SymDroid: Symbolic Execution for Dalvik Bytecode

Technical Report CS-TR-5022, July 2012

Jinseong Jeon, Kristopher K. Micinski, Jeffrey S. Foster

Department of Computer Science, University of Maryland, College Park

Abstract

Apps on Google's Android mobile device platform are written in Java, but are compiled to a special bytecode language called Dalvik. In this paper, we introduce SymDroid, a symbolic executor that operates directly on Dalvik bytecode. SymDroid begins by first translating Dalvik into μ-Dalvik, a simpler language that has only 16 instructions, in contrast to Dalvik's more than 200 instructions. We present a formalism for SymDroid's symbolic executor, which can be described with a small number of operational semantics rules; this semantics may be of independent interest. In addition to modeling bytecode instructions, SymDroid also contains models of some key portions of the Android platform, including libraries and the platform's lifecycle control code. We evaluated SymDroid in two ways. First, we ran it on the Android Compatibility Test Suite, and found it passed all tests except ones that used library or system routines we have not yet implemented. On this test suite, SymDroid runs about twice as slow as the Dalvik VM, and about twice as fast as the Java VM. Second, we used SymDroid to discover the (path) conditions under which contacts may be accessed on an Android app, and found it was able to do so successfully. These results suggest that SymDroid, while still a prototype, is a promising first step in enabling direct, precise analysis of Android apps.

Keywords:
Android, Symbolic Execution, Dalvik Bytecode

1. Introduction

Google's Android is currently the most popular mobile device platform, running on a majority of all smartphones [1]. The key feature of these devices (the "smart" in smartphone) is the apps that run on them, providing a wide variety of capabilities and services. While the mobile device space is still evolving, it is clear that developers and users are concerned about app correctness, security, and privacy. However, while Android apps are written in Java, they are compiled to Google's Dalvik Virtual Machine bytecode format. Thus, while existing Java-based program analysis tools could potentially be used to reason about properties of apps (including correctness, security, and privacy), in practice doing so requires either access to an app's Java source, or decompilation from Dalvik back to Java. The former is problematic for many uses (e.g., any case where we want to analyze an app without source), and the latter requires significant engineering effort, adds a layer of complication in understanding the tool output, and introduces yet another source of potential bugs in an analysis. Moreover, as far as we are aware, there is currently no 100% robust and correct Dalvik-to-Java reverse translation tool, though several efforts are close [2, 3].

In this paper, we take a first step toward developing a suite of program analysis tools that work directly on Dalvik bytecode. We introduce SymDroid, a symbolic executor [4, 5, 6, 7, 8, 9, 10, 11, 12] for Dalvik. SymDroid is essentially a Dalvik bytecode interpreter, but with the additional ability to operate on *symbolic expressions*, which represent potentially unknown quantities. SymDroid uses an SMT solver to test whether assertions involving those expressions are always true; if not, SymDroid can produce a counter-example showing a cause of the assertion failure.

Email addresses: jsjeon@cs.umd.edu (Jinseong Jeon), micinski@cs.umd.edu (Kristopher K. Micinski), jfoster@cs.umd.edu (Jeffrey S. Foster)

SymDroid may also branch its execution if a symbolic expression is used as the guard of a conditional, in the case that both the true and false branch are feasible. Symbolic execution has been used for a wide variety of purposes, but some of the most promising results have been for finding bugs, including security vulnerabilities, in software systems (see citation list above). We hope that, among others, SymDroid could be used for a similar purpose. We refer the reader to the works mentioned above for more background on symbolic execution.

One way to view a symbolic executor is as a runnable operational semantics, and thus we envision that SymDroid's semantics for Dalvik might be of independent interest. Thus, we aimed to develop as clean and simple a semantics as possible. The result is μ-Dalvik, a language that contains just 16 instructions, compared to more than 200 Dalvik bytecode instructions, and to which it is easy to translate Dalvik. μ-Dalvik achieves its compactness through three basic transformations. First, it coalesces multiple Dalvik instructions that are distinguished only by bit widths, e.g., `goto +AA`, `goto/16 +AAAA`, and `goto/32 +AAAAAAAA` become a single μ-Dalvik goto statement. Second, it encodes operand types in the operand, rather than in the operator, e.g., `aput-byte`, `aput-char`, and `aput-short` all map to the same μ-Dalvik instruction, and we store the operand type in the operand. Finally, μ-Dalvik expands some complex Dalvik instructions into sequences of simpler instructions, e.g., `packed-switch` becomes a sequence of conditions. Note that μ-Dalvik aims to minimize the number of instructions (and thus keep the semantics cleaner), whereas Dalvik's goal is to maximize performance and minimize code size. (Section 2 presents μ-Dalvik in detail.)

The core of SymDroid, the symbolic execution rules for each μ-Dalvik instruction, are standard and quite straightforward, as μ-Dalvik is so compact. Section 3 fits essentially all of the main rules on a single page of text, and those rules correspond directly to our implementation, which comprises approximately 6,700 lines of OCaml code. Of course, in addition to modeling the bytecode itself, SymDroid also needs to provide models of the platform, including the system libraries (many of which contain native code, and hence cannot be directly executed by SymDroid) and the Android control framework (which is quite complex). As a prototype tool, currently SymDroid implements just enough of these to support our case study (see below). (Section 4 describes our Android platform model in more detail.)

We evaluated SymDroid in two ways. First, we used it to run the Android Compatibility Test Suite (CTS) [13], which tries to thoroughly exercise Dalvik bytecode and platform functionality. We found that SymDroid passed 26 out of 92 CTS tests. It failed the remaining tests not because of errors in instruction handling, but because we do have not yet implemented all the Java and Android libraries used by CTS. Thus, SymDroid passes all the tests that it could be expected to pass. We also measured SymDroid's performance, and found it is roughly 2x slower than the Dalvik virtual machine, and roughly 2x faster than a Java virtual machine. Note that in these experiments there was no symbolic computation—all values were concrete. Thus, these results suggest that SymDroid is likely fast enough in practice, especially since in our experience symbolic executors spend much of their time in the SMT solver.

Second, we used SymDroid to discover under what conditions certain privileged operations were used in PickContact, an Activity from the API demonstration app supplied with the Android SDK. This problem is a good fit for symbolic execution as the interaction between the user and the system is complex on Android, and determining whether a call is privileged can depend on subtle semantics. We ran SymDroid on the PickContact and found it was able to discover the correct conditions under which the READ_CONTACTS permission was used. (Section 5 describes our experimental results.)

In summary, the contributions of this paper are

- A clean and concise core bytecode language, μ-Dalvik, to which Dalvik can be easily translated (Section 2), and which has a simple semantics (Section 3).
- A discussion of the issues of modeling the Android platform and other challenges in building SymDroid, a symbolic executor for μ-Dalvik (Section 4).
- Experiments demonstrating the correctness of SymDroid and suggesting how it may be useful in practice (Section 5).

2. μ-Dalvik

Dalvik bytecode is designed to run in a resource constrained environment, namely on mobile devices. Among others, Dalvik is carefully designed to reduce the overall size of applications and for performance [14]. In contrast,

P	::=	$\langle cls^*, fld^*, mtd^*, str^* \rangle$	DEX binary
cls	::=	`class` $@s <$ $@c$ `imp` $@c^*$ $\{@f^*\ @m^*\}$	Class definition
fld	::=	`field` $@s : @c$	Field definition
mtd	::=	`method` $@s : @c^* \rightarrow @c\ \{b\}$	Method definition
b	::=	$\cdot \mid s\ ;\ b$	Method body

s	::=	`goto` pc	Unconditional branch
	$\mid$	`if` $r \lessgtr r$ `then` pc	Conditional branch
	$\mid$	$lhs \leftarrow rhs$	Move
	$\mid$	$r \leftarrow r \oplus r$	Binary operation
	$\mid$	$r \leftarrow \odot\, r$	Unary operation
	$\mid$	$r \leftarrow$ `new` $@c$	New instance
	$\mid$	$r \leftarrow$ `newarray` $@c[r]$	New array
	$\mid$	$r \leftarrow (@c)\ r$	Type cast
	$\mid$	$r \leftarrow r$ `instanceof` $@c$	Instance of
	$\mid$	$r.@m(argv)$	Dynamic method invocation
	$\mid$	$@m(argv)$	Static method invocation
	$\mid$	`return`	Method return
	$\mid$	$r \leftarrow$ `sym`	New symbolic variable
	$\mid$	`assert` r	Assertion

$\oplus$	::=	$+ \mid - \mid \times \mid \div \mid \cdots$	Binary operators
$\lessgtr$	::=	$< \mid > \mid \cdots$	Comparison operators
$\odot$	::=	$- \mid \neg \mid \cdots$	Unary operators
lhs	::=	r	Register
	$\mid$	$r[r]$	Array access
	$\mid$	$r.@f$	Instance fields
	$\mid$	$@f$	Static fields
rhs	::=	lhs	
	$\mid$	c	Constants
$argv$	::=	$\cdot \mid r,\ argv$	Arguments
c	::=	n	Integers
	$\mid$	$@s$	String indexes
	$\mid$	`true` $\mid$ `false`	Booleans
	$\mid$	`null`	Null

Figure 1: μ-Dalvik syntax.

we are interested in performing more expensive, off-device analyses, in particular symbolic execution. For research purposes, we also want to have as simple and concise a semantics as possible. μ-Dalvik represents our attempt to achieve these aims.

