy) applications.

There are many relevant security concerns for Android applications, e.g., privilege escalation [12], [5] and component hijacking [26], but the most important challenge in the area is arguably *information flow control*, since Android applications are routinely granted access to personal information and other sensitive data stored on the device where they are installed. To counter the threats posed by malicious applications, the research community has proposed a plethora of increasingly sophisticated (static) information flow control frameworks for Android [41], [42], [27], [14], [22], [3], [40], [15], [7]. Despite all this progress, however, none of these static analysis tools is able to properly reconcile soundness and precision in its treatment of heap-allocated data structures.

A. Soundness vs. Precision in Android Analyses

Designing a static analysis for Android applications which is both sound and precise on the heap abstraction is very challenging, most notably because the Android ecosystem is highly concurrent, featuring multiple components running in the same application at the same time and sharing part of the heap. More complications come from the scheduling of these components, which is user-driven, e.g., via button clicks, and thus statically unknown. This means that it is hard to devise precise *flow-sensitive* heap abstractions for Android applications without breaking their soundness. Indeed, most existing static analysers for Android applications turn out to be unsound and miss malicious information leaks ingeniously hidden in the control flow: for instance, Table I shows a leaky code snippet that cannot be detected by FlowDroid [3], a state-of-the-art taint tracker for Android applications[1].

```
1 public class Leaky extends Activity {
2   Storage st = new Storage();
3   Storage st2 = new Storage();
4   onRestart() { st2 = st; }
5   onResume() { st2.s = getDeviceId(); }
6   onPause() { send(st.s, "http://www.myapp.com/"); }
7 }
```

TABLE I
A Subtle Information Leak

Assume that the `Storage` class has only one field `s` of type `String`, populated with the empty string by its default constructor. The activity class `Leaky` has two fields `st` and `st2` of type `Storage`. A leak of the device id may be performed in three steps. First, the activity is stopped and then restarted: after the execution of the `onRestart()` callback, `st2` becomes an alias of `st`. Then, the activity is paused and resumed. As a result, the execution of the `onPause()` callback communicates the empty string over the Internet, while the `onResume()` callback stores the device id in `st2` and thus in `st` due to aliasing. Finally, the activity is paused again and the device id is leaked by `onPause()`.

HornDroid [7] is the only provably sound static analyser for Android applications to date and, as such, it correctly deals with the code snippet in Table I. In order to retain soundness, however, HornDroid is quite conservative on the prediction of the control flow of Android applications and implements a *flow-insensitive* heap abstraction by computing just one static over-approximation of the heap, which is proved to be correct at all reachable program points. This is a significant

[1]Android applications are written in Java and compiled to bytecode run by a register-based virtual machine (Dalvik). Most static analysis tools for Android analyse Dalvik bytecode, but we present our examples using a Java-like language to improve readability.

limitation of the tool, since it prevents *strong updates* [23] on heap-allocated data structures and thus negatively affects the precision of the analysis. Concretely, to understand the practical import of this limitation, consider the Java code snippet in Table II.

```
1 public class Anon extends Activity {
2   Contact[] m = new Contact[]();
3   onStart() {
4     for (int i = 0; i < contacts.length(); i++) {
5       Contact c = contacts.getContact(i);
6       c.phone = anonymise(c.phone);
7       m[i] = c;
8     }
9     send(m, "http://www.cool-apps.com/");
10  }
11 }
```

TABLE II
ANONYMIZING CONTACT INFORMATION

This code reads the contacts stored on the phone, but then calls the `anonymise` method at line 6 to erase any sensitive information (like phone numbers) before sending the collected data on the Internet. Though this code is benign, HornDroid raises a false alarm, since the field `c.phone` stores sensitive information after line 5 and strong updates of object fields are not allowed by the static analysis implemented in the tool.

B. Contributions

In the present paper we make the following contributions:

1) we extend an operational semantics for a core fragment of the Android ecosystem [7] with multi-threading and exception handling, in order to provide a more accurate representation of the control flow of Android applications;
2) we present the first static analysis for Android applications which is both flow-sensitive on the heap abstraction and provably sound with respect to the model above. Our proposal borrows ideas from *recency abstraction* [4] in order to hit a sweet spot between precision and efficiency, extending it for the first time to a concurrent setting;
3) we implement our analysis as an extension of HornDroid [7]. This extension allows HornDroid to perform strong updates on heap-allocated data structures, thus significantly increasing the precision of the tool;
4) we test our extension of HornDroid against DroidBench, a popular benchmark proposed by the research community [3]. We show that our changes to HornDroid lead to an improvement in the precision of the tool, while having only a moderate cost in terms of efficiency. We also discuss analysis results for 64 real applications to demonstrate the scalability of our approach. Our tool and more details on the experiments are available online [1].

II. DESIGN AND KEY IDEAS

A. Our Proposal

Our proposal starts from the pragmatic observation that statically predicting the control flow of an Android application is daunting and error-prone [15]. For this reason, our analysis simply assumes that all the activities, threads and callbacks of the application to analyse are concurrently executed under an interleaving semantics[2]. (In the following paragraphs, we just refer to threads for brevity.)

The key observation to recover precision despite this conservative assumption is that the runtime behaviour of a given thread can only invalidate the static approximation of the heap of another thread whenever the two threads share memory. This means that the heap of each thread can be soundly analysed in a flow-sensitive fashion, as long as the thread runs isolated from all other threads. Our proposal refines this intuition and achieves a much higher level of precision by using two separate static approximations of the heap: a *flow-sensitive abstract heap* and a *flow-insensitive abstract heap*.

Abstract objects on the flow-sensitive abstract heap approximate concrete objects which are guaranteed to be local to a single thread (not shared). Moreover, these abstract objects always approximate exactly one concrete object, hence it is sound to perform *strong updates* on them. Abstract objects on the flow-insensitive abstract heap, instead, approximate either (1) one concrete object which may be shared between multiple threads, or (2) multiple concrete objects, e.g., produced by a loop. Thus, abstract objects on the flow-insensitive abstract heap only support *weak updates* to preserve soundness. In case (1), this is a consequence of the analysis conservatively assuming the concurrent execution of all the threads and the corresponding loss of precision on the control flow. In case (2), this follows from the observation that only one of the multiple concrete objects represented by the abstract object is updated at runtime, but the updated abstraction should remain sound for all the concrete objects, including those which are not updated. The analysis moves abstract objects from the flow-sensitive abstract heap to its flow-insensitive counterpart when one of the two invariants of the flow-sensitive abstract heap may be violated: this mechanism is called *lifting*.

Technically, the analysis identifies heap-allocated data structures using their allocation site, like most traditional abstractions [32], [17], [23], [21]. Unlike these, however, each allocation site λ is bound to *two* distinct abstract locations: $\mathsf{FS}(\lambda)$ and $\mathsf{NFS}(\lambda)$. We use $\mathsf{FS}(\lambda)$ to access the flow-sensitive abstract heap and $\mathsf{NFS}(\lambda)$ to access the flow-insensitive abstract heap. The abstract location $\mathsf{FS}(\lambda)$ contains the abstraction of the *most-recently-allocated* object created at λ, provided that this object is *local* to the creating thread. Conversely, the abstract location $\mathsf{NFS}(\lambda)$ contains a sound abstraction of all the other objects created at λ.

Similar ideas have been proposed in *recency abstraction* [4], but standard recency abstraction only applies to sequential programs, where it is always sound to perform strong updates on the abstraction of the most-recently-allocated object. Our analysis, instead, operates in a concurrent setting and assumes that all the threads are concurrently executed under an interleaving semantics. As we anticipated, this means that, if a

[2]We are aware of the fact that the Java Memory Model allows more behaviours than an interleaving semantics (see [24] for a formalisation), but since its connections with Dalvik depend on the Android version and its definition is very complicated, in this work we just consider an interleaving semantics for simplicity.

pointer may be shared between different threads, performing strong updates on the abstraction of the object indexed by the pointer would be unsound. Our analysis allows strong updates without sacrificing soundness by statically keeping track of a set of pointers which are known to be local to a single thread: only the abstractions of the most-recently-allocated objects indexed by these pointers are amenable for strong updates.

B. Examples

By being conservative on the execution order of callbacks, our analysis is able to soundly analyse the leaky example of Table I. We recall it in Table III, where we annotate it with a simplified version of the facts generated by the analysis: the heap fact H provides a *flow-insensitive* heap abstraction, while the Sink fact denotes communication to a sink. We use line numbers to identify allocation sites and to index the heap abstractions.

```
1 public class Leaky extends Activity {
  H(1, {|Leaky; st ↦ NFS(2), st2 ↦ NFS(3)|})
  // flow-insensitivity on activity object
2   Storage st = new Storage();
  H(2, {|Storage; s ↦ ""|}) // after the constructor
3   Storage st2 = new Storage();
  H(3, {|Storage; s ↦ ""|}) // after the constructor
4   onRestart() { st2 = st; }
  H(1, {|Leaky; st ↦ NFS(2), st2 ↦ NFS(2)|}) // aliasing
5   onResume() { st2.s = getDeviceId(); }
  H(2, {|Storage; s ↦ id|}) ∧ H(3, {|Storage; s ↦ id|})
  // due to flow-insensitivity on activity object
6   onPause() { send(st.s, "http://www.myapp.com/");
  Sink("") ∧ Sink(id) // the leak is detected
7   }
8 }
```

TABLE III
A SUBTLE INFORMATION LEAK (DETECTED)

In our analysis, activity objects are always abstracted in a flow-insensitive way, which is crucial for soundness, since we do not predict the execution order of their callbacks. When the activity is created, an abstract flow-insensitive heap fact $\mathsf{H}(1, \{|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(3)|\})$ is introduced, and two facts $\mathsf{H}(2, \{|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\})$ and $\mathsf{H}(3, \{|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\})$ abstract the objects pointed by the activity fields `st` and `st2`. Then the life-cycle events are abstracted: the `onRestart` method performs a weak update on the activity object, adding a fact $\mathsf{H}(1, \{|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(2)|\})$ which tracks aliasing; after the `onResume` method, `st` can thus point to two possible objects, as reflected by the abstract flow-insensitive heap facts generated at line 2 and at line 5. Since the latter fact tracks a sensitive value in the field `s`, the leak is caught in `onPause`.