μ-Dalvik has three main differences compared to Dalvik:

- Dalvik includes many instruction variants that differ only in the number of bits reserved for operands. For example, consider three Dalvik move instructions, `move vA, vB`; `move/from16 vAA, vBBBB`; and `move/16 vAAAA, vBBBB`. These instructions all move values of the same size among registers; the only difference between them is how many bits they use to represent registers indices (`vA`, `vAA`, and `vAAAA` require 4, 8, and 16 bits, respectively). Since we are not constrained in terms of bytecode space representation, we instead always use 32-bit indices to refer to registers.

- Many Dalvik instructions encode their operand type in the operator. For example, to read an instance field, Dalvik includes opcodes `iget` (read an integer or float instance field), `iget-object` (read an Object instance field), `iget-boolean` (read a boolean instance field), `iget-byte` (read a byte instance field), etc.. From the perspective of an analysis tool, we prefer to have one generic instruction of each kind, but allow the operand type to vary.

- Dalvik includes some complex instructions that μ-Dalvik desugars to simpler instruction sequences. For example, the `filled-new-array(/range)` and `fill-array-data` instructions fill the given array with the supplemental data. In SymDroid, these instructions are desugared into a sequence of μ-Dalvik instructions that copy constant bytes into the array.

Figure 1 presents the syntax of μ-Dalvik and a simplified version of the Dalvik bytecode format. A μ-Dalvik program is made up of definitions of classes, fields, and methods, and also contains a *string pool* mapping integer indices to string values. In full Dalvik, the string pool (actually, several different pools) exist to make the bytecode compact by reusing strings across the entire codebase of an app; even such strings as class names, method names, and types are shared in the string pool. We maintain this indirect representation only for aesthetic reasons. Thus, in μ-Dalvik, all strings are accessed via their indices, which we write as $@c$ (class index), $@f$ (field index), $@m$ (method index), and $@s$ (program string index).

Java source code	Dalvik instructions	μ-Dalvik instructions
`static byte foo(int x) {`	*parameter* `x = v2`	
	`const/16 v0 1000`	$r_0 \leftarrow 1000$
`  if(x > 1000) {`	`if-le v2 v0 +9`	`if` $r_2 \leq r_0$ `then` ℓ_2
	`rem-int/lit16 v0 v2 1000`	$r_0 \leftarrow r_2 \% 1000$
`    byte y = foo(x % 1000);`	`invoke-static v0` $@m_0$	$@m_0(r_0)$
	`move-result v0`	$r_0 \leftarrow r_{ret}$
`    return y;`	`return v0`	ℓ_1 : $r_{ret} \leftarrow r_0$ `return`
`  }`	`const/4 v0 2`	ℓ_2 : $r_0 \leftarrow 2$
`  byte [] data = {7, 9};`	`new-array v0 v0` $@c_0$	$r_0 \leftarrow$ `newarray` $@c_0[r_0]$
	`fill-array-data v0 +8`	$r_t \leftarrow 0; r_0[r_t] \leftarrow 7$
		$r_t \leftarrow 1; r_0[r_t] \leftarrow 9$
`  byte z = data[x % 2];`	`rem-int/lit8 v1 v2 2`	$r_1 \leftarrow r_2 \% 2$
	`aget-byte v0 v0 v1`	$r_0 \leftarrow r_0[r_1]$
`  return z;`	`goto -11`	`goto` ℓ_1
`}`	`[0: 7]`	*See fill-array-data translation above*
	`[1: 9]`	

$@c_0$ = byte array $@m_0$ = foo()

Figure 2: Translation example.

Class definitions contain the class name, its superclass, its implemented interfaces, and its fields and methods. Field definitions are comprised of the field name and type. Finally, method definitions include the method name, argument types, return type, and method body.

A method body is a sequence of statements. As execution progresses we maintain the *program counter pc*, which is the index of the currently executing statement in the sequence. (Note that in Dalvik, the program counter is a pointer to the bytecode instruction's offset, which can be slightly different as different bytecodes have different numbers of operands, and hence use different numbers of bytes.) As in many imperative languages, we distinguish the left and right-hand side operands of move statements. On the left-hand side we allow a single register name; an array access; and instance and static field access with field index $@f$. Right-hand side operands of a move statement can have any left-hand side operands as well as constants.

Next, μ-Dalvik includes binary and unary operations. `new` and `newarray` statements create a new instance of $@c$ and an array of class $@c$, respectively. For array allocation, a register containing the array size is also required. μ-Dalvik also includes type cast and `instanceof`. Method calls refer to method index $@m$, and all arguments must be in registers; dynamically dispatched method calls also include a receiver object. Finally, μ-Dalvik includes a special statement to insert symbolic variables and an `assert` statement that checks a property of interest.

Translation from Dalvik to μ-Dalvik. Translating Dalvik bytecode into μ-Dalvik is a fairly straightforward process. Figure 2 illustrates the translation process from Java source code (left column) into Dalvik (middle column) and then into μ-Dalvik (right column). For the sake of clarity, we label key statements to represent program counters. The example code includes method call and return, array initialization, and various instructions that can be translated into simpler μ-Dalvik instructions. Note that μ-Dalvik's `return` statement does not have any operands; instead, there is a special register r_{ret} for holding method return values, so return values must be copied into r_{ret} before `return`, as depicted in the figure. (Notice that for the statement `return z`, the correct value is copied into r_{ret} immediately after the jump.)

This example demonstrates all three of μ-Dalvik's key differences from Dalvik. First, we can see that `const/16` and `const/4`, which both load constant values into registers, are translated into the same μ-Dalvik instruction, and similarly for `rem-int/lit16` and `rem-int/lit-8`. Second, we can see that the `aget-byte` instruction is translated into μ-Dalvik's generic array access instructions; the other variants, such as `aget`, `aget-boolean`, etc., would be translated similarly. Finally, this example shows how SymDroid translates the complex `fill-array-data` instruc-

```
match op with
...
| OP_CONST_4
| OP_CONST_16
| OP_CONST
| OP_MOVE
| OP_MOVE_16
| OP_MOVE_FROM16 →
  let [lhs; rhs] = opr in
  [Mu_move (lhs, rhs)]
```

(a) `const` and `move`

```
...
| OP_IPUT
| OP_IPUT_OBJECT
| OP_IPUT_BOOLEAN
| OP_IPUT_BYTE
| OP_IPUT_CHAR
| OP_IPUT_SHORT →
  let [lhs; rhs; fid] = opr in
  [Mu_move (i_fld rhs fid, lhs)]
```

(b) `iput`

```
...
| OP_FILL_ARRAY_DATA →
  let [r_a; off] = opr in
  let dat, sz = Dex.get_array_data dx off in
  let r_t = Dex.free_register cur_mtd in
  let helper opr (idx, inss) =
    let ins1 = Mu_move (r_t, idx) in
    let ins2 = Mu_move (a_acc r_a r_t, opr) in
    idx − 1, ins1 :: ins2 :: inss
  in snd (List.fold_right helper dat (sz−1, []))
```

(c) `fill_array_data`

Figure 3: OCaml implementation of translation (sketch).

tion, which loads an array appended to the end of the code segment, into a sequence of multiple μ-Dalvik move instructions.

Figure 3 shows portions of the SymDroid code to translate Dalvik bytecode into corresponding μ-Dalvik instructions. Figure 3a shows that various constant loading and move operations, whose only differences are operand sizes and kinds, are translated into a single μ-Dalvik move instruction. Figure 3b is similar, showing how all the different `iput` variants are translated into a μ-Dalvik move. Lastly, Figure 3c iterates through the supplemental data for a `fill-array-data` instruction, emitting the appropriate sequence of μ-Dalvik operations. Similar code is used for other array filling operations and for switch statements.

3. Symbolic Execution

In this section, we present a formalism for symbolic execution of μ-Dalvik, and discuss our implementation. Figure 4 provides several definitions used in our formalism. Figure 4a summarizes the domains used by our symbolic executor. There are three basic kinds of *values* v used in the semantics: constants (defined in Figure 1), *heap locations* ℓ, and *symbolic expressions* π or ϕ, which are comprised of symbolic variables and constants combined with unary, binary, and relational operators.

As the symbolic executor runs, it maintains a *program state* Σ, which includes a call stack C, path condition ϕ, heap H, and static field state S. The *call stack* is a list of *local states* comprising a program counter, method body, and register file mapping registers to values. (Notice that each method gets its own registers, and hence these are used for local variables.) The top of the call stack is on the left, and represents the state of the currently executing method.

The state also contains a *path condition* ϕ, which records which conditional branches have been taken thus far. (For clarity, we will use ϕ to denote a symbolic expression that is a path condition, and π for other symbolic expression.)

The *heap* maps locations to *memory blocks* β, which are either *objects* o, which record their class and field values, and *arrays* α, which record the array type and the values in the array. Finally, the *static field state* is a mapping from static field names to their values.

In what follows, we will write $\Sigma.x$ for the x component of Σ, e.g., $\langle C', \phi', H', S' \rangle.H = H'$. When we write $\Sigma.pc$, $\Sigma.b$, or $\Sigma.R$, we will mean those components of the current (top-most) local state in $\Sigma.C$. Similarly, we write $o.@c$ and $\alpha.@c$ for the class of an object or array type, respectively, and refer to object fields and array elements via $o[@f]$ and $\alpha[i]$, respectively. We also write Σ^+ to mean state Σ but with the program counter of the current local state incremented by one.

Figure 4b defines the usual Java subtype relation, which is the reflexive, transitive closure of the superclass and interface relations defined in the program. Note that Java allows covariant subtyping on arrays (SUBARR). This is statically unsound, and so Java dynamically tracks the type of each array and forbids writes of objects that are not subtypes of the contents type.

Finally, Figure 4c defines a convenience relation $\Sigma[\![rhs]\!] = v$ for evaluating the right-hand side of a move expression. These rules are straightforward: constants are evaluated to themselves, and registers, static fields, array accesses, and field accesses are evaluated based on the state Σ.

$$
\begin{array}{rcll}
\ell & \in & \textit{Heap locations} & \\
x & \in & \textit{Symbolic variables} & \\
\pi, \phi & ::= & x \mid c \mid \odot\,\pi \mid \pi \oplus \pi \mid \pi \oslash \pi & \text{Symbolic expressions} \\
\upsilon & ::= & c \mid \ell \mid \pi & \text{Values} \\
R & ::= & \{r \mapsto \upsilon, \ldots\} & \text{Register file} \\
L & ::= & pc, b, R & \text{Local state} \\
C & ::= & L \mid L :: C & \text{Call stack}
\end{array}
$$

$$
\begin{array}{rcll}
o & ::= & \langle @c;\ \{@f \mapsto \upsilon, \ldots\}\rangle & \text{Objects} \\
\alpha & ::= & @c[\upsilon, \ldots] & \text{Arrays} \\
\beta & ::= & o \mid \alpha & \text{Memory block} \\
H & ::= & \{\ell \mapsto \beta, \ldots\} & \text{Heap} \\
S & ::= & \{@f \mapsto \upsilon, \ldots\} & \text{Static field state} \\
\Sigma & ::= & \langle C, \phi, H, S\rangle & \text{Program state}
\end{array}
$$

(a) Semantic domains.

$\boxed{@c \leq_P @d}$

$$\textsc{SubRefl}\ \frac{}{@c \leq_P @c} \qquad \textsc{SubTrans}\ \frac{@b \leq_P @c \quad @c \leq_P @d}{@b \leq_P @d} \qquad \textsc{SubSuper}\ \frac{}{@c \leq_P \mathsf{superclass}(P, @c)}$$

$$\textsc{SubItf}\ \frac{@d \in \mathsf{interface}(P, @c)}{@c \leq_P @d} \qquad \textsc{SubArr}\ \frac{@c \leq_P @d}{@c\ \mathsf{array} \leq_P @d\ \mathsf{array}}$$

(b) Subtyping.

$\boxed{\Sigma[\![rhs]\!] = v}$

$$\textsc{Ereg}\ \frac{}{\Sigma[\![r]\!] = \Sigma.R[r]} \qquad \textsc{Earr}\ \frac{\ell = \Sigma[\![r_a]\!] \qquad \alpha = \Sigma.H[\ell] \qquad i = \Sigma[\![r_i]\!]}{\Sigma[\![r_a[r_i]]\!] = \alpha[i]}$$

$$\textsc{Eobj}\ \frac{\ell = \Sigma[\![r_o]\!] \qquad o = \Sigma.H[\ell]}{\Sigma[\![r_o.@f]\!] = o[@f]} \qquad \textsc{Estt}\ \frac{}{\Sigma[\![@f]\!] = \Sigma.S[@f]} \qquad \textsc{Econst}\ \frac{}{\Sigma[\![c]\!] = c}$$

(c) Evaluation of right-hand sides.

Figure 4: Definitions used in symbolic execution.

Figure 5 gives the symbolic semantics for μ-Dalvik statements, which prove judgments of the form $\langle \Sigma, s\rangle \Downarrow_P \Sigma'$, meaning in program P, starting in state Σ, statement s updates the state to Σ'. The rules are mostly standard.

The rule SEGoto updates the program counter unconditionally. Rules SEIf-true and SEIf-false model conditional branches. Here SAT(ϕ) asserts that ϕ is satisfiable. In SEIf-true, we evaluate the guard and conjoin it with the current path condition. If the resulting path condition is satisfiable, it means the true branch is feasible, so we can branch to the specified program counter, and we update the path condition. SEIf-false is similar, permitting fall-through if the guard is satisfiable. Notice that these rules may be simultaneously valid, hence we have non-determinism in the semantics. As is standard in symbolic execution, we can choose whatever heuristics we like to decide whether to explore zero, one, or both possible branches.

Rules SEMove-reg evaluates the right-hand side subexpression and then updates the current register file. Rules SEMove-arr, SEMove-inst-fld, and SEMove-static-field are analogous, updating the appropriate array element, instance field, or static field. Rule SEMove-arr checks whether the given value is a subtype of the contents type, as mentioned earlier. Rules SEBop and SEUop compute a binary or unary expression and store the results in the appropriate register.

Rule SENew-obj allocates a new object in the heap, giving it the appropriate class and an empty set of fields. Note that we do not call a constructor here—Dalvik bytecode will contain an explicit call to method <init> to initialize any object fields. Rule SENew-arr is analogous, initializing the array elements with null values. Notice here we require that the type passed to `newarray` is an array type, which is also required in Dalvik [15]. Rules SECast and

$\boxed{\langle \Sigma, s\rangle \Downarrow_P \Sigma'}$

$$\textsc{SEGoto}\ \frac{}{\langle \Sigma, \texttt{goto}\ pc'\rangle \Downarrow_P \Sigma[pc \mapsto pc']}$$

$$\textsc{SEIf-true}\ \frac{\pi = (\Sigma[\![r_1]\!] \circledcirc \Sigma[\![r_2]\!]) \quad \phi_t = \pi \wedge \Sigma.\phi \quad \text{SAT}(\phi_t)}{\langle \Sigma, \texttt{if}\ r_1 \circledcirc r_2\ \texttt{then}\ pc_t\rangle \Downarrow_P \Sigma[\phi \mapsto \phi_t,\ pc \mapsto pc_t]}$$

$$\textsc{SEIf-false}\ \frac{\pi = \neg\,(\Sigma[\![r_1]\!] \circledcirc \Sigma[\![r_2]\!]) \quad \phi_f = \pi \wedge \Sigma.\phi \quad \text{SAT}(\phi_f)}{\langle \Sigma, \texttt{if}\ r_1 \circledcirc r_2\ \texttt{then}\ pc_t\rangle \Downarrow_P \Sigma^+[\phi \mapsto \phi_f]}$$

$$\textsc{SEMove-reg}\ \frac{\upsilon = \Sigma[\![rhs]\!] \quad R' = \Sigma.R[r \mapsto \upsilon]}{\langle \Sigma, r \leftarrow rhs\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEMove-arr}\ \frac{\upsilon = \Sigma[\![rhs]\!] \quad l = \Sigma[\![r_a]\!] \quad \alpha = \Sigma.H[l] \quad \alpha.@c = @c'\ \mathsf{array} \quad \upsilon.@c \leq @c' \quad i = \Sigma[\![r_i]\!] \quad H' = \Sigma.H[l \mapsto \alpha[i \mapsto \upsilon]]}{\langle \Sigma, r_a[r_i] \leftarrow rhs\rangle \Downarrow_P \Sigma^+[H \mapsto H']}$$

$$\textsc{SEMove-inst-fld}\ \frac{\upsilon = \Sigma[\![rhs]\!] \quad l = \Sigma[\![r_o]\!] \quad o = \Sigma.H[l] \quad H' = \Sigma.H[l \mapsto o[@f \mapsto \upsilon]]}{\langle \Sigma, r_o.@f \leftarrow rhs\rangle \Downarrow_P \Sigma^+[H \mapsto H']}$$

$$\textsc{SEMove-static-fld}\ \frac{\upsilon = \Sigma[\![rhs]\!] \quad S' = \Sigma.S\,[@f \mapsto \upsilon]}{\langle \Sigma, @f \leftarrow rhs\rangle \Downarrow_P \Sigma^+[S \mapsto S']}$$