Our analysis can also precisely deal with the benign example of Table II thanks to recency abstraction. We show a simplified version of the facts generated by the analysis in Table IV. If our static analysis only used a traditional allocation-site abstraction, the benefits of flow-sensitivity would be voided by the presence of the "for" loop in the code. Indeed, the allocation site of `c` would need to identify all the concrete objects allocated therein, hence a traditional static analysis could not perform strong updates on `c.phone` without breaking soundness and would raise a false alarm on the code.

```
1 public class Anon extends Activity {
  H(1, {|Anon; m ↦ NFS(2)|})
  // flow-insensitivity on activity object
2   Contact[] m = new Contact[];
  H(2, []) // new empty array is created
3   onStart() {
  LState3(c ↦ null; 5 ↦ ⊥)
  // no allocated contact at location 5 yet
4     for (int i = 0; i < contacts.length(); i++) {
  LState4(c ↦ null; 5 ↦ ⊥) ∧ LState4(c ↦ NFS(5); 5 ↦ ⊥)
  // loop invariant (see below)
5       Contact c = contacts.getContact(i);
  LState5(c ↦ FS(5); 5 ↦ oc) // flow-sensitivity
6       c.phone = anonymise(c.phone);
  LState6(c ↦ FS(5); 5 ↦ oc{phone ↦ ""}) // strong update
7       m[i] = c;
  LState7(c ↦ NFS(5); 5 ↦ ⊥) ∧ H(5, oc{phone ↦ ""}) ∧
  H(2, [NFS(5)]) // lifting is performed
8     }
9     send(m, "http://www.cool-apps.com/");
  Sink([oc{phone ↦ ""}]) // no leak is detected
10  }
11 }
```

TABLE IV
ANONYMIZING CONTACT INFORMATION (ALLOWED)

The local state fact LState_{pp} provides a flow-sensitive abstraction of the state of the registers and the heap at program point *pp*. Recall that activity objects are always abstracted in a flow-insensitive fashion, therefore the `Contact` array `m` is also abstracted by a flow-insensitive heap fact $\mathsf{H}(2, [])$. At each loop iteration, our static analysis abstracts the most-recently-allocated `Contact` object at line 5 in a flow-sensitive fashion. This is done by putting the abstract flow-sensitive location $\mathsf{FS}(5)$ in `c` and by storing the abstraction of the `Contact` object o_c in the flow-sensitive local state abstraction LState_5, using its allocation site 5 as a key. This allows us to perform a strong update on the `c.phone` field at line 6, overwriting the private information with a public one. At line 7 the program stores the public object in the array `m`, which is abstracted by a flow-insensitive heap fact: to preserve soundness, the flow-sensitive abstraction of o_c is *lifted* (downgraded) to a flow-insensitive abstraction by generating a flow-insensitive heap fact $\mathsf{H}(5, o_c[\texttt{phone} \mapsto \texttt{""}])$ and by changing the abstraction of `c` from $\mathsf{FS}(5)$ to $\mathsf{NFS}(5)$. We then perform a weak update on the array stored in `m` by generating a flow-insensitive heap fact $\mathsf{H}(2, [\mathsf{NFS}(5)])$. Thanks to the previous strong update, however, the end result is that `m` only stores public information at the end of the loop and no leak is detected.

III. CONCRETE SEMANTICS

Our static analysis is defined on top of an extension of μ-Dalvik$_A$, a formal model of a core fragment of the Android ecosystem [7]. It includes the main bytecode instructions of Dalvik, the register-based virtual machine running Android applications, and a few important API methods. Moreover, it captures the life-cycle of the most common and complex application components (*activities*), as well as inter-component communication based on asynchronous messages (*intents*, with

a dictionary-like structure). Our extension of μ-Dalvik$_A$ adds two more ingredients to the model: *multi-threading* and *exceptions*, which are useful to get a full account of the control flow of Android applications. For space reasons, the presentation focuses on a relatively high-level overview of our extensions: the formal details, including the full operational semantics, can be found in the long version of the paper [6].

A. Basic Syntax

We write $(r_i)^{i \leq n}$ to denote the sequence $r_1, \ldots, r_n$. When the length of the sequence is unimportant, we simply write r^*. Given a sequence r^*, r_j stands for its j-th element and $r^*[j \mapsto r']$ denotes the sequence obtained from r^* by substituting its j-th element with r'. We let $k_i \mapsto v_i$ denote a key-value binding and we represent partial maps using a sequence of key-value bindings $(k_i \mapsto v_i)^*$, where all the keys k_i are pairwise distinct; the order of the keys in a partial map is immaterial.

We introduce in Table V a few basic syntactic categories. A program P is a sequence of classes. A class $\texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$ consists of a name c, a super-class c', a sequence of implemented interfaces c^*, a sequence of fields fld^*, and a sequence of methods mtd^*. A method $m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$ consists of a name m, the type of its arguments τ^*, the return type τ, and a sequence of statements st^* defining the method body; the syntax of statements is explained below. The integer n on top of the arrow declares how many registers are used by the method. Observe that field declarations $f : \tau$ include the type of the field. A left-hand side lhs is either a register r, an array cell $r_1[r_2]$, an object field $r.f$, or a static field $c.f$, while a right-hand side rhs is either a left-hand side lhs or a primitive value prim.

P	::=	cls^*
cls	::=	$\texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$
τ_{prim}	::=	$\texttt{bool} \mid \texttt{int} \mid \ldots$
τ	::=	$c \mid \tau_{\mathit{prim}} \mid \texttt{array}[\tau]$
fld	::=	$f : \tau$
mtd	::=	$m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$
lhs	::=	$r \mid r[r] \mid r.f \mid c.f$
prim	::=	$\mathit{true} \mid \mathit{false} \mid \ldots$
rhs	::=	$\mathit{lhs} \mid \mathit{prim}$

TABLE V
BASIC SYNTACTIC CATEGORIES

Table VI reports the syntax of selected statements, along with a brief intuitive explanation of their semantics. Observe that statements do not operate directly on values, but rather on the content of the registers of the Dalvik virtual machine. The extensions with respect to [7] are in bold and are discussed in more detail in the following. Some of the next definitions are dependent on a program P, but we do not make this dependency explicit to keep the notation more concise.

B. Local Reduction

a) Notation: Table VII shows the main semantic domains used in the present section. We let p range over pointers from a countable set *Pointers*. A program point pp is a triple c, m, pc including a class name c, a method name m and a program counter pc (a natural number identifying a specific statement of the method). Annotations λ are auxiliary information with no semantic import, their use in the static analysis is discussed in Section IV. A location ℓ is an annotated pointer p_λ and a value v is either a primitive value or a location.

A *local state* $L = \langle pp \cdot u^* \cdot st^* \cdot R \rangle$ stores the state information of an invoked method, run by a given thread or activity. It is composed of a program point pp, identifying the currently executed statement; the method calling context u^*, which keeps track of the method arguments and is only used in the static analysis; the method body st^*, defining the method implementation; and a register state R, mapping registers to their content. Registers are local to a given method invocation.

A *local state list* $L^\#$ is a list of local states. It is used to keep track of the state information of all the methods invoked by a given thread or activity. The *call stack* α is modeled as a local state list $L^\#$, possibly qualified by the $\texttt{AbNormal}(\cdot)$ modifier if the thread or activity is recovering from an exception.

Coming to memory, we define the *heap* H as a partial map from locations to *memory blocks*. There are three types of memory blocks in the formalism: objects, arrays and intents. An *object* $o = \{\!|c; (f_\tau \mapsto v)^*|\!\}$ stores its class c and a mapping between fields and values. Fields are annotated with their type, which is typically omitted when unneeded. An *array* $a = \tau[v^*]$ contains the type τ of its elements and the sequence of the values v^* stored into it. An *intent* $i = \{\!|@c; (k \mapsto v)^*|\!\}$ is composed by a class name c, identifying the intent recipient, and a sequence of key-value bindings $(k \mapsto v)^*$, defining the intent payload (a dictionary). The *static heap* S is a partial map from static fields to values.

Finally, we have *local configurations* $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$, representing the full state of a specific activity or thread. They include a location ℓ, pointing to the corresponding activity or thread object; a call stack α; a pending activity stack π, which is a list of intents keeping track of all the activities that have been started; a pending thread stack γ, which is a list of pointers to the threads which have been started; a heap H, storing memory blocks; and a static heap S, storing the values of static fields.

We use several substitution notations in the reduction rules, with an obvious meaning. The only non-standard notations are Σ^+, which stands for Σ where the value of pc is replaced by $pc + 1$ in the top-most local state of the call stack, and the substitution of registers $\Sigma[r_d \mapsto u]$, which sets the value of the register r_d to u in the top-most local state of the call stack. This reflects the idea that the computation is performed on the local state of the last invoked method.

b) Local Reduction Relation: The *local reduction* relation $\Sigma \rightsquigarrow \Sigma'$ models the evolution of a local configuration Σ into a new local configuration Σ' as the result of a computation step. The definition of the local reduction relation uses two auxiliary relations:

- $\Sigma[\![\mathit{rhs}]\!]$, which evaluates a right-hand side expression rhs in the local configuration Σ;
- $\Sigma, st \Downarrow \Sigma'$, which executes the statement st on the local configuration Σ to produce Σ'.

The simplest rule defining a local reduction step $\Sigma \rightsquigarrow \Sigma'$ just fetches the next statement st to run and performs a look-up

$st ::=$			
`goto` pc	unconditionally jump to program counter pc	`invoke` r_o m r^*	invoke method m of the object in r_o with args r^*
`if`$_{\otimes}$ r_1 r_2 `then` pc	jump to program counter pc if $r_1 \otimes r_2$	`return`	get the value of the special return register r_{res}
`move` *lhs rhs*	move *rhs* into *lhs*	`newintent` r_i c	put a pointer to a new intent for class c in r_i
`unop`$_{\odot}$ r_d r_s	compute $\odot r_s$ and put the result in r_d	`put-extra` r_i r_k r_v	bind the value of r_v to key r_k of the intent in r_i
`binop`$_{\oplus}$ r_d r_1 r_2	compute $r_1 \oplus r_2$ and put the result in r_d	`get-extra` r_i r_k τ	get the τ-value bound to key r_k of the intent in r_i
`new` r_d c	put a pointer to a new object of class c in r_d	`start-act` r_i	start a new activity by sending the intent in r_i
`newarray` r_d r_l τ	put a pointer to a new τ-array of length r_l in r_d	**`start-thread`** r_t	start the thread in r_t
`throw` r_e	throw the exception stored in r_e	**`interrupt`** r_t	interrupt the thread in r_t
`move-except` r_e	store a pointer to the last thrown exception in r_e	**`join`** r_t	join the current thread with the thread in r_t

TABLE VI
SYNTAX AND INFORMAL SEMANTICS OF SELECTED STATEMENTS

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

TABLE VII
SEMANTIC DOMAINS FOR LOCAL REDUCTION

on the auxiliary relation $\Sigma, st \Downarrow \Sigma'$. Formally, assuming a function *get-stm*(Σ) fetching the next statement based on the program counter of the top-most local state in Σ, we have:

$$\frac{\Sigma, \textit{get-stm}(\Sigma) \Downarrow \Sigma'}{\Sigma \rightsquigarrow \Sigma'} \quad \text{(R-NEXTSTM)}$$

We show a subset of the new local reduction rules added to μ-Dalvik$_A$ in Table VIII and we explain them below.

c) Exception Rules: In Dalvik, method bodies can contain special annotations for exception handling, specifying which exceptions are caught and where, as well as the program counter of the corresponding exception handler (handlers are part of the method body). In our formalism, we assume the existence of a partial map $\mathsf{ExcptTable}(pp, c) = pc$ which provides, for all program points pp where exceptions can be thrown and for all classes c extending the Throwable interface, the program counter pc of the corresponding exception handler. If no handler exists, then $\mathsf{ExcptTable}(pp, c) = \bot$. Moreover, all local states contain a special register r_{excpt} that is only accessed by the exception handling rules: this stores the location of the last thrown exception.