$$\textsc{SEBop}\ \frac{\upsilon = \left(\Sigma[\![r_{s_1}]\!] \oplus \Sigma[\![r_{s_2}]\!]\right) \quad R' = \Sigma.R[r_d \mapsto \upsilon]}{\langle \Sigma, r_d \leftarrow r_{s_1} \oplus r_{s_2}\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEUop}\ \frac{\upsilon = \odot(\Sigma[\![r_s]\!]) \quad R' = \Sigma.R[r_d \mapsto \upsilon]}{\langle \Sigma, r_d \leftarrow \odot\, r_s\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SENew-obj}\ \frac{o = \langle @c, \emptyset\rangle \quad \ell \notin dom(\Sigma.H) \quad H' = \Sigma.H[\ell \mapsto o] \quad R' = \Sigma.R[r_o \mapsto \ell]}{\langle \Sigma, r_o \leftarrow \texttt{new}\ @c\rangle \Downarrow_P \Sigma^+[H \mapsto H'][R \mapsto R']}$$

$$\textsc{SENew-arr}\ \frac{j = \Sigma[\![r_i]\!] \quad \alpha = \langle @c\ \mathsf{array}, [\overbrace{\texttt{null}, \ldots}^{j}]\rangle \quad \ell \notin dom(\Sigma.H) \quad H' = \Sigma.H[\ell \mapsto \alpha] \quad R' = \Sigma.R[r_a \mapsto \ell]}{\langle \Sigma, r_a \leftarrow \texttt{newarray}\ @c\ \mathsf{array}[r_i]\rangle \Downarrow_P \Sigma^+[H \mapsto H'][R \mapsto R']}$$

$$\textsc{SECast}\ \frac{\beta = \Sigma[\![r_s]\!] \quad \beta.@c \leq_P @c' \quad R' = \Sigma.R[r_d \mapsto \beta]}{\langle \Sigma, r_d \leftarrow (@c')\ r_s\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEInstance-of}\ \frac{\beta = \Sigma[\![r_s]\!] \quad R' = \Sigma.R[r_d \mapsto (\beta.@c \leq_P @c')]}{\langle \Sigma, r_d \leftarrow r_s\ \texttt{instanceof}\ @c'\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SECall-static}\ \frac{b_m, r_i = \mathsf{lookup}(P, @m) \quad R' = \{r_i \mapsto \Sigma[\![r_1]\!], \ldots, r_{i+n-1} \mapsto \Sigma[\![r_n]\!]\} \quad C' = \langle 0, b_m, R'\rangle :: \Sigma.C}{\langle \Sigma, @m(r_1, \ldots, r_n)\rangle \Downarrow_P \Sigma^+[C \mapsto C']}$$

$$\textsc{SECall-dyn}\ \frac{\ell = \Sigma[\![r_{this}]\!] \quad o = \Sigma.H[\ell] \quad b_m, r_i = \mathsf{lookup}(P, @m, o) \quad R' = \{r_i \mapsto \ell, r_{i+1} \mapsto \Sigma[\![r_1]\!], \ldots\} \quad C' = \langle 0, b_m, R'\rangle :: \Sigma.C}{\langle \Sigma, r_{this}.@m(r_1, \ldots, r_n)\rangle \Downarrow_P \Sigma^+[C \mapsto C']}$$

$$\textsc{SEReturn}\ \frac{C :: C' = \Sigma.C \quad R' = C'.R[r_{ret} \mapsto \Sigma[\![r_{ret}]\!]]}{\langle \Sigma, \texttt{return}\rangle \Downarrow_P \Sigma[C \mapsto C'][R \mapsto R']}$$

$$\textsc{SESym}\ \frac{\mathit{fresh}(x) \quad R' = \Sigma.R[r \mapsto x]}{\langle \Sigma, r \leftarrow \texttt{sym}\rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEAssert}\ \frac{\neg\text{SAT}(\neg\Sigma[\![r]\!])}{\langle \Sigma, \texttt{assert}\ r\rangle \Downarrow_P \Sigma^+}$$

Figure 5: Symbolic semantics for μ-Dalvik statements.

SEInstance-of check subtype relations defined in Figure 4b, and either allow the cast or return the appropriate boolean value. Note that, for simplicity, we do not model exceptions in these semantics; hence a failed cast is simply not allowed, rather than raising an exception.

Rules SECall-static, SECall-dyn and SEReturn model method call and return. Both method call rules look up the appropriate method, in the dynamic case from the receiver object. We omit the definition of lookup, which is standard. The Dalvik virtual machine conforms to the ARM architecture's calling convention, in which the caller and callee share part of their register files; thus, the caller passes arguments by setting the appropriate range of registers. We do the same in μ-Dalvik, to make the translation from Dalvik to μ-Dalvik simple. We assume here the lookup function returns the first register r_i that should be set as a parameter. In dynamic dispatch, that first register is set to the receiver object. In both cases we advance the current program counter (so that return will continue at the correct instruction) and push another frame onto the call stack. Rule SEReturn models return, which copies the value from a special return register r_{ret} from the callee back to the caller, and pops the call stack.

Finally, the last two rules are for symbolic execution. The rule SESym introduces a fresh symbolic variable, and

```
module IntMap = Map.Make(int)
type value_ =                                  (* v *)
  | Const of const                             (* c *)
  | Loc   of loc                               (* ℓ *)
  | Sym   of SMT.exp                           (* π *)
type regs = value_ IntMap.t                    (* R *)
type l_state  = pc * instr  list  * regs       (* L *)

type c_stack = l_state  list                   (* C *)
type block =                                   (* β *)
  | Obj of (id_c * value_ IntMap.t)            (* o *)
  | Arr of (id_c * value_ IntMap.t)            (* α *)
type heap = block IntMap.t                     (* H *)
type static  = value_ IntMap.t                 (* S *)
type state = c_stack * SMT.exp * heap * static (* Σ *)
```

(a) Semantic domains.

```
val deref_H : state → loc → block
val adv_pc : state → state
val upd_pc : state → pc → state
val upd_R : state → reg → value_ → state
val upd_H : state → loc → block → state
val upd_o : block → id_f → value_ → block

let step (p: dex) (st: state): state * state option = function
  | Mu_move (lh, rh) →
    let v = eval st rh in
    ( match lh with
      | Register r → adv_pc (upd_R st r v)
      | InstFld (ro, f) →
        let l = eval st ro in
        let o = deref_H st l in
        let o' = upd_o o f v in
          adv_pc (upd_H st l (Obj o')), None
      | ... )
  (* definition of step cont'd ... *)
  | Mu_sym r →
    let x = SMT.fresh_var () in
      adv_pc (upd_R r (Sym x)), None
  | Mu_if (r1, cmp, r2, pc) →
    let v1 = eval st r1 in
    let v2 = eval st r2 in
    let pi_t = ... in
    let pi_f = ... in
    let sat = SMT.query pi_t in
    let n_sat = SMT.query pi_f in
      ( match sat, n_sat with
        | true, true → upd_pc st pc, Some (adv_pc st)
        | true, false → upd_pc st pc, None
        | false, true → adv_pc st, None
        | false, false → raise Infeasible )
  | ...
```

(b) Symbolic semantics.

```
val worklist : state Queue.t

let vm (p: dex) (drv: mu_instr list ) =
  let local_st = 0, drv, IntMap.empty in
  let init_st =
    local_st , SMT.true, IntMap.empty, IntMap.empty in
  Queue.add init_st worklist ;
while not (Queue.is_empty worklist) do
  let st = Queue.pop worklist in
  let ins = Dex.get_ins dx st.pc in
  let st1, so = step st ins in
  Queue.add st1 worklist;
  match so with Some st2 →Queue.add st2 worklist | _ → ()
done
```

(c) Symbolic execution driver.

Figure 6: OCaml implementation of symbolic execution (sketch).

SEAssert checks that the argument assert is always true (i.e., that its negation is not satisfiable).

We omit the remaining rules for executing a whole program, as they are straightforward. Note that, unlike standard Java programs, Android programs do not have a single `main()` entry point; thus, to symbolically execute an app, we must create a "driver" that mimics the way the Android system runs an app. Full details will be discussed in Section 4.

Implementation. Our prototype implementation, SymDroid, implements the formal symbolic execution rules just described. SymDroid uses apktool [16] to unpack apk files (which contain bytecode and other application resources), and the Dr. Android [17] tool to parse and represent Dalvik bytecode. SymDroid uses Z3 [18], an SMT solver, to check satisfiability of path conditions and assertions. In total, SymDroid comprises approximately 6,700 lines of OCaml code.

Figure 6 sketches our implementation, which follows the formal system very closely. Figure 6a shows the OCaml definitions matching the formal semantic domains from Figure 4a; we omit some definitions of primitive types such

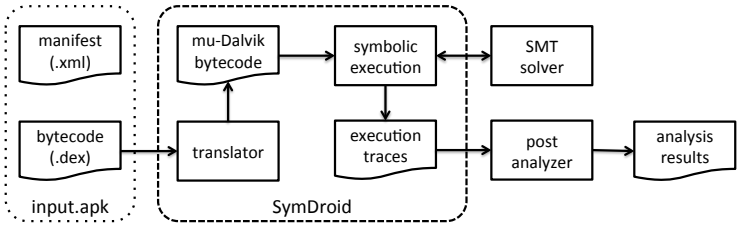

Figure 7: SymDroid architecture.

as pc etc.. Notice that the representation of symbolic expressions comes from the SMT solver (type SMT.exp).

Figure 6b lists the types of some utility functions and then gives a partial definition of the step function, which corresponds to the $\langle \Sigma, s \rangle \Downarrow_P \Sigma'$ relation in Figure 5. The input to step is a Dalvik bytecode file (of type dex), a program state, and a bytecode instruction, and the output is a pair containing a state and a state option; the latter is None in all cases except at a conditional branch when both branches are possible.

We give code for a few of the instruction handlers, for illustration. The first case, for Mu_move, evaluates the right-hand side and then updates the state appropriately depending on whether the left-hand side is a Register (SEMOVE-REG), an instance field (SEMOVE-INST-FLD), and so on. The second case, for Mu_sym, gets a fresh symbolic variable from the SMT solver and updates the corresponding register. Finally, the last case, Mu_if, checks satisfiability of the guard and the negated guard cojoined with the current path condition, and then returns the state updated with the new pc(s). Notice in the code for Mu_if, the last match case, when both branches are unsatisfiable, should never occur unless there is a bug in SymDroid.

Finally, Figure 6c shows the function vm that orchestrates the whole symbolic execution process. It maintains a (mutable) queue worklist of states to explore. After adding the initial state (which sets the pc to the beginning of the code passed as drv), the vm function repeatedly picks a state off the worklist, single-steps it, and then updates the worklist with the resulting state(s). Notice that this implementation explores *all* possible program paths; in practice we must carefully limit the use of the Mu_sym instruction so that full path exploration is feasible. On the other hand, it would be very easy to modify this driver to include heuristics for exploring a subset of paths [10, 11, 19].

Note also one other important design decision here: The state of the symbolic executor is fully captured in state, which is a purely functional data structure. This makes path exploration very easy, since we can explore executions in any order. In contrast, symbolic executors that actually run the program under test on the underlying system [8, 9] must be careful that side effects from different executions do not interfere with each other (see Section 6 for more discussion).

Additional features of implementation. Our formalism includes almost every feature in our implementation (and thus almost every feature of Dalvik), except for two. First, we omitted Dalvik's `array-length` instruction; SymDroid includes the same instruction, and its semantics is straightforward to implement. (Using a separate instruction rather than making it a unary operator was an arbitrary choice.) Second, we omitted exception handling and propagation from our formalism, but these can be supported with some minor tedium: First, we need to attach exception handlers to method definitions, and add a rule that searches for an appropriate handler and changes the control-flow accordingly when an exception is raised. Second, for the case when an exception is raised but there is no handler, we need a rule to propagate that exception to the caller. SymDroid includes both of these features and the corresponding instruction `throw`. Recall that, compared to more than 200 Dalvik bytecode instructions, μ-Dalvik has just 16 instructions, which are made up of 14 instructions shown in the syntax as well as `array-length` and `throw` instructions.

SymDroid Architecture. Figure 7 shows SymDroid's architecture. SymDroid receives an input apk file that contains the Dalvik bytecode (in a Dalvik executable (.dex) file). The Dalvik bytecode is first translated into μ-Dalvik, which is then processed by the symbolic execution core. The symbolic executor calls out to the SMT solver (recall this is Z3 [18]) as needed, and prints out execution traces as an intermediate result. Finally, the post analyzer inspects the output traces and summarizes the final result (e.g., to report basic summary statistics).

4. Modeling the Android Platform

In order to symbolically execute Android apps, we not only need to model each bytecode instruction, but we also need to model the platform that apps run on top of. Modeling the platform is challenging even for C programs [10, 20], but in our opinion it is even harder for Android, as there are many system libraries; the platform itself is quite large and complex; apps have several different entry points; and the interaction with the Android framework is quite involved, with various layers of callbacks.

One approach to modeling Android would be to pull in the bytecode for the large chunks of Android that are written in Java, and then symbolically execute it along with the app. However, in our experience [12, 19, 20], this is significantly harder than it sounds, for three reasons. First, there are so many interdependencies among parts of the system that it is hard to only pull in a small piece of the system at a time, which significantly increases the logistical challenges. Second, Android includes significant amounts of native code and runs on top of Linux; SymDroid cannot execute either of these, and so we would have to write models for those portions of the system in any case. Finally, in our experience real systems code contains many special cases for performance and unusual circumstances; symbolically executing this code can simply be too expensive, and induce too much branching, to be practical.

Thus, in our current work, we take a pragmatic approach to modeling Android, manually implementing only as much of a model as we need to carry out our particular case study (Section 5). We leave as future work the challenge of more fully modeling Android. There are three main portions of our current model: system libraries, system services and views, and the component lifecycle.

4.1. System Libraries

On Android, third-party libraries are statically linked with apps, but system libraries and the Java standard libraries are loaded at run time to reduce app size. Thus, SymDroid includes the ability to add "hooks" in for rules SEcall-static and SEcall-dyn that are invoked when the target method body is not in the app code (and thus it must be dynamically linked, assuming type safety). These hooks then transfer control to OCaml code that implements the desired functionality. Note that this is used when SymDroid needs to handle some method call specially; if the handler for a call can be written purely in Java, then we can simply write a Java model of the code and link it in statically (see 4.3, below).

We found that two of the most important system classes to model are Intent and Bundle, which are used to pass information between the system and an app, and between components of an app; SymDroid includes special internal support for both classes. In more detail, a Bundle is essentially a mapping from arbitrary string keys to values, and it is up to the sender and receiver of a Bundle to agree on the meaning of any particular element in the mapping. SymDroid stores Bundle keys and values in the field map for the Bundle object, using the negative hash values of the string keys as field "names" (so as not to conflict with other field ids; recall field ids are actual indices into the string pool). For example, assuming "aa" hashes to 153 and "bb" hashes to 90, a Bundle mapping those two fields to v_1 and v_2, respectively, would be represented as $\langle \mathsf{Bundle}, \{-153 \to v_1, -90 \to v_2\}\rangle$.

Intents are used to specify component names to launch. Intents may also include extra Bundle-style key-value mappings, e.g., added with intent.putExtra("aa", v_1). As with Bundles, we add those mappings directly to the field set of an Intent object.

In addition to the above two classes, SymDroid currently includes partial support for several commonly used Java libraries, including String, StringBuilder, Object, Class, and Integer. Analogously to Intent, we represent the actual contents of these objects using special fields, e.g., we use field __string__ to represent the underlying value of a String. The OCaml code hooked into method calls to these objects then manipulate the internal fields appropriately.

4.2. Runtime Instances

In the process of building a model of the Android platform, we found that several key methods in Android return a variety of different object types, depending on their arguments. For example, the following code sets an event handler for a button in a view:

```
setContentView(R.layout.start);
Button b = (Button)findViewById(R.id.startButton);
b.setOnClickListener (...);
```

```
class Driver {
  public static void main(String [] args) {
    String comp = "Lcom/.../PickContact;";
    Object o = Mock.__new__(comp);
    Mock.__invoke__(o, "onCreate", null);
    Mock.__click_rand__ ();
  }

  // android.app.Activity.startActivityForResult (...)
  public static void startActivityForResult
    (Object receiver, Intent i, int req) {
    ... // the designated Activity is invoked
    Object res = Mock.__new_sym__("res");
    Mock.__invoke__(receiver, "onActivityResult", req, res, i);
  }
```

```
  /* Driver cont'd */
  // android.content.ContentResolver.query(...)
  public static Cursor query(Object receiver, Uri uri, ...) {
    String contacts = "content:// com.android.contacts";
    assert(!uri.getPath().startsWith(contacts));
    ... // invokes the corresponding system API
} }
class Mock {
  public static Object __new__(String ty) { }
  public static Object __invoke__
    (Object _this, String mtd, Object... args) { }
  public static void __click_rand__ () { }
  public static Object __new_sym__(String var) { }
  ...
}
```

Figure 8: Example client-oriented specification.

Notice that findViewById could in general return many different kinds of objects (e.g., Button, EditText, Spinner, etc.), depending on the particular id value passed as an argument. Moreover, the association between ids and objects is typically stored in an XML file that is part of the app's apk file but separate from the app code. Thus, it is non-trivial to determine the exact object returned from findViewById without implementing XML parsing and a significant chunk of the GUI logic in Android.