An exception object stored in r_e can be thrown by the statement `throw` r_e using rule (R-THROW): it checks that r_e contains the location of a (throwable) object, stores this location into the register r_{excpt} and moves the local configuration into an abnormal state. After entering an abnormal state, there are two possibilities: if there exists an handler for the thrown exception, we exit the abnormal state and jump to the program counter of the exception handler using rule (R-CAUGHT); otherwise, the exception is thrown back to the method caller using rule (R-UNCAUGHT). Finally, the location of the last thrown exception object can be copied from the register r_{excpt} into the register r_e by the statement `move-except` r_e, as formalized by rule (R-MOVEEXCEPTION)

d) Thread Rules: Our formalism covers the core methods of the Java Thread API [18]: they enable thread spawning and thread communication by means of interruptions and synchronizations. Rule (R-STARTTHREAD) models the statement `start-thread` r_t: it allows a thread to be started by simply pushing the location of the thread object stored in r_t on the pending thread stack. The actual execution of the thread is left to the virtual machine, which will spawn it at an unpredictable point in time, as we discuss in the next section. The statement `interrupt` r_t sets the interrupt field (named inte) of the thread object whose location is stored in r_t to $true$, as formalized by rule (R-INTERRUPTTHREAD). We now describe the semantics of thread synchronizations. If the thread t' calling `join` r_t was not interrupted at some point, rule (R-JOINTHREAD) checks whether the thread whose location is stored in r_t has finished; if this is the case, it resumes the execution of t', otherwise t' remains stuck. If instead t' was interrupted before calling `join` r_t, rule (R-INTERRUPTJOIN) performs the following operations: the inte field of t' is reset to $false$, an IntExcpt exception is thrown (this creates a new exception object) and the local configuration enters an abnormal state.

C. Global Reduction

a) Notation: Table IX introduces the main semantic domains used in the present section. First, we assume the existence of a set of activity states *ActStates*, which is used to model the Android activity life-cycle (see [31]). Then we have two kinds of *frames*, modeling running processes. An *activity frame* $\varphi = \langle \ell, s, \pi, \gamma, \alpha \rangle$ describes the state of an activity: it includes a location ℓ, pointing to the activity object; the activity state s; a pending activity stack π, representing other activities started by the activity; a pending thread stack γ, representing threads spawned by the activity; and a call stack α. A *thread frame* $\psi = \langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle$ describes a running thread: it includes a location ℓ, pointing to the activity object that started the thread; a location ℓ' pointing to the

$$\text{(R-THROW)}\quad \frac{\ell = \Sigma[\![r_e]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\}}{\Sigma, \texttt{throw}\ r_e \Downarrow \Sigma[\alpha \mapsto \texttt{AbNormal}(\alpha)][r_{\mathsf{excpt}} \mapsto \ell]}$$

$$\text{(R-CAUGHT)}\quad \frac{\ell = \Sigma_A[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \mathsf{ExcptTable}(c, m, pc, c') = pc' \qquad \alpha_c = \langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle :: \alpha'}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \alpha_c]}$$

$$\text{(R-UNCAUGHT)}\quad \frac{\ell = \Sigma_A[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \mathsf{ExcptTable}(c, m, pc, c') = \bot}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \texttt{AbNormal}(\alpha')][r_{\mathsf{excpt}} \mapsto \ell]}$$

$$\text{(R-MOVEEXCEPTION)}\quad \frac{\ell = \Sigma[\![r_{\mathsf{excpt}}]\!]}{\Sigma, \texttt{move-except}\ r_e \Downarrow \Sigma^+[r_e \mapsto \ell]}$$

$$\text{(R-STARTTHREAD)}\quad \frac{\ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \gamma' = \ell :: \gamma}{\Sigma, \texttt{start-thread}\ r_t \Downarrow \Sigma^+[\gamma \mapsto \gamma']}$$

$$\text{(R-INTERRUPTTHREAD)}\quad \frac{\ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \_|\!\} \qquad H' = H[\ell \mapsto \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \mathit{true}|\!\}]}{\Sigma, \texttt{interrupt}\ r_t \Downarrow \Sigma^+[H \mapsto H']}$$

$$\text{(R-JOINTHREAD)}\quad \frac{H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathit{false}|\!\} \qquad \ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{finished} \mapsto \mathit{true}|\!\}}{\Sigma, \texttt{join}\ r_t \Downarrow \Sigma^+}$$

$$\text{(R-INTERRUPTJOIN)}\quad \frac{H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathit{true}|\!\} \qquad o = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathit{false}|\!\} \qquad p_{c,m,pc} \notin \mathit{dom}(H) \qquad H' = H, p_{c,m,pc} \mapsto \{\!|\mathsf{IntExcpt};|\!\} \qquad \alpha_c = \texttt{AbNormal}(\alpha[r_{\mathsf{excpt}} \mapsto p_{c,m,pc}])}{\Sigma, \texttt{join}\ r_t \Downarrow \Sigma[\alpha \mapsto \alpha_c, H \mapsto H'[\ell_r \mapsto o]]}$$

Convention: let $\Sigma = \ell_r \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha'$ and $\Sigma_A = \ell_r \cdot \alpha_A \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha_A = \texttt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha')$.

TABLE VIII
SMALL STEP SEMANTICS OF EXTENDED μ-DALVIK$_A$ - EXCERPT

thread object; a pending activity stack π, representing activities started by the thread; a pending thread stack γ, representing other threads spawned by the thread; and a call stack α.

Activity frames are organized in an *activity stack* Ω, containing all the running activities; one of the activities may be singled out as *active*, represented by an underline, and it is scheduled for execution. We assume that each Ω contains at most one underlined activity frame. Thread frames, instead, are organized in a *thread pool* Ξ, containing all the running threads. A *configuration* $\Psi = \Omega \cdot \Xi \cdot H \cdot S$ includes an activity stack Ω, a thread pool Ξ, a heap H and a static heap S. It represents the full state of an Android application.

Activity states	s	$\in$	*ActStates*
Activity frames	φ	::=	$\langle \ell, s, \pi, \gamma, \alpha\rangle \mid \underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}$
Activity stacks	Ω	::=	$\varphi \mid \varphi :: \Omega$
Thread frames	ψ	::=	$\langle\!\langle \ell, \ell', \pi, \gamma, \alpha\rangle\!\rangle$
Thread pools	Ξ	::=	$\emptyset \mid \psi :: \Xi$
Configurations	Ψ	::=	$\Omega \cdot \Xi \cdot H \cdot S$

TABLE IX
SEMANTIC DOMAINS FOR GLOBAL REDUCTION

b) Global Reduction Relation: The *global reduction* relation $\Psi \Rightarrow \Psi'$ models the evolution of a configuration Ψ into a new configuration Ψ', either by executing a statement in a thread or activity according to the local reduction rules, or as the result of processing life-cycle events of the Android platform, including user inputs, system callbacks, inter-component communication, etc.

Before presenting the global reduction rules, we define a few auxiliary notions. First, we let *lookup* be the function such that $\mathit{lookup}(c, m) = (c', st^*)$ iff c' is the class obtained when performing dispatch resolution of the method m on an object of type c and st^* is the corresponding method body. Then, we assume a function *sign* such that $\mathit{sign}(c, m) = \tau^* \xrightarrow{n} \tau$ iff there exists a class cls_i such that $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*, m : \tau^* \xrightarrow{n} \tau\ \{st^*\}\}$. Finally, we let a *successful* call stack be the call stack of an activity or thread which has completed its computation, as formalized by the following definition.

Definition 1 *A call stack α is* successful *if and only if $\alpha = \langle pp \cdot u^* \cdot \texttt{return} \cdot R\rangle :: \varepsilon$ for some pp, u^* and R. We let $\overline{\alpha}$ range over successful call stacks.*

The core of the global reduction rules are taken from [7], extended with a few simple rules used, e.g., to manage the thread pool. The main new rules are given in Table X and the full set can be found in the long version [6]. We start by describing rule (A-THREADSTART), which models the starting of a new thread by some activity. Let ℓ' be a pointer to a pending thread spawned by an activity identified by the pointer ℓ, the rule instantiates a new thread frame $\psi = \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \alpha'\rangle\!\rangle$ with empty pending activity stack and empty pending thread stack, executing the run method of the thread object referenced by ℓ'. We then have two other rules: rule (T-REDUCE) allows the reduction of any thread in the thread pool, using the reduction relation for local configurations; rule (T-KILL) allows the system to remove a thread which has finished its computations, by checking that its call stack is successful.

IV. ABSTRACT SEMANTICS

Our analysis takes as input a program P and generates a set of Horn clauses $(\!|P|\!)$ that over-approximate the concrete semantics of P. We can then use an automated theorem prover such as Z3 [28] to show that $(\!|P|\!)$, together with a set of facts Δ over-approximating the initial state of the program, does not entail a formula ϕ representing the reachability of some undesirable program state (e.g., leaking sensitive information). By the over-approximation, the unsatisfiability of the formula ensures that also P does not reach such a program state.

(A-THREADSTART)

$$\frac{\varphi = \underline{\langle \ell, s, \pi, \gamma :: \ell' :: \gamma', \alpha \rangle} \qquad \varphi' = \underline{\langle \ell, s, \pi, \gamma :: \gamma', \alpha \rangle} \qquad \psi = \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle \qquad H(\ell') = \{\!|c'; (f \mapsto v)^*|\!\}}{\mathit{lookup}(c', \mathsf{run}) = (c'', st^*) \qquad \mathit{sign}(c'', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void} \qquad \alpha' = \langle c'', \mathsf{run}, 0 \cdot \ell' \cdot st^* \cdot (r_k \mapsto \mathbf{0})^{k \leq loc}, r_{loc+1} \mapsto \ell' \rangle}{\Omega :: \varphi :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \psi :: \Xi \cdot H \cdot S}$$

(T-REDUCE)

$$\frac{\ell_t \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell_t \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi, \gamma, \alpha \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H' \cdot S'}$$

(T-KILL)

$$\frac{H(\ell') = \{\!|c; (f \mapsto v)^*, \mathsf{finished} \mapsto \_|\!\} \qquad H' = H[\ell' \mapsto \{\!|c; (f \mapsto v)^*, \mathsf{finished} \mapsto \mathit{true}|\!\}]}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \overline{\alpha} \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \Xi' \cdot H' \cdot S}$$

TABLE X
NEW GLOBAL REDUCTION RULES - EXCERPT

A. Syntax of Terms

We assume two disjoint countable sets of variables *Vars* and *BVars*. The syntax of the *terms* of the abstract semantics is defined in Table XI and described below.