Rather than try to mock the GUI logic, we observe that in the example above (and in practice), the result of findViewById and similar methods is always downcast to the actual final object type; thus we can use the cast itself to determine the object to return. In particular, we extend our state Σ to include a mapping from resource identifiers to View objects such as Buttons. At a call to findViewById, we return a dummy object that tracks the resource id. When we see the cast (e.g., to Button), we create a new Button instance, add it to the mapping in Σ, and return that from the cast. Later on, if we see a request for the same resource id, we return the same object.

We can use the same trick to model Android system services that, rather than being accessed via static methods, are provided through instance methods. For example, the following code creates a LocationManager instance and uses it to find the current location of the device:

```
String name = Context.LOCATION_SERVICE;
LocationManager lm = (LocationManager)getSystemService(name);
Location l = lm.getLastKnownLocation(...);
```

As above, getSystemService could return several different kinds of managers (e.g., LocationManager, TelephonyManager, etc.). Rather than model the mapping between service names (such as Context.LOCATION_SERVICE) and their classes, we determine the kind of service instance to create using the downcast.

4.3. Component Lifecycle

As mentioned above, Android apps run under quite a different model than standard desktop applications. Rather than have a main method at which execution begins, Android apps instead declare (in an XML "manifest" file) which components respond to which Intents, and apps begin execution at these points when the system's ActivityManager receives a corresponding Intent. These Intents could be sent from another app (e.g., apps often use this feature to launch the web browser to show a particular web page) and are even sent when starting an app from the home screen: tapping an app's icon sends an Intent to the app's launcher activity.

Moreover, even once an app is launched, apps are largely event-driven. Apps dynamically register various event handlers (e.g., for GUI events or for handling additional Intents), and control flow alternates between app code and the system's event dispatch loop. This is again in contrast to more standard, non-reactive systems.

For symbolic execution purposes, we need to model all of this behavior. In this paper, we chose to use *client-oriented specifications* [21] (co-spec) to model the system side of an app's execution. It is up to the developer to write such specifications so that they drive the app under test as desired.

For example, consider the specification code in Figure 8, which we will use in our case study (Section 5.2). This code defines a class Driver with a main() method; SymDroid uses this as the entry point for symbolic execution. The main() method first launches the PickContact activity of the app under test by calling its onCreate method [22]. In turn, this method (whose code appears later in Figure 11 and will be discussed in more detail then) registers several callbacks for button clicks, and then passes control back to the system.

Now SymDroid continues with Driver.main(), which clicks on a non-deterministically chosen button. (This is a symbolic execution branch point, where symbolic execution will fork for all possible button clicks.) A simulated button click turns into a callback to the handler that was registered for that button. In turn, that handler creates a new Intent and then passes control back to the system with a call to startActivityForResult(); the Intent includes a call back that should be invoked by the system once the activity is displayed on screen. SymDroid treats this call specially, routing it to the corresponding method in Driver. Inside of that method, we create a fresh symbolic variable (named "res") to represent the contact returned from the user's selection and invoke the onActivityResult() method for the Activity with this selection. In turn, that callback invokes a system-supplied query method to get available contacts. Again, SymDroid recognizes this specially, and routes that call to Driver's query() method. Finally, inside the query() code, we added an assertion that the particular query was not for contacts. Thus, if SymDroid explores the program and finds that this assertion has failed, then we have identified a path on which the contacts permission was actually used to query the contacts database.

Putting this all together, the overall control flow is

$$\begin{aligned}&\text{Driver.main()} \rightarrow \text{PickContact.onCreate()} \rightarrow \text{PickContact\$ResultDisplayer.onClick()} \rightarrow \\ &\quad \text{Driver.startActivityForResult()} \rightarrow \text{PickContact.onActivityResult()} \rightarrow \text{Driver.query()}\end{aligned}$$

where PickContact is the activity under test.

Notice that there is quite a complex interleaving between the subject app and the system, even in this very simple example. Moreover, we found it necessary to model this interleaving, as there is both state shared across various event handlers and state captured in the sequence in which event handlers are called.

In SymDroid, co-specs are written and compiled into standalone bytecode files, and we use Dr. Android's bytecode merging capability [17] to combine the target app and the co-spec into a single, analyzable Dalvik bytecode file. Notice that in the example, Driver performs invocations using class Mock, which has various unimplemented methods. This class is specially recognized by SymDroid, which ignores Mock method bodies and instead performs the action specified by the method name, e.g., creating a new instance of the given type string, invoking a method, etc.. We use Mock rather than calling app methods directly because doing the latter would require linking against the app code, which would be complicated because we expect SymDroid may often be used without direct access to app source code.

5. Experiments

We performed two kinds of experiments to evaluate SymDroid. First, we ran SymDroid against the Android Compatibility Test Suite (CTS) [13], which tests whether a Dalvik virtual machine implementation is correct. Our results suggest that SymDroid's translation to μ-Dalvik and semantics thereof are correct. Second, we used SymDroid to determine the conditions under which certain privileged system calls would be invoked by a chosen activity in a target app. This case study, while preliminary, demonstrates how SymDroid might be used in practice.

5.1. Compatibility Test Suite

We ran SymDroid against the Compatibility Test Suite version 4.0, which contains 93 test cases. We found that SymDroid passes 26 of the test cases. We manually inspected the failing test cases and concluded that all of them were due to unimplemented system libraries (recall we only implemented as much of Android as needed for our case study). Thus, despite the seemingly low coverage, SymDroid passed all of the CTS tests it could be expected to pass without a complete system model. We leave implementing the remaining libraries (including reflection, various I/O Streams and Buffers, the System class, and several others) as future work.

Next, we compared the performance of SymDroid (compiled to native code with OCaml version 3.12.1) to a Java virtual machine (Java 1.6.0_33) and a Dalvik virtual machine (the Dalvik VM from the Android source branch 4.0.4

Name	LoC	DEX (B)	# Ins	DVM (s)	SymDroid (s)	JVM (s)
005-args	20	2,004	121	0.066	0.139 (2.1x)	0.257 (3.9x)
006-count10	8	720	10	0.072	0.124 (1.7x)	0.261 (3.6x)
007-exceptions	25	1,232	26	0.068	0.133 (2.0x)	0.249 (3.7x)
008-instanceof	63	2,684	102	0.070	0.122 (1.7x)	0.292 (4.2x)
009-instanceof2	59	2,380	64	0.069	0.154 (2.2x)	0.259 (3.8x)
012-math	78	2,696	382	0.062	0.120 (1.9x)	0.263 (4.2x)
013-math2	10	940	15	0.064	0.128 (2.0x)	0.261 (4.1x)
015-switch	80	2,576	217	0.065	0.126 (1.9x)	0.249 (3.8x)
017-float	14	1,212	53	0.065	0.126 (1.9x)	0.260 (4.0x)
019-wrong-array-type	13	960	18	0.066	0.124 (1.9x)	0.249 (3.8x)
022-interface	52	2,080	50	0.077	0.121 (1.6x)	0.302 (3.9x)
026-access	14	952	15	0.063	0.115 (1.8x)	0.248 (3.9x)
029-assert	12	1,276	29	0.066	0.121 (1.8x)	0.255 (3.9x)
034-call-null	14	1,188	28	0.072	0.128 (1.8x)	0.279 (3.9x)
038-inner-null	24	1,680	31	0.066	0.123 (1.9x)	0.251 (3.8x)
040-miranda	58	2,612	151	0.063	0.125 (2.0x)	0.289 (4.6x)
043-privates	33	1,816	105	0.061	0.123 (2.0x)	0.247 (4.0x)
047-returns	46	1,868	83	0.065	0.121 (1.9x)	0.263 (4.0x)
052-verifier-fun	90	2,276	80	0.067	0.124 (1.9x)	0.262 (3.9x)
056-const-string-jumbo	6	1,158,088	17	0.069	3.207 (46x)	0.248 (3.6x)
076-boolean-put	20	1,580	31	0.063	0.118 (1.9x)	0.251 (4.0x)
081-hot-exceptions	23	1,688	45	0.066	0.129 (2.0x)	0.284 (4.3x)
085-old-style-inner-class	25	2,120	87	0.067	0.121 (1.8x)	0.255 (3.8x)
090-loop-formation	31	1,488	94	0.067	0.494 (7.1x)	0.280 (4.0x)
091-deep-interface-hierarchy	48	5,396	10	0.070	0.122 (1.7x)	0.319 (4.6x)
095-switch-MAX_INT	9	964	12	0.067	0.121 (1.8x)	0.259 (3.9x)

Figure 9: Results for Android compatibility test suite.

as of July 2, 2012). Figure 9 summarizes the results for the 26 test cases that passed. For each test case, the figure lists its size (in terms of its Java source code), the size of the corresponding Dalvik bytecode file, and its number of Dalvik bytecode instructions. The next three columns report the test case's running time on the DVM, SymDroid, and JVM. The reported performance is the average of ten runs on a 1.8 GHz Intel Core i7 with 2 GB RAM, running 64-bit Ubuntu 12.04.