Boolean variables	x_b	$\in$	*BVars*
Variables	x	$\in$	*Vars*
Abstract elements	$\hat{d}$	$\in$	$\hat{D}$
Booleans	bb	::=	$0 \mid 1 \mid x_b$
Abstract locations	$\hat{\lambda}$	::=	$\mathsf{FS}(\lambda) \mid \mathsf{NFS}(\lambda)$
Abstract values	$\hat{u}, \hat{v}$	::=	$\hat{d} \mid x \mid f(\hat{v}^*)$
Abstract objects	$\hat{o}$	::=	$\{\!|c; (f_\tau \mapsto \hat{v})^*|\!\}$
Abstract arrays	$\hat{a}$	::=	$\tau[\hat{v}]$
Abstract intents	$\hat{i}$	::=	$\{\!|@c; \hat{v}|\!\}$
Abstract blocks	$\hat{b}$	::=	$\hat{o} \mid \hat{a} \mid \hat{i}$
Abstract flow-sensitive blocks	$\hat{l}$	::=	$\hat{b} \mid \bot$
Abstract flow-sensitive heap	$\hat{h}$	::=	$(pp \mapsto \hat{l})^*$
Abstract filter	$\hat{k}$	::=	$(pp \mapsto bb)^*$

TABLE XI
SYNTAX OF TERMS

Each location p_λ is abstracted by an *abstract location* $\hat{\lambda}$, which is either an abstract flow-sensitive location $\mathsf{FS}(\lambda)$ or an abstract flow-insensitive location $\mathsf{NFS}(\lambda)$. Recall the syntax of annotations: in the concrete semantics, $\lambda = c$ means that p_λ stores an activity of class c; $\lambda = in(c)$ means that p_λ stores an intent received by an activity of class c; and $\lambda = pp$ means that p_λ stores a memory block (object, array or intent) created at program point pp. Only the latter elements are amenable for a sound flow-sensitive analysis, since activity objects are shared by all the activity callbacks and received intents are shared between at least two activities, but the analysis assumes the concurrent execution of all callbacks and activities.

The analysis assumes a bounded lattice $(\hat{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ for approximating concrete values such that the abstract domain $\hat{D}$ contains at least all the abstract locations $\hat{\lambda}$ and the abstractions $\widehat{prim}$ of any primitive value $prim$. We also assume a set of interpreted functions f, containing at least sound over-approximations $\hat{\odot}, \hat{\oplus}, \hat{\circledcirc}$ of the unary, binary and comparison operators $\odot, \oplus, \circledcirc$. Abstract values $\hat{v}$ are elements $\hat{d}$ of the abstract domain $\hat{D}$, variables x from *Vars* or function applications of the form $f(\hat{v}^*)$.

The abstraction of objects $\hat{o}$ is field-sensitive, while the abstraction of arrays $\hat{a}$ and intents $\hat{i}$ is field-insensitive. The reason is that the structure of objects is statically known thanks to their type, while array lengths and intent fields (strings) may only be known at runtime. It would clearly be possible to use appropriate abstract domains to have a more precise representation of array lengths and intent fields, but we do not do it for the sake of simplicity. An *abstract block* $\hat{b}$ can be an abstract object $\hat{o}$, an abstract array $\hat{a}$ or an abstract intent $\hat{i}$. An abstract *flow-sensitive* heap $\hat{h}$ is a total mapping from the set of allocation sites pp to abstract memory blocks $\hat{b}$ or the symbol $\bot$, representing the lack of a flow-sensitive abstraction of the memory blocks created at pp.

There is just one syntactic element in Table XI which we did not discuss yet: *abstract filters*. Abstract filters $\hat{k}$ are total mappings from the set of allocation sites pp to boolean flags bb. They are technically needed to keep track of the allocation sites whose memory blocks must be downgraded to a flow-insensitive analysis when returning from a method call. The downgrading mechanism, called *lifting* of an allocation site, is explained in Section IV-C.

B. Ingredients of the Analysis

a) Overview: Our analysis is *context-sensitive*, which means that the abstraction of the elements in the call stack keeps track of a representation of their calling context. In this work, contexts are defined as tuples $(\hat{\lambda}_t, \hat{u}^*)$, where $\hat{\lambda}_t$ is an abstraction of the location storing the thread or activity which called the method, while $\hat{u}^*$ is an abstraction of the method arguments. Abstracting the calling thread or activity increases the precision of the analysis, in particular when dealing with the `join` r_t statement for thread synchronization.

Moreover, our analysis is *flow-sensitive* and computes a different over-approximation $\hat{h}$ of the state of the heap at each reachable program point, satisfying the following invariant: for each allocation site pp, if $\hat{h}(pp) = \hat{b}$, then $\hat{b}$ is an over-approximation of the most-recently allocated memory block at pp and this memory block is local to the allocating thread or activity. Otherwise, $\hat{h}(pp) = \bot$ and the memory blocks allocated at pp, if any, do not admit a flow-sensitive analysis. These memory blocks are then abstracted by an abstract *flow-insensitive* heap, defining an over-approximation of the state of the heap which is valid at all reachable program points. As such, the abstract flow-insensitive heap is not indexed by a program point.

$\mathsf{f} ::=$	
$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}^*; \hat{h}; \hat{k})$	Abstract local state
$\mathsf{AState}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}^*; \hat{h}; \hat{k})$	Abstract abnormal state
$\mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}, \hat{v}^*); \hat{v}; \hat{h}; \hat{k})$	Abstract result of method call
$\mathsf{Uncaught}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}; \hat{h}; \hat{k})$	Abstract uncaught exception
$\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$	Abstract value of right-hand side
$\mathsf{LiftHeap}(\hat{h}; \hat{k})$	Abstract heap lifting
$\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$	Abstract heap reachability
$\mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$	Abstract heap look-up
$\mathsf{H}(\lambda, \hat{b})$	Abstract flow-insensitive heap entry
$\mathsf{S}_{\mathsf{c,f}}(\hat{v})$	Abstract static field
$\mathsf{I}_{\mathsf{c}}(\hat{i})$	Abstract pending activity
$\mathsf{T}(\lambda, \hat{o})$	Abstract pending thread
$\hat{u} \sqsubseteq \hat{v}$	Partial ordering on abstract values
$\tau \leq \tau'$	Subtyping fact

TABLE XII
ANALYSIS FACTS

For space reasons, we just present selected excerpts of the analysis in the remaining of this section: the full analysis specification can be found in [6].

b) Analysis Facts: The syntax of the analysis *facts* f is defined in Table XII. The fact $\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ is used to abstract local states: it denotes that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, the state of the registers at the pc-th statement is over-approximated by $\hat{v}^*$, while $\hat{h}$ provides a flow-sensitive abstraction of the state of the heap and $\hat{k}$ tracks the set of the allocation sites which must be lifted after returning from the method. The fact $\mathsf{AState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ has an analogous meaning, but it abstracts local states trying to recover from an exception. The fact $\mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}; \hat{h}; \hat{k})$ states that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, its return value is over-approximated by $\hat{v}$; the information $\hat{h}$ and $\hat{k}$ has the same meaning as before and it is used to update the abstract state of the caller after returning from the method m. The fact $\mathsf{Uncaught}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}; \hat{h}; \hat{k})$ ensures that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, it throws an uncaught exception at the pc-th statement and the location of the exception object is over-approximated by $\hat{v}$; here, $\hat{h}$ and $\hat{k}$ are needed to update the abstract state of the caller of m, which becomes in charge of handling the uncaught exception. The fact $\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ states that $\hat{v}$ over-approximates the right-hand side of a `move` *lhs rhs* statement at program point pp.

We then have a few facts used to abstract the heap and lift the allocation sites. The facts $\mathsf{LiftHeap}(\hat{h}; \hat{k})$, $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$ and $\mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$ are the most complicated and peculiar, so they are explained in detail later on. The fact $\mathsf{H}(\lambda, \hat{b})$ models the abstract flow-insensitive heap: it states that the location p_λ stores a memory block over-approximated by $\hat{b}$ at some point of the program execution. The fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ states that the static field f of class c contains a value over-approximated by $\hat{v}$ at some point of the program execution.

Finally, the fact $\mathsf{I}_{\mathsf{c}}(\hat{i})$ tracks that an activity of class c has sent an intent over-approximated by $\hat{i}$. The fact $\mathsf{T}(\lambda, \hat{o})$ tracks that an activity or thread has started a new thread stored at some location p_λ and over-approximated by $\hat{o}$. We then have standard partial order facts $\hat{u} \sqsubseteq \hat{v}$ and subtyping facts $\tau \leq \tau'$.

c) Horn Clauses: We define *Horn clauses* as logical formulas of the form $\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}$ without free variables. In order to improve readability, we always omit the universal quantifiers in front of Horn clauses and we distinguish constants from universally quantified variables by using a sans serif font for constants, e.g., we write c to denote some specific class c. When an element in a Horn clause is unimportant, we just replace it with an underscore (_). Also, we write $\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}'_1 \wedge \ldots \wedge \mathsf{f}'_k$ for the set $\{\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}'_i \mid i \in [1, k]\}$.

d) Abstract Programs: We define *abstract programs* Δ as sets of facts and Horn clauses, where facts over-approximate program states, while Horn clauses over-approximate the concrete semantics of the analysed program.

C. The Lifting Mechanism

The *lifting* mechanism is the central technical contribution of the static analysis. It is convenient to abstract for a moment from the technical details and explain it in terms of three separate sequential steps, even though in practice these steps are interleaved together upon Horn clause resolution.

a) Computing the Abstract Filter: Let pp_a be the allocation site to lift, i.e., assume that the most-recently-allocated memory block b at pp_a must be downgraded to a flow-insensitive analysis, for example because it was shared with another activity or thread. Hence, all the memory blocks which can be reached by following a chain of locations (pointers) starting from any location in b must also be downgraded for soundness. In the analysis, we over-approximate this set of locations with facts of the form $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$, meaning that the abstract filter $\hat{k}$ represents a subset of the flow-sensitive abstract locations which are reachable along $\hat{h}$ from any flow-sensitive abstract location over-approximated by $\hat{v}$. The Horn clauses deriving $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$ are in Table XIII and should be read as a recursive computation, whose goal is to find the set of all the abstract flow-sensitive locations reachable from $\hat{v}$ and hence a sound over-approximation of the set of the allocation sites which need to be lifted. The definition uses the function $\hat{k} \hat{\sqcup} \hat{k}'$, computing the point-wise maximum between $\hat{k}$ and $\hat{k}'$.

b) Performing the Lifting: Once $\mathsf{Reach}(\mathsf{FS}(pp_a); \hat{h}; \hat{k})$ has been recursively computed, the analysis introduces a fact $\mathsf{LiftHeap}(\hat{h}; \hat{k})$ to force the lifting of the allocation sites pp such that $\hat{k}(pp) = 1$, moving their abstract blocks from the abstract flow-sensitive heap $\hat{h}$ to the abstract flow-insensitive heap. The lifting is formalized by the following Horn clause:

$$\mathsf{LiftHeap}(\hat{h}; \hat{k}) \wedge \hat{k}(\mathsf{pp}) = 1 \wedge \hat{h}(\mathsf{pp}) = \hat{b} \implies \mathsf{H}(\mathsf{pp}; \hat{b})$$

c) Housekeeping: Finally, we need to update the data structures used by the analysis to reflect the lifting, using the computed abstract filter $\hat{k}$ to update:

1) the current abstraction of the registers $\hat{v}^*$. This is done by using a function $\mathsf{lift}(\hat{v}^*; \hat{k})$, which updates $\hat{v}^*$ so that all the abstract flow-sensitive locations $\mathsf{FS}(pp)$ such that $\hat{k}(pp) = 1$ are changed to $\mathsf{NFS}(pp)$. This ensures that the next abstract heap accesses via the register abstractions perform a look-up on the abstract flow-insensitive heap

$$\mathsf{Reach}(\widehat{prim};\hat{h};0^*) \qquad \mathsf{Reach}(\mathsf{NFS}(\lambda);\hat{h};0^*) \qquad \mathsf{Reach}(\mathsf{FS}(\mathsf{pp});\hat{h};0^*[\mathsf{pp}\mapsto 1]) \qquad \mathsf{Reach}(\hat{u};\hat{h};\hat{k}) \wedge \hat{u} \sqsubseteq \hat{v} \implies \mathsf{Reach}(\hat{v};\hat{h};\hat{k})$$

$$\mathsf{Reach}(\hat{v};\hat{h};\hat{k}) \wedge \mathsf{Reach}(\hat{v};\hat{h};\hat{k}') \implies \mathsf{Reach}(\hat{v};\hat{h};\hat{k} \mathbin{\hat{\sqcup}} \hat{k}') \qquad \left.\begin{array}{r}\hat{h}(\mathsf{pp}) = \{\!|c;\_, f\mapsto \hat{v}|\!\}\\ \hat{h}(\mathsf{pp}) = \tau[\hat{v}]\\ \hat{h}(\mathsf{pp}) = \{\!|@c;\hat{v}|\!\}\end{array}\right\} \wedge \mathsf{Reach}(\hat{v};\hat{h};\hat{k}) \implies \mathsf{Reach}(\mathsf{FS}(\mathsf{pp});\hat{h};\hat{k})$$

TABLE XIII
HORN CLAUSES USED TO DERIVE THE PREDICATE $\mathsf{Reach}(\hat{v};\hat{h};\hat{k})$

$$\frac{\hat{k}(pp)=0}{\mathsf{lift}(\mathsf{FS}(pp);\hat{k})=\mathsf{FS}(pp)} \qquad \frac{\hat{k}(pp)=1}{\mathsf{lift}(\mathsf{FS}(pp);\hat{k})=\mathsf{NFS}(pp)}$$

$$\mathsf{lift}(\mathsf{NFS}(\lambda);\hat{k})=\mathsf{NFS}(\lambda) \qquad \mathsf{lift}(\widehat{prim};\hat{k})=\widehat{prim}$$

$$\frac{\hat{u}\sqsubseteq\hat{v}}{\mathsf{lift}(\hat{u};\hat{k})\sqsubseteq\mathsf{lift}(\hat{v};\hat{k})} \qquad \frac{\forall i:\mathsf{lift}(\hat{v}_i;\hat{k}))=\hat{u}_i}{\mathsf{lift}(\hat{v}^*;\hat{k})=\hat{u}^*}$$

TABLE XIV
AXIOMS REQUIRED ON THE FUNCTION $\mathsf{lift}(\hat{v}^*;\hat{k})$

for lifted allocation sites. Formally, we require the lift function to satisfy the axioms in Table XIV;

2) the current abstract flow-sensitive heap $\hat{h}$. This is done by the function $\mathsf{hlift}(\hat{h};\hat{k})$, which replaces all the entries of the form $pp\mapsto\hat{b}$ in $\hat{h}$ with $pp\mapsto\bot$ if $\hat{k}(pp)=1$, thus invalidating their flow-sensitive abstraction. If $\hat{k}(pp)=0$, instead, the function calls $\mathsf{lift}(\hat{v};\hat{k})$ on all the abstract values $\hat{v}$ occurring in $\hat{b}$, so that $\hat{b}$ itself is still analysed in a flow-sensitive fashion, but it is correctly updated to reflect the lifting of its sub-components;
3) the current abstract filter $\hat{k}'$. This is done by the function $\hat{k}\mathbin{\hat{\sqcup}}\hat{k}'$, computing the point-wise maximum between $\hat{k}$ and $\hat{k}'$. This tracks the allocation sites which must be lifted upon returning from the current method call, so that also the caller can correctly update the abstraction of its registers by using the lift function.

For simplicity, we just say that we lift some abstract value $\hat{v}$ when we lift all the allocation sites pp such that $\mathsf{FS}(pp)\sqsubseteq\hat{v}$.

d) Example: Assume integers are abstracted by their sign and consider the following abstract flow-sensitive heap:

$$\begin{array}{rl}\hat{h}= & pp_1\mapsto\tau[\mathsf{FS}(pp_2)], pp_2\mapsto\{\!|c;g\mapsto\mathsf{FS}(pp_1),g'\mapsto+|\!\}\\ & pp_3\mapsto\{\!|c';f\mapsto\mathsf{NFS}(pp_2),f'\mapsto\mathsf{FS}(pp_4)|\!\}\\ & pp_4\mapsto\{\!|c';f\mapsto\mathsf{FS}(pp_1),f'\mapsto\mathsf{FS}(pp_3)|\!\}\end{array}$$

Assume we want to lift the allocation site pp_1, the computation of the abstract filter gives: $\hat{k}=pp_1\mapsto 1, pp_2\mapsto 1, pp_3\mapsto 0, pp_4\mapsto 0$. The result of the lifting is then the following:

$$\begin{array}{rcl}\mathsf{hlift}(\hat{h};\hat{k}) & = & pp_1\mapsto\bot, pp_2\mapsto\bot,\\ & & pp_3\mapsto\{\!|c';f\mapsto\mathsf{NFS}(pp_2),f'\mapsto\mathsf{FS}(pp_4)|\!\}\\ & & pp_4\mapsto\{\!|c';f\mapsto\mathsf{NFS}(pp_1),f'\mapsto\mathsf{FS}(pp_3)|\!\}\end{array}$$

D. Abstracting Local Reduction

a) Accessing the Abstract Heaps: We observe that in the concrete semantics one often needs to read a location stored in a register and then access the contents of that location on the heap. In the abstract semantics we rely on a similar mechanism, adapted to read from the correct abstract heap. The fact $\mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\hat{\lambda};\hat{b})$ states that if $\hat{v}^*$ is an over-approximation of the content of the registers and $\hat{h}$ is an abstract flow-sensitive heap, then $\hat{\lambda}$ is an abstract location over-approximated by $\hat{v}_i$ and $\hat{b}$ is an abstract block over-approximating the memory block that register i is pointing to. Formally, this fact can be proved by the two Horn clauses below, discriminating on the flow-sensitivity of $\hat{\lambda}$:

$$\begin{array}{rcl}\mathsf{FS}(\lambda)\sqsubseteq\hat{v}_i\wedge\hat{h}(\lambda)=\hat{b} & \implies & \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\hat{b})\\ \mathsf{NFS}(\lambda)\sqsubseteq\hat{v}_i\wedge\mathsf{H}(\lambda,\hat{b}) & \implies & \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\hat{b})\end{array}$$

b) Evaluation of Right-Hand Sides: The abstract semantics needs to be able to over-approximate the evaluation of right-hand sides. This is done via a translation $\langle\!\langle rhs\rangle\!\rangle_{pp}$ generating a set of Horn clauses, which over-approximate the value of rhs at program point pp. For example, the following translation rule generates one Horn clause which approximates the content of the register r_i at pp, based on the information stored in the corresponding local state abstraction:

$$\langle\!\langle r_i\rangle\!\rangle_{pp}=\{\mathsf{LState}_{\mathsf{pp}}(\_;\hat{v}^*;\_;\_)\implies\mathsf{RHS}_{\mathsf{pp}}(\hat{v}_i)\}$$

c) Standard Statements: The abstract semantics defines, 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. We start by discussing the top part of Table XV, presenting the abstract semantics of some statements considered in the original HornDroid paper [7]. We focus in particular on the main additions needed to generalize their abstraction to implement a flow-sensitive heap analysis:

- $(\!|\texttt{new } r_d\ c'|\!)_{pp}$: When allocating a new object at pp, the abstraction of the object that was the most-recently allocated one before the new allocation, if any, must be downgraded to a flow-insensitive analysis. Therefore, we lift the allocation site pp by computing an abstract filter $\hat{k}'$ via the Reach predicate and using it to perform the lifting as described in Section IV-C. We then put in the resulting abstract flow-sensitive heap a new abstract object $\{\!|c';(f\mapsto\hat{\mathbf{0}}_\tau)^*|\!\}$ initialized to default values ($\hat{\mathbf{0}}_\tau$ represents the abstraction of the default value used to populate fields of type τ). The abstraction of the register r_d is set to the abstract flow-sensitive location $\mathsf{FS}(pp)$ to enable a flow-sensitive analysis of the new most-recently-allocated object;
- $(\!|\texttt{move } r_o.f\ rhs|\!)_{pp}$: We first use $\langle\!\langle rhs\rangle\!\rangle_{pp}$ to generate the Horn clauses over-approximating the value of rhs at program point pp. Assume then we have the over-approximation $\hat{v}''$ in a RHS fact. We have two possibilities, based on the abstract value $\hat{v}_o$ over-approximating