In almost every case, the DVM is the fastest, followed by SymDroid (about twice as slow), followed by the JVM (another factor of two slower). The one exception to this trend is 056-const-string-jumbo, for which SymDroid is dramatically slower than either the DVM or JVM. We investigated this further, and found that SymDroid's core is very fast in this case, and what is taking most of the time is unpacking the apk (which is extremely large). The DVM and JVM take a .jar file as input, and apparently need not pay the same cost. Nonetheless, SymDroid is surprisingly fast, and we expect its performance to be adequate in practice, especially as SymDroid will be run on desktop machines that are much more powerful than the mobile devices the DVM would more typically be run on.

5.2. Case Study: Finding Privileged Calls

There are many possible ways to use SymDroid, as the literature on symbolic execution in general suggests [4, 5, 6, 7, 8, 9, 10, 11, 12]. To get a sense for how SymDroid might be used in practice, we applied it to the problem of discovering under what conditions various privileged system calls could be made.

In more detail, Android's security model includes *permissions* that protect sensitive platform APIs, such as access to the Internet, telephony, GPS, and so on. At app installation time, the user is presented with the set of permissions requested by an app. The user can then decide to proceed with installation, in which case all permissions are granted to the app; or the user can abort installation. While this model shows the user *what* permissions apps request, it does not explain *why* those permissions are needed, and under what circumstances they will be used. With SymDroid, however, we can find this information out.

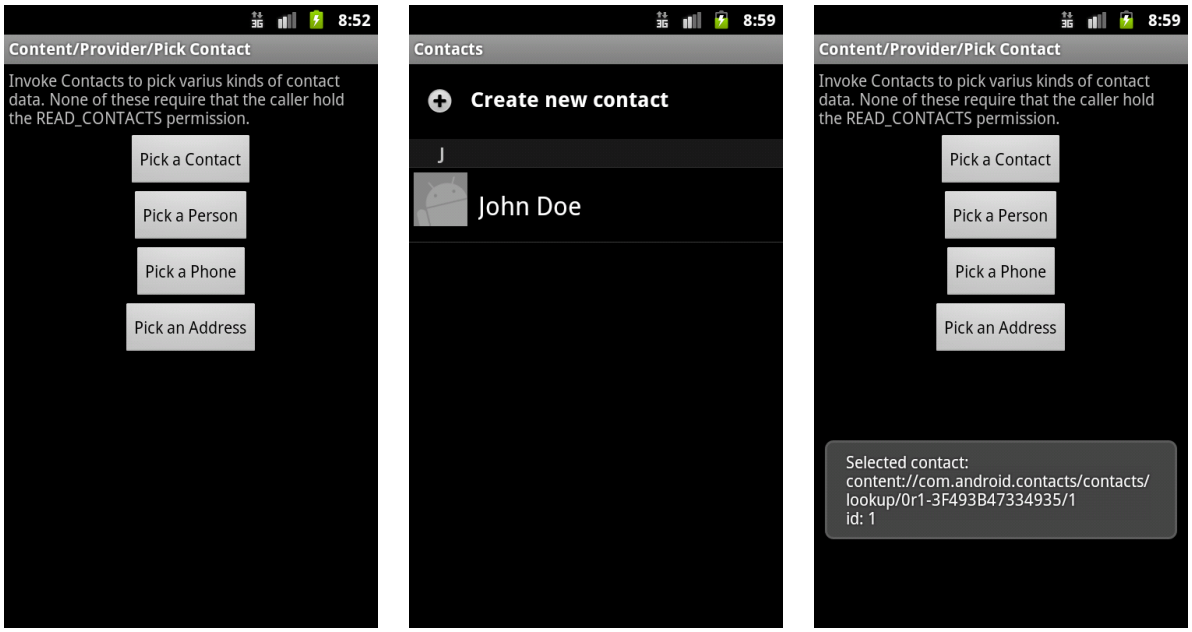

Figure 10: Sequence of screens in the PickContact Activity.

For purposes of this initial study, we decided it was particularly convenient to analyze an app whose source code was available. Thus, we elected to study the Android API demonstration app, which is included in the Android SDK [23]. We looked in detail at one of this app's activities (an Activity essentially corresponds to a screen shown to the user): PickContact, which lets the user select a single contact from the contacts database on the phone.

PickContact. Figure 10 shows a sequence of screenshots from PickContact.[1] On the left is the initial screen displayed when PickContact is launched within the demo app. The user is presented with four choices to filter the set of contacts that will be shown—any contact, those that are for a person, those with a phone number, or those with an address. In this case, we clicked on the *Pick a Contact* button. The app then sends an Intent to the standard Android contacts app, which launches that app (if it is not already running) and brings up the contact picker window, shown in the middle screenshot. We click to select a contact, and then control passes back to PickContact, which displays the URI for the selected contact on screen.

We wanted to use SymDroid to investigate under what conditions PickContact's READ_CONTACTS permission was used in this sequence of events. Somewhat confusingly, it is *not* used when the contact picker is launched, as that is done in a different app on the phone, which runs in its own process and has its own separate set of permissions. Thus, for example, if the user gets to the contact picker but then clicks the back button, PickContact will not try to read any contact information. The permission will only be used if the user actually selects a contact, in which case PickContact will query the contacts database. (The id returned from querying the contacts database is shown in the right screenshot in Figure 10.)

Figure 11 gives a portion of the source code for the PickContact activity. Recapping some of the earlier discussion, when this activity is started, its onCreate() method on line 15 is called. This method sets callbacks for the four buttons shown in the left screenshot in Figure 10; the code for setting one callback is shown on lines 16–19. In this case, the callbacks are instances of the ResultDisplayer class parameterized by the mime type of the contacts to select.

When a button is clicked, the corresponding callback is invoked, in this case calling the onClick() method on lines 7–13. This method then creates an Intent for the contact picker app (the Intent kind is specified on line 9) and launches it on line 12. When this returns, the system automatically calls onActivityResult() of the Intent sender

[1] The misspellings in the screenshots are that way in the app source code.

```
1  public class PickContact extends Activity {
2    class ResultDisplayer implements OnClickListener {
3      String mMimeType;
4      ResultDisplayer(String msg, String mimeType) {
5        mMimeType = mimeType;
6      }
7      public void onClick(View v) {
8        Intent intent =
9          new Intent(Intent.ACTION_GET_CONTENT);
10       intent.setType(mMimeType);
11       ...
12       startActivityForResult(intent, 1);
13     }
14   }
15   @Override protected void onCreate (Bundle saved) {
16     ((Button)findViewById(R.id.pick_contact)).
17       setOnClickListener(
18         new ResultDisplayer("Selected contact'',
19         ContactsContract.Contacts.CONTENT_ITEM_TYPE));
20     // set three more call-back listeners
21   }
22   @Override protected void onActivityResult
23     (int req, int res, Intent data) {
24     if (data != null) {
25       Uri uri = data.getData();
26       if (uri != null) {
27         Cursor c = getContentResolver().query(...);
28         ... }
29 } } }
```

Figure 11: PickContact source code (excerpt).

(line 22), which then performs the query call (line 25) that actually requires the READ_CONTACTS permission. Notice that this call will not occur if no contact is selected (e.g., if the user clicked the back button), as in that case data will be null. (The uri null check is an extra sanity check; that should always be non-null.)

We ran SymDroid against this program using the co-spec in Figure 8. Recall that in this case, there are four symbolic variables: three for onActivityResult parameters (req, res, data) and one for information retrieved from another symbolic variable (uri). SymDroid explored a total of 16 different paths, and 4 of them included a privileged call that used READ_CONTACTS:

```
privilege call :
  android.content.ContentResolver→query
    requires READ_CONTACTS
    where NOT(sym3 = 0x0) AND NOT(sym3.getData = 0x0)
```

We can see that the only path triggering the condition is along the path when neither the data (corresponding to sym3) nor uri (corresponding to sym3.getData, as it was derived by calling getData on sym3) fields are null. This corresponds to the case when the user did not close the contact picker without selecting a contact, and the contact they picked does indeed exist in the phone's database. The path condition does *not* include the check indicating that asserts the path begins with the URI specific to the contacts database. However, while this path condition is asserted, the co-spec uses only concrete instances of strings, rather than symbolic strings (as SymDroid currently does not support symbolic strings). We verified manually that this is the correct set of path conditions leading to privileged calls in this example.

Over all paths, SymDroid executed a total of 4,462 μ-Dalvik instructions, which included 54 system calls that were hooked specially by SymDroid. The average of ten runs on the same machine on which CTS tests were conducted is 30.93 seconds. This running time shows, again, that SymDroid is fast enough to analyze real apps.

6. Related Work

There are several threads of related work.

Symbolic execution. Many researchers have explored symbolic execution [4, 5, 6, 7] recently, with several promising results using symbolic execution to find bugs in software systems [8, 9, 10, 11, 12, 19]. Symbolic executors can roughly be divided into two kinds. The first kind, so-called *concolic* executors, perform symbolic execution at program run time by shadowing underlying concrete system values with symbolic expressions [8, 9]. There are two main advantages to this approach. First, in the case when a program has only a small amount of symbolic computation, concolic execution could be very fast, as much of the program will run on a true CPU, rather than being virtualized through an interpreter. However, our experience is that most of the time in symbolic execution is spent in the SMT

solver, which would be the same in either approach. Second, when faced with a call to unavailable (library or system) code, concolic executors can simply call the actual external code with the underlying concrete values, extending the path condition to constrain the corresponding symbolic expression to equal the concrete value. This process is called *concretization*, and it is highly scalable, but with two main drawbacks. First, side-effecting system calls (e.g., writes to the file system) cannot be undone without more effort, thus making forking and path exploration tricky. Second, concretization reduces opportunities for path exploration, as concolic execution can never fork within unknown code.

The second kind of symbolic executors are "pure" in the sense that they do not directly execute the subject program on the underlying platform. SymDroid, KLEE [10], and Otter [12] are examples of this kind of symbolic executor. The main drawback to this approach is the significant effort required to model the underlying system, and the potential reduction in performance.

Orthogonally to the type of symbolic executor, another key research area has been search strategies to allow symbolic executors to find "interesting" executions to explore, since in practice symbolic execution cannot cover all paths. KLEE [10] uses a round-robin-based heuristic that attempts to reach the closest uncovered nodes in the control-flow graph. SAGE [11] maintains a coverage-guided worklist to explore execution paths in a generational order. In prior work, we explored shortest-distance symbolic execution and call-chain-backward symbolic execution to target particular lines of interest [19]. Currently, SymDroid does not include any strategy, but it would be easy to incorporate them into the symbolic execution driver (shown in Figure 6c). Researchers have also begun exploring how to symbolically execute multi-threaded programs [24, 20]. As many Android apps include some threading (although typically not in the main part of the app, which is single-threaded), these techniques could be useful for SymDroid.

Symbolic execution for Android. The most closely related work to SymDroid is ACTEve [25], a concolic executor for Android apps. The key contribution of ACTEve is mimicking user interactions by automatically generating event sequences. SymDroid, in contrast, requires the user to write a driver to reflect app usage. ACTEve uses ded [2, 3] to translate from Dalvik to Java bytecode, and then performs symbolic execution inside of Soot. It is unclear how ACTEve deals with native code, particularly the Android runtime, as it does not run on top of Android.

Android app analysis. Researchers have developed many different static analyses for Android apps; we discuss a few here. Barrera et al. [26] empirically analyzed permission usage patterns using self-organizing maps (SOMs). They found that only a small number of permissions are widely acquired but that some of these are overly broad. Stowaway [27] is a static analysis tool that checks whether acquired permissions are actually used; the Stowaway authors found that many apps are over-privileged. ComDroid [28] finds vulnerabilities related to inter-app communications. Ded [2, 3], a Dalvik-to-Java decompiler, has been used to discover security vulnerabilities. Woodpecker [29] uses data-flow analysis to find capability leaks on Android phones. These tools all use other styles of static analysis (typically data flow analysis). SymDroid, which uses symbolic execution, could potentially provide much more precise information about apps; we leave exploring these ideas to future work.

Several researchers have developed dynamic analysis tools to identify specific security vulnerabilities in Android apps. TaintDroid [30] tracks the flow of sensitive information and looks for confidentiality violations. QUIRE [31], IPC Inspection [32], and XManDroid [33] aim to prevent privilege-escalation, in which an app is tricked into providing sensitive capabilities to another app. As symbolic execution is in many ways close to a dynamic analysis, it may be possible to use it to check similar properties of apps.

7. Conclusion

In this paper, we presented SymDroid, a symbolic executor for Dalvik bytecode. SymDroid actually operates on μ-Dalvik, a language with far fewer instructions than Dalvik, and to which Dalvik can be easily translated. In addition to modeling bytecode instructions, SymDroid includes limited support for system libraries including Bundle and Intent, two critical classes used for communication on Android. Since Android apps are event-driven, we use client-oriented specifications to model the system and drive the app under test in the desired ways. Running SymDroid against the Android Compatibility Test Suite, we found it passed all test cases that did not require more system modeling, and was only about twice as slow as the Dalvik VM running on the same machine. We also used SymDroid to discover the conditions under which the PickContact activity in the API demonstration app actually used contacts. These results suggest that, while still a prototype, SymDroid is a promising first step in direct, precise analysis of Android apps.

Acknowledgement

This research was supported in part by NSF CNS-1064997 and by a research award from Google.

References

[1] Gartner, Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent, http://www.gartner.com/it/page.jsp?id=2237315 (Nov. 14 2012).
[2] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A Study of Android Application Security, in: USENIX Security Symposium, 2011.
[3] D. Octeau, S. Jha, P. McDaniel, Retargeting Android Applications to Java Bytecode, in: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, FSE-20, 2012.
[4] J. C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.
[5] R. S. Boyer, B. Elspas, K. N. Levitt, SELECT–a formal system for testing and debugging programs by symbolic execution, in: International Conference on Reliable Software (ICRS), 1975, pp. 234–245.
[6] W. E. Howden, Symbolic testing and the DISSECT symbolic evaluation system, IEEE Transactions on Software Engineering 3 (4) (1977) 266–278.
[7] L. J. Osterweil, L. D. Fosdick, Program testing techniques using simulated execution, in: Symposium on Simulation of Computer Systems (ANSS), 1976, pp. 171–177.
[8] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05, 2005, pp. 213–223.
[9] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for c, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, 2005, pp. 263–272.
[10] C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI '08, 2008, pp. 209–224.
[11] P. Godefroid, M. Y. Levin, D. A. Molnar, Automated whitebox fuzz testing, in: Network & Distributed Security Symposium, NDSS '08, 2008.
[12] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, A. Porter, Using symbolic evaluation to understand behavior in configurable software systems, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10, 2010, pp. 445–454.
[13] Android, Compatibility Test Suite, http://source.android.com/compatibility/cts-intro.html.
[14] P. Brady, Anatomy & Physiology of an Android, https://sites.google.com/site/io/anatomy--physiology-of-an-android.
[15] Android, Bytecode for the Dalvik VM, http://source.android.com/tech/dalvik/dalvik-bytecode.html.
[16] Google, Tool for reengineering Android apk files, http://code.google.com/p/android-apktool/.
[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, T. Millstein, Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications, in: ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Raleigh, NC, USA, 2012, pp. 3–14.
[18] L. M. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: TACAS, 2008, pp. 337–340.
[19] K.-K. Ma, K. Y. Phang, J. S. Foster, M. Hicks, Directed symbolic execution, in: Proceedings of the 18th international conference on Static analysis, SAS '11, 2011, pp. 95–111.
[20] J. Turpie, E. Reisner, J. S. Foster, M. Hicks, MultiOtter: Multiprocess Symbolic Execution, Tech. rep., CS-TR-4982, Department of Computer Science, University of Maryland, College Park (Aug 2011).
[21] C. M. Hayden, S. Magill, M. Hicks, N. Foster, J. S. Foster, Specifying and verifying the correctness of dynamic software updates, in: Proceedings of the 4th international conference on Verified Software: theories, tools, experiments, VSTTE'12, 2012, pp. 278–293.
[22] Android, Application Fundamentals, http://developer.android.com/guide/components/fundamentals.html.
[23] Android, Android SDK, http://developer.android.com/sdk/index.html.
[24] S. Bucur, V. Ureche, C. Zamfir, G. Candea, Parallel symbolic execution for automated real-world software testing, in: Proceedings of the sixth conference on Computer systems, EuroSys '11, 2011, pp. 183–198.
[25] S. Anand, M. Naik, H. Yang, M. J. Harrold, Automated Concolic Testing of Smartphone Apps, in: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, FSE-20, 2012.
[26] D. Barrera, H. Kayacik, P. van Oorschot, A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to Android, in: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, 2010, pp. 73–84.
[27] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, 2011, pp. 627–638.
[28] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing Inter-Application Communication in Android, in: MobiSys, 2011.
[29] M. Grace, Y. Zhou, Z. Wang, X. Jiang, Systematic Detection of Capability Leaks in Stock Android Smartphones, in: Network & Distributed Security Symposium, NDSS '12, 2012.
[30] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in: OSDI, 2010.
[31] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, D. S. Wallach, Quire: Lightweight provenance for smart phone operating systems, in: USENIX Security Symposium, 2011.
[32] A. Felt, H. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses, in: USENIX Security Symposium, 2011.
[33] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, B. Shastry, Towards taming privilege-escalation attacks on android, in: Network & Distributed System Security Symposium, NDSS '12, 2012.