- $(\!|\texttt{new } r_d\ c'|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{c},\mathsf{m},\mathsf{pc});\hat{h};\hat{k}')$
  $\implies \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}')[d \mapsto \mathsf{FS}(\mathsf{c},\mathsf{m},\mathsf{pc})];\mathsf{hlift}(\hat{h};\hat{k}')[\mathsf{c},\mathsf{m},\mathsf{pc} \mapsto \{\!|\mathsf{c}';(f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}];\hat{k} \hat{\sqcup} \hat{k}')\}$
- $(\!|\texttt{move } r_o.f\ rhs|\!)_{c,m,pc} =$
  $\langle\!\langle rhs \rangle\!\rangle_{c,m,pc} \cup \{\mathsf{RHS}_{\mathsf{c,m,pc}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*;\hat{h}[\lambda \mapsto \{\!|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\!\};\hat{k})\} \cup$
  $\{\mathsf{RHS}_{\mathsf{c,m,pc}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\}) \wedge \mathsf{Reach}(\hat{v}'';\hat{h};\hat{k}') \implies$
  $\mathsf{H}(\lambda,\{\!|\mathsf{c}';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k} \hat{\sqcup} \hat{k}')\}$
- $(\!|\texttt{return}|\!)_{c,m,pc} = \{\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}_{\mathsf{res}};\hat{h};\hat{k})\}$
- $(\!|\texttt{invoke } r_o\ m'\ (r_{i_j})^{j\le n}|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge c' \le \mathsf{c}'' \implies$
  $\mathsf{LState}_{\mathsf{c}'',\mathsf{m}',0}((\hat{\lambda}_t,(\hat{v}_{i_j})^{j\le n});(\hat{\mathbf{0}}_k)^{k\le loc},(\hat{v}_{i_j})^{j\le n};\hat{h};0^*) \mid \mathsf{c}'' \in \widehat{lookup}(\mathsf{m}') \wedge sign(\mathsf{c}'',\mathsf{m}') = (\tau_j)^{j\le n} \xrightarrow{loc} \tau\} \cup$ **(1)**
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge c' \le \mathsf{c}'' \wedge \mathsf{Res}_{\mathsf{c}'',\mathsf{m}'}((\hat{\lambda}'_t,\hat{w}^*);\hat{v}'_{\mathsf{res}};\hat{h}_{\mathsf{res}};\hat{k}_{\mathsf{res}})$
  $\wedge\ \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j\le n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\hat{\lambda}_t,\_);\mathsf{lift}(\hat{v}^*;\hat{k}_{\mathsf{res}})[\mathsf{res} \mapsto \hat{v}'_{\mathsf{res}}];\hat{h}_{\mathsf{res}};\hat{k} \hat{\sqcup} \hat{k}_{\mathsf{res}}) \mid \mathsf{c}'' \in \widehat{lookup}(\mathsf{m}')\} \cup$ **(2)**
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge c' \le \mathsf{c}'' \wedge \mathsf{Uncaught}_{\mathsf{c}'',\mathsf{m}'}((\hat{\lambda}'_t,\hat{w}^*));\hat{v}'_{\mathsf{excpt}};\hat{h}_{\mathsf{res}};\hat{k}_{\mathsf{res}})$
  $\wedge\ \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j\le n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{AState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t,\_);\mathsf{lift}(\hat{v}^*;\hat{k}_{\mathsf{res}})[\mathsf{excpt} \mapsto \hat{v}'_{\mathsf{excpt}}];\hat{h}_{\mathsf{res}};\hat{k} \hat{\sqcup} \hat{k}_{\mathsf{res}}) \mid \mathsf{c}'' \in \widehat{lookup}(\mathsf{m}')\}$ **(3)**
- $(\!|\texttt{throw } r_i|\!)_{c,m,pc} = \{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{AState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*[\mathsf{excpt} \mapsto \hat{v}_i];\hat{h};\hat{k})\}$
- $(\!|\texttt{start-thread } r_i|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge c' \le \mathsf{Thread}$
  $\implies \mathsf{T}(\lambda,\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*;\hat{h};\hat{k})\} \cup$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge c' \le \mathsf{Thread} \wedge \mathsf{Reach}(\mathsf{FS}(\lambda);\hat{h};\hat{k}')$
  $\implies \mathsf{T}(\lambda,\{\!|c';(f \mapsto \hat{u})^*|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k} \hat{\sqcup} \hat{k}')\}$
- $(\!|\texttt{join } r_i|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{H}(\lambda_t,\{\!|c';(f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\}) \wedge \widehat{false} \sqsubseteq \hat{v}' \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\} \cup$
  $\{\mathsf{LState}_{\mathsf{c,m,pc}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{H}(\lambda_t,\{\!|c';(f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\}) \wedge \widehat{true} \sqsubseteq \hat{v}' \implies$
  $\mathsf{H}(\mathsf{c},\mathsf{m},\mathsf{pc};\{\!|\mathsf{IntExcpt};|\!\}) \wedge \mathsf{AState}_{\mathsf{c,m,pc}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*[\mathsf{excpt} \mapsto \mathsf{NFS}(\mathsf{c},\mathsf{m},\mathsf{pc})];\hat{h};\hat{k}) \wedge \mathsf{H}(\lambda_t,\{\!|c';(f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{false}|\!\})\}$

TABLE XV
ABSTRACT SEMANTICS OF STATEMENTS - EXCERPT

the content of the register r_o. If GetBlk_o returns an abstract flow-sensitive location $\mathsf{FS}(\lambda)$, then we perform a strong update on the corresponding element of the abstract flow-sensitive heap. If GetBlk_o returns an abstract flow-insensitive location $\mathsf{NFS}(\lambda)$, we use λ to get an abstract heap fact $\mathsf{H}(\lambda,\{\!|c';(f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\})$ and we update the field f of this object in a new heap fact: this implements a weak update, since the old fact is still valid. The abstract value $\hat{v}''$ moved to the flow-insensitive heap fact may contain abstract flow-sensitive locations, which must be downgraded by lifting $\hat{v}''$ when propagating the local state abstraction to the next program point;

- $(\!|\texttt{return}|\!)_{pp}$: The callee generates a return fact Res containing the calling context $(\hat{\lambda}_t,\hat{v}^*_{call})$, the abstract value $\hat{v}_{\mathsf{res}}$ over-approximating the return value, its abstract flow-sensitive heap $\hat{h}$ and its abstract filter $\hat{k}$ recording which allocation sites were lifted during its computation. All this information is propagated to the analysis of the caller, as we explain in the next item;
- $(\!|\texttt{invoke } r_o\ m'\ (r_{i_j})^{j\le n}|\!)_{pp}$: We statically know the name m' of the invoked method, but not the class of the receiver object in the register r_o. In part **(1)** we over-approximate dynamic dispatching as follows: we collect all the abstract objects accessible via the abstraction $\hat{v}_o$ of the content of the register r_o, but we only consider as possible receivers the ones whose type is a subtype of a class $c'' \in \widehat{lookup}(m')$, where $\widehat{lookup}(m')$ just returns the set of classes which define or inherit a method named m'. For all of them, we introduce an abstract local state fact LState over-approximating the local state of the invoked method, instantiating it with the calling context, the abstract flow-sensitive heap of the caller and an empty abstract filter.

  Part **(2)** handles the propagation of the abstraction of the return value from the callee to the caller. This is done by using the Res fact generated by the `return` statement of the callee: the caller matches appropriate callees by checking the context of the Res fact. Specifically, the caller checks that: (i) its own abstraction $\hat{\lambda}_t$ matches the abstraction $\hat{\lambda}'_t$ in the context of the callee, and (ii) that the meet of its arguments $\hat{v}_{i_j}$ and the context arguments $\hat{w}_j$ is not $\bot$. This prevents a callee from returning to a caller that could not have invoked it, in case (i) because caller and callee are being executed by different threads, and in case (ii) because the over-approximation of the arguments used by the caller and the over-approximation of the arguments supplied to the callee are disjoint. We then instantiate the abstract local state of the next program point by inheriting the abstract flow-sensitive heap of the callee $\hat{h}_{\mathsf{res}}$, lifting the abstraction of the caller registers, joining the caller abstract filter $\hat{k}$ with the callee abstract filter $\hat{k}_{\mathsf{res}}$, and storing the abstraction of the returned value $\hat{v}'_{\mathsf{res}}$ in the abstraction of the return register.

  Finally, part **(3)** of the rule is used to handle the propagation of uncaught exceptions from the callee to the caller. It uses an abstract uncaught exception fact Uncaught, generated by the exception rules explained below: it tries to throw back the exceptions to an appropriate caller, by matching the context of the Uncaught fact with the abstract local state of the caller.

d) Exceptions and Threads: The bottom part of Table XV presents the abstract semantics of some selected new

statements of the concrete semantics:

- $(\!|\texttt{throw } r_i|\!)_{pp}$: We generate an abstract *abnormal* local state fact AState from the abstract local state throwing the exception, and we set the abstraction of the special exception register accordingly;
- $(\!|\texttt{start-thread } r_i|\!)_{pp}$: We create an abstract pending thread fact T, tracking that a new thread was started. The actual instantiation of the abstract thread object is done by the abstract counterpart of the global reduction rules, which we discuss later. Observe that, if the abstract location pointing to the abstract thread object has the form $\mathsf{FS}(\lambda)$, then λ is lifted, since the parent thread can access the state of the new thread, but the two threads are concurrently executed;
- $(\!|\texttt{join } r_i|\!)_{pp}$: We just check whether the inte field of the abstract object over-approximating the running thread or activity is over-approximating $\widehat{true}$, in which case an abstract abnormal local state throwing an IntExcpt exception is generated, or $\widehat{false}$, in which case the abstract local state is propagated to the next program point.

e) Example: We show in Table XVI a (simplified) bytecode program corresponding to the code snippet in Table I. A few comments about the bytecode: the activity constructor <init> is explicitly defined; by convention, the first register after the local registers of a method is used to store a pointer to the activity object and the register ret is used to store the result of the last invoked method.

We assume that the class Leaky extends Activity and implements at least the methods send and getDeviceId, whose code is not shown here. We also use line numbers to refer to program points, which makes the notation lighter. Notice that there are only two allocation points, lines 7 and 9, therefore the abstract flow-sensitive heap will contain only two entries and have the form $7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2$.

We selected three bytecode instructions and we give for each of them the Horn clauses generated by our analysis. We briefly comment on the clauses: the new instruction at line 7 computes all the abstract flow-sensitive locations reachable from $\mathsf{FS}(7)$ with the predicate Reach: bb'_1 (resp. bb'_2) is set to 1 iff the location 7 (resp. 9) needs to be lifted. These abstract flow-sensitive locations are then lifted, if needed, using:

$$\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2),$$

and the abstract flow-sensitive heap is updated by putting a fresh Storage object in 7 and by lifting 9, if needed:

$$7 \mapsto \{\!|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\!\}, 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2).$$

The invoke instruction at line 18 has two clauses: the first clause retrieves the callee's class c' and performs an abstract virtual method dispatch (here there is only one class implementing getDeviceId, hence this step is trivial); the second clause gets the result from the called method and returns it to the caller, checking that the caller's abstract thread pointer $\hat{\lambda}_t$ and supplied argument $\hat{v}$ match the callee's context $(\hat{\lambda}'_t, \hat{v}')$ with the constraint $\hat{\lambda}_t = \hat{\lambda}'_t \wedge \hat{v} \sqcap \hat{v}' \not\sqsubseteq \bot$. We removed the exception handling clauses, as they are not relevant here.

Finally, the move instruction at line 20 is abstracted by four Horn clauses: the first one evaluates the right-hand side of the move; the two subsequent clauses execute the move in case the left-hand side is the field s of, respectively, the abstract flow-sensitive location 7 or 9; finally, the last clause is used if the left-hand side is the field s of an abstract flow-insensitive location, in which case a new abstract flow-insensitive heap entry is created.

E. Abstracting Global Reduction

The abstract counterpart of the global reduction rules is a set of Horn clauses over-approximating system events and the Android activity life-cycle. We extended the original rules of HornDroid [7] with some new rules needed to support our richer concrete semantics including threads and exceptions. Table XVII shows two of these rules to exemplify, the other rules can be found in [6]. Rule *Tstart* over-approximates the spawning of new threads by generating an abstract local state executing the run method of the corresponding thread object. Rule *AbState* abstracts the mechanism by which a method recovers from an exception: part $(\mathbf{A})$ turns an abstract abnormal state into an abstract local state if the abstraction of the exception register contains the abstract location of an object of class c extending the Throwable interface and if there exists an appropriate entry for exception handling in the exception table; part $(\mathbf{B})$ is triggered if no such entry exists, and generates an abstract uncaught exception fact, which is then used in the abstract semantics of the method invocation performed by the caller.

Let $\mathcal{R}$ denote the set of all the Horn clauses defining the auxiliary facts, like GetBlk_i, plus the Horn clauses abstracting system events and the activity life-cycle. We define the translation of a program P into Horn clauses, noted as $(\!|P|\!)$, by adding to $\mathcal{R}$ the translation of the individual statements of P.

F. Formal Results

The soundness of the analysis is proved by using *representation functions* [29]: we define a function β_{Cnf} mapping each concrete configuration Ψ to a set of abstract configurations over-approximating it. We then define a partial order $<:$ between abstract configurations, where $\Delta <: \Delta'$ should be interpreted as: Δ is no coarser than Δ'. The soundness theorem can be stated as follows; its proof can be found in [6].

Theorem 1 (Global Preservation) *If $\Psi \Rightarrow^* \Psi'$ under a given program P, then for any $\Delta_1 \in \beta_{Cnf}(\Psi)$ and $\Delta_2 :> \Delta_1$ there exist $\Delta'_1 \in \beta_{Cnf}(\Psi')$ and $\Delta'_2 :> \Delta'_1$ s.t. $(\!|P|\!) \cup \Delta_2 \vdash \Delta'_2$.*

We now discuss how a sound static taint analysis can be implemented on top of our formal result. First, we extend the syntax of concrete values as follows:

$$\begin{array}{llll} \text{Taint} & t & ::= & \mathsf{public} \mid \mathsf{secret} \\ \text{Values} & u, v & ::= & prim^t \mid \ell \end{array}$$

The set of taints is a two-valued lattice, and we use $\sqsubseteq^{\mathsf{t}}$ and $\sqcup^{\mathsf{t}}$ to denote respectively the standard ordering on taints (where $\mathsf{public} \sqsubseteq^{\mathsf{t}} \mathsf{secret}$) and their join. When performing unary and binary operations, taints are propagated by having the taint of the result be the join of the taints of the arguments.

Bytecode Example:

```
1  .class public Leaky
2  .super Activity
3  .field st:Storage
4  .field st2:Storage
5  .method constructor <init>()
6  .1 local register
7      new r0 Storage
8      move r1.st r0
9      new r0 Storage
10     move r1.st2 r0
11 .end method
12 .method onRestart()
13 .1 local register
14     move r1.st2 r1.st
15 .end method
16 .method onResume()
17 .1 local register
18     invoke r1 getDeviceId()
19     move r0 r1.st2
20     move r0.s ret
21 .end method
22 .method onPause()
23 .2 local registers
24     move r0 r2.st
25     move r1 r0.s
26     move r0 "http://myapp.com/"
27     invoke r2 send() r1 r0
28 .end method
```

Generated Horn Clauses for Line 7:

- $\mathsf{LState}_7(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge \mathsf{Reach}(\mathsf{FS}(7); 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \implies$
  $\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge \mathsf{LState}_8(\_; r_0 \mapsto \mathsf{FS}(7), r_1 \mapsto \mathsf{lift}(\hat{u}; 7 \mapsto bb'_1, 9 \mapsto bb'_2);$
  $7 \mapsto \{\!|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\!\}, 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2); 7 \mapsto bb_1 \hat{\sqcup} bb'_1, 9 \mapsto bb_2 \hat{\sqcup} bb'_2)$

Generated Horn Clauses for Line 18:

- $\mathsf{LState}_{18}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{GetBlk}_1(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \_; \{\!|c'; \_|\!\}) \wedge c' \leq \texttt{Leaky} \implies$
  $\mathsf{LState}_0((\hat{\lambda}_t, \hat{v}); r_0 \mapsto \hat{v}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto 0, 9 \mapsto 0)$
- $\mathsf{LState}_{18}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{GetBlk}_1(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \_; \{\!|c'; \_|\!\}) \wedge c' \leq \texttt{Leaky} \wedge$
  $\mathsf{Res}_{\texttt{getDeviceId}}((\hat{\lambda}'_t, \hat{v}'); \hat{u}'_{\mathsf{res}}; 7 \mapsto \hat{l}'_1, 9 \mapsto \hat{l}'_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \hat{v} \sqcap \hat{v}' \not\sqsubseteq \bot \implies$
  $\mathsf{LState}_{19}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{u}'_{\mathsf{res}}; 7 \mapsto \hat{l}'_1, 9 \mapsto \hat{l}'_2; 7 \mapsto bb_1 \hat{\sqcup} bb'_1, 9 \mapsto bb_2 \hat{\sqcup} bb'_2)$

Generated Horn Clauses for Line 20:

- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \implies \mathsf{RHS}_{20}(\hat{w})$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{RHS}_{20}(\hat{u}') \wedge \mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{FS}(7); \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{u}'|\!\}, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2)$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{RHS}_{20}(\hat{u}') \wedge \mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{FS}(9); \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{u}'|\!\}; 7 \mapsto bb_1, 9 \mapsto bb_2)$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge \mathsf{RHS}_{20}(\hat{u}') \wedge$
  $\mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \texttt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{NFS}(\mathsf{pp}); \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{v}'|\!\}) \wedge \mathsf{Reach}(\hat{u}'; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \implies$
  $\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge \mathsf{H}(\mathsf{pp}, \{\!|\texttt{Storage}; \texttt{s} \mapsto \hat{u}'|\!\}) \wedge$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \mathsf{lift}(\hat{u}; 7 \mapsto bb'_1, 9 \mapsto bb'_2), r_1 \mapsto \mathsf{lift}(\hat{v}; 7 \mapsto bb'_1, 9 \mapsto bb'_2), \texttt{ret} \mapsto \mathsf{lift}(\hat{w}; 7 \mapsto bb'_1, 9 \mapsto bb'_2);$
  $7 \mapsto \mathsf{hlift}(\hat{l}_1; 7 \mapsto bb'_1, 9 \mapsto bb'_2), 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2); 7 \mapsto bb_1 \hat{\sqcup} bb'_1, 9 \mapsto bb_2 \hat{\sqcup} bb'_2)$

TABLE XVI
EXAMPLE OF DALVIK BYTECODE AND EXCERPT OF THE CORRESPONDING HORN CLAUSES

$$
\begin{array}{lcl}
\mathit{Tstart} & = & \{\mathsf{T}(\lambda, \{\!|c; (f \mapsto \_)^*|\!\}) \wedge c \leq \mathsf{c}' \wedge c \leq \mathsf{Thread} \implies \\
 & & \mathsf{LState}_{\mathsf{c}',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*) \mid \mathsf{c}' \in \widehat{\mathit{lookup}}(\mathsf{run}) \wedge \mathit{sign}(\mathsf{c}', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void}\} \\
\mathit{AbState} & = & \{\mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*; \hat{h}; \_; \{\!|\mathsf{c}'; \_|\!\}) \wedge \mathsf{c}' \leq \mathsf{Throwable} \implies \\
 & & \mathsf{LState}_{\mathsf{c},\mathsf{m},\mathsf{pc}'}(\_; \hat{v}^*; \hat{h}; \hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c}, \mathsf{m}, \mathsf{pc}, \mathsf{c}') = \mathsf{pc}'\} \cup \qquad (\mathbf{A}) \\
 & & \{\mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*; \hat{h}; \_; \{\!|\mathsf{c}'; \_|\!\}) \wedge \mathsf{c}' \leq \mathsf{Throwable} \implies \\
 & & \mathsf{Uncaught}_{\mathsf{c},\mathsf{m}}(\_; \hat{v}_{\mathsf{excpt}}; \hat{h}; \hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c}, \mathsf{m}, \mathsf{pc}, \mathsf{c}') = \bot\} \qquad (\mathbf{B})
\end{array}
$$

TABLE XVII
GLOBAL RULES OF THE ABSTRACT SEMANTICS - EXCERPT

We then define a taint extraction function taint_Ψ: informally it is a function that, given a value v, extracts its taint by doing a recursive computation: if v is a primitive value, the function just returns the taint of the value; if v is a location, the function recursively computes the join of all the taints accessible from v along the heap of Ψ.

We also define the abstract counter-part Taint of taint_Ψ: the analysis fact $\mathsf{Taint}(\hat{v}, \hat{h}, \hat{t})$ holds when $\hat{v}$ has taint $\hat{t}$ in the abstract flow-sensitive heap $\hat{h}$. The rules defining Taint are similar to the rules defining Reach, since both predicates need to perform a recursive computation on the abstract heap. The formal definitions underlying this intuitive description can be found in the long version [6].

Finally, we assume two sets *Sinks* and *Sources*, where *Sinks* (resp. *Sources*) contains a pair (c, m) if and only if a method m of a class c is a sink (resp. a source). We assume that when a source returns a value, it always has the secret taint.

Definition 2 *A program P* leaks *starting from a configuration Ψ if there exists $(c, m) \in$ Sinks such that $\Psi \Rightarrow^* \Omega \cdot \Xi \cdot H \cdot S$ and there exists $\langle \ell, s, \pi, \gamma, \alpha \rangle \in \Omega$ or $\langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle \in \Xi$ such that $\alpha = \langle c, m, 0 \cdot u^* \cdot st^* \cdot R \rangle :: \alpha'$, $R(r_k) = v$ and $\mathsf{taint}_\Psi(v) =$ secret for some r_k and v.*

We then state the soundness of our taint tracking analysis in the following lemma: its proof is rather simple and can be found in [6].

Lemma 1 *If for all sinks* $(c, m) \in \mathit{Sinks}$, $\Delta \in \beta_{\mathit{Cnf}}(\Psi)$*:*

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c,m,0}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{Taint}(\hat{v}_i, \hat{h}, \mathsf{secret})$$

is unsatisfiable for each i*, then* P *does not leak from* Ψ*.*

V. Experiments

We implemented a prototype of our flow-sensitive analysis as an extension of an existing taint tracker, HornDroid [7]. Our tool encodes the application to analyse as a set of Horn clauses, as we detailed in the previous section, and then uses the SMT solver Z3 [28] to statically detect information leaks. More specifically, the tool automatically generates a set of queries for the analysed application based on a public database of Android sources and sinks [33]; if no query is satisfiable according to Z3, no information leak may occur by the soundness results of our analysis.

A. Testing on DroidBench

We tested our flow-sensitive extension of HornDroid (called fsHornDroid) against DroidBench [3], a common benchmark of 115 small applications proposed by the research community to test information flow analysers for Android[3]. In our experiments we compared with the most popular and advanced static taint trackers for Android applications: FlowDroid [3], AmanDroid [40], DroidSafe [15] and the original version of HornDroid [7]. For all the tools, we computed standard validity measures (sensitivity for soundness and specificity for precision) and we tracked the analysis times on the 115 applications included in DroidBench: the experimental results are summarised in Table XVIII.

Like the original version of HornDroid, fsHornDroid detects all the information leaks in DroidBench, since its sensitivity is 1. However, fsHornDroid turns out to be the most precise static analysis tool to date, with a value of specificity which is strictly higher than the one of all its competitors. In particular, fsHornDroid produces only 4 false positives on DroidBench: a leak inside an exception that is never thrown; a leak inside an unregistered callback which cannot be triggered; a leak inside an undeclared activity which cannot be started; and a leak of a public element of a list which contains also a confidential element. The last two cases should be easy to fix: the former by parsing the application manifest and the latter by implementing field-sensitivity for lists.

We also evaluated the analysis times of the applications in DroidBench for the different tools. In terms of performances, the original version of HornDroid is better than fsHornDroid as expected. However, the performances of fsHornDroid are satisfying: the median analysis time does not change too much with respect to HornDroid, which is the fastest tool, while the average analysis time is comparable with other flow-sensitive analysers like FlowDroid and AmanDroid.

[3]We removed from DroidBench 4 applications testing implicit information flows, since none of the available tools aims at supporting them.

B. Testing on Real Applications

In order to test the scalability of fsHornDroid, we picked the top 4 applications from 16 categories in a publicly available snapshot of the Google Play market [39]. For each application, we run fsHornDroid setting a timeout of 3 hours for finding the first information leak. In the end, we managed to get the analysis results within the timeout for 62 applications, whose average and median sizes were 7.4 Mb and 5 Mb respectively. The tool reported 47 applications as leaky and found no direct information leaks for 15 applications. Unfortunately, the absence of a ground truth makes it hard to evaluate the validity of the reported leaks, which we plan to manually investigate in the future. To preliminarily assess the improvement in precision due to flow-sensitivity, however, we sampled 3 of the potentially leaky applications and we checked all their possible information leaks. On these applications, fsHornDroid eliminated 17 false positives with respect to HornDroid, which amount to the 18% of all the checked flows.

In terms of performances, fsHornDroid spent 17 minutes on average to perform the analysis, with a median analysis time of 2 minutes on an Intel Xeon E5-4650L 2.60 GHz. The constantly updated experimental evaluation is available online, along with the web version of the tool and its sources [1]. Our results demonstrate that fsHornDroid scales to real applications, despite the increased performance overhead with respect to the original HornDroid.

C. Limitations

Our implementation of fsHornDroid does not aim at solving a few important limitations of HornDroid. First, a comprehensive implementation of *analysis stubs* for unknown methods is missing: this issue was thoroughly discussed by the authors of DroidSafe [15] and we think their research may be very helpful to improve on this. Moreover, the analysis does not capture *implicit* information flows, but only direct information leaks, and it does not cover native code, but only Dalvik bytecode. Finally, the analysis has no way of being less conservative on *intended* information flows: implementing declassification mechanisms would be important to analyse real applications without raising a high number of false alarms.

VI. Related Work

There are several static information flow analysers for Android applications (see, e.g., [41], [42], [27], [14], [22], [3], [40], [15], [7]). We thoroughly compared with the current state of the art in the rest of the paper, so we focus here on other related works.

a) Sound Analysis of Android Applications: The first paper proposing a formally sound static analysis of Android applications is a seminal work by Chaudhuri [8]. The paper presented a type-based analysis to reason on the data-flow security properties of Android applications modeled in an idealised calculus. A variant of the analysis was implemented in a prototype tool, SCanDroid [13]. Unfortunately, SCanDroid is in an early prototype phase and it cannot analyse the applications in DroidBench [3].

Validity Measures on DroidBench:

	FlowDroid	AmanDroid	DroidSafe	HornDroid	fsHornDroid
Sensitivity	0.67	0.74	0.92	1	1
Specificity	0.58	0.74	0.47	0.68	0.79
F-Measure	0.62	0.74	0.62	0.81	0.88

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

Analysis Times on DroidBench:

	FlowDroid	AmanDroid	DroidSafe	HornDroid	fsHornDroid
Average	22s	11s	2m92s	1s	14s
1st Quartile	13s	9s	2m38s	1s	1s
2nd Quartile	14s	10s	3m1s	1s	2s
3rd Quartile	15s	11s	3m26s	1s	5s

TABLE XVIII
VALIDITY MEASURES AND ANALYSIS TIMES ON DROIDBENCH

Sound type systems for Android applications have also been proposed in [25] to prove non-interference and in [5] to prevent privilege escalation attacks. In both cases, the considered formal models are significantly less detailed than ours and the purpose of the static analyses is different. Though the framework in [25] can be used to prevent implicit information flows, unlike our approach, the analysis proposed there is not fully automatic, it does not approximate runtime value, thus sacrificing precision, and it was not experimentally evaluated.

Julia is a static analysis tool based on abstract interpretation, first developed for Java and recently extended to Android [30]. It is a commercial product and supports many useful features, including class analysis, nullness analysis and termination analysis for Android applications, but it does not track information flows. Moreover, Julia does not handle multi-threading and we are not aware of the existence of a soundness proof for its extension to Android.

b) Pointer Analysis: Pointer analysis aims at over-approximating the set of objects that a program variable can refer to, and it is a well-established and rich research field [20], [37], [36]. The most prominent techniques in pointer analysis are variants of the classical Andersen algorithm [2], including flow-insensitive analyses [10], [32], [17], [21] and flow-sensitive analyses [9], [11], [19], [23]; light-weight analyses in the flavor of the unification-based Steensgaard analysis [38], which are flow-insensitive and very efficient; and shape analysis techniques [35], which can be used to prove complex properties about the heap, often at the price of efficiency.

Although pointer analysis of sequential programs is well-studied, much less attention has been paid to pointer analysis of concurrent programs. Most flow-insensitive analyses for sequential programs remain sound for concurrent programs [34], because flow-insensitivity forces a sound analysis to consider all the possible interleavings of reads and writes to the heap. Designing a sound flow-sensitive pointer analysis for concurrent programs is more complicated and most flow-sensitive analyses for sequential programs cannot be easily adapted to concurrent programs. Still, flow-sensitive sound analyses for concurrent programs exist. The approach of Rugina and Rinard [34] handles concurrent programs with an unbounded number of threads, recursion and dynamic allocations, but it does not allow strong updates on dynamically allocated heap objects. Gotsman *et al.* [16] proposed a framework to prove complex properties about programs with dynamic allocations by using shape analysis and separation logic, but their approach requires users or external tools to provide annotations, and it is restricted to a bounded number of threads.

VII. CONCLUSION

We presented the first static analysis for Android applications which is both flow-sensitive on the heap abstraction and provably sound with respect to a rich formal model of the Android ecosystem. Designing a sound yet precise analysis in this setting is particularly challenging, due to the complexity of the control flow of Android applications. In this work, we adapted ideas from *recency abstraction* [4] to hit a sweet spot in the analysis design space: our proposal is sound, precise, and efficient in practice. We substantiated these claims by implementing the analysis in HornDroid [7], a state-of-the-art static information flow analyser for Android applications, and by performing an experimental evaluation of our extension. Our work takes HornDroid one step further towards the sound information flow analysis of real Android applications.

Acknowledgements: This work has been partially supported by the MIUR project ADAPT, by the CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA, and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA). This work also acknowledges support by the FWF project W1255-N23 and the DAAD-MIUR Joint Mobility Program "Client-side Security Enforcement for Mobile and Web Applications".

REFERENCES

[1] secpriv.tuwien.ac.at/tools/horndroid, website of fsHornDroid
[2] Andersen, L.O.: Program analysis and specialization for the C programming language. Tech. rep., University of Copenhagen (1994)
[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI. pp. 259–269. ACM (2014)

[4] Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: SAS. pp. 221–239. Springer-Verlag (2006)

[5] Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: Towards security type-checking of Android applications. In: FMOODS/FORTE. pp. 289–304 (2013)

[6] Calzavara, S., Grishchenko, I., Koutsos, A., Maffei, M.: A sound flow-sensitive heap abstraction for the static analysis of Android applications (2017), full version at arXiv:1705.10482

[7] Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of Android applications by SMT solving. In: EuroS&P. IEEE (2016)

[8] Chaudhuri, A.: Language-based security on Android. In: PLAS. pp. 1–7. ACM (2009)

[9] Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL. pp. 232–245. ACM (1993)

[10] Das, M.: Unification-based pointer analysis with directional assignments. SIGPLAN Not. 35(5), 35–46 (May 2000)

[11] Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. SIGPLAN Not. 29(6), 242–256 (Jun 1994)

[12] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)

[13] Fuchs, A.P., Chaudhuri, A., , Foster, J.S.: Scandroid: Automated security certification of Android applications. Tech. rep., University of Maryland (2009)

[14] Gibler, C., Crussell, J., Erickson, J., Chen, H.: Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In: TRUST. pp. 291–307. Springer-Verlag (2012)

[15] Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of Android applications in DroidSafe. In: NDSS. IEEE (2015)

[16] Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI. pp. 266–277. ACM (2007)

[17] Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. SIGPLAN Not. 42(6), 290–299 (Jun 2007)

[18] Java 8 Documentation on Thread. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html, last accessed on February 2017

[19] Kahlon, V.: Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. SIGPLAN Not. 43(6), 249–259 (Jun 2008)

[20] Kanvar, V., Khedker, U.P.: Heap abstractions for static analysis. CoRR abs/1403.4910 (2014), http://arxiv.org/abs/1403.4910

[21] Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. SIGPLAN Not. 48(6), 423–434 (Jun 2013)

[22] Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in Android applications. In: MoST (2012)

[23] Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. SIGPLAN Not. 46(1), 3–16 (Jan 2011)

[24] Lochbihler, A.: Making the java memory model safe. ACM Trans. Program. Lang. Syst. 35(4), 12:1–12:65 (Jan 2014), http://doi.acm.org/10.1145/2518191

[25] Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: Towards a certifying app store for Android. In: SPSM@CCS. pp. 93–104. ACM (2014)

[26] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In: CCS. pp. 229–240. ACM (2012)

[27] Mann, C., Starostin, A.: A framework for static detection of privacy leaks in Android applications. In: SAC. pp. 1457–1462. ACM (2012)

[28] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer-Verlag (2008)

[29] Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer-Verlag (1999)

[30] Payet, É., Spoto, F.: Static analysis of Android programs. Information & Software Technology 54(11), 1192–1201 (2012)

[31] Payet, É., Spoto, F.: An operational semantics for Android activities. In: PEPM. pp. 121–132. ACM (2014)

[32] Pereira, F.M.Q., Berlin, D.: Wave propagation and deep propagation for pointer analysis. In: GCO. pp. 126–135 (2009)

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

[34] Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. SIGPLAN Not. 34(5), 77–90 (May 1999)

[35] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL. pp. 105–118. ACM (1999)

[36] Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Found. Trends Program. Lang. 2(1), 1–69 (Apr 2015)

[37] Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming, pp. 196–232. Springer-Verlag, Berlin, Heidelberg (2013), http://dl.acm.org/citation.cfm?id=2554511.2554523

[38] Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41. ACM (1996)

[39] The Collection of Android Apps and Metadata. https://archive.org/details/android_apps&tab=about, last accessed on February 2017

[40] Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In: CCS. pp. 1329–1341. ACM (2014)

[41] Yang, Z., Yang, M.: Leakminer: Detect information leakage on Android with static taint analysis. In: WCSE. pp. 101–104. IEEE (2012)

[42] Zhao, Z., Osorio, F.C.C.: Trustdroid: Preventing the use of smartphones for information leaking in corporate networks through the use of static analysis taint tracking. In: MALWARE. pp. 135–143. IEEE (2012)