EasyChair Preprint

№ 10387

K-Smali: an Executable Semantics for Program Verification of Reversed Android Applications

Marwa Ziadia, Mohamed Mejri and Jaouhar Fattahi

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

June 13, 2023

𝕂-Smali: an Executable Semantics for Program Verification of Reversed Android Applications

Marwa Ziadia[0000-0002-8563-4736], Mohamed Mejri[0000-0003-4820-3176], and Jaouhar Fattahi[0000-0002-3905-9099]

Department of Computer Science and Software Engineering, Laval University, 2325, rue de l'Université, Québec (Québec) G1V 0A6, Canada.

Abstract. One of the main weaknesses threatening smartphone security is the abysmal lack of tools and environments that allow formal verification of application actions, thus early detection of any malicious behavior, before irreversible damage is done. In this regard, formal methods appear to be the most natural and secure way for rigorous and unambiguous specification as well as for the verification of such applications. In previous work, we proposed a formal approach to build the operational semantics of a given Android application by reverse engineering its assembly code, which we called *Smali*$^+$. In this paper, we rely on the same idea and we enhance it by using a language definitional framework. We choose 𝕂 framework to define Smali semantics. We briefly introduce the 𝕂 framework. Then, we present a formal 𝕂 semantics of Smali code, called 𝕂-Smali. Semantics includes multi-threading, threads scheduling and synchronization. The proposed semantics supports linear temporal logic model-checking that provides a suitable and comprehensive formal environment for checking a wide range of Android security-related properties.

Keywords: Android applications, 𝕂 framework, formal semantics, formal verification, Smali.

1 Introduction

Android platform users are increasingly exposed to attacks on the Android environment via untrusted applications. The McAfee 2020 report confirms that fake applications are the most active mobile threat category, generating almost half of all malicious telemetry, with a 30% increase from 2018 [1]. SMS trojan such as *AsiaHitGroup* and *GGTracker* are prime examples of attacks that manifest at the application level (e.g. *Fake Player* application). This may cause financial losses to the user by sending text messages to premium-rate numbers without their knowledge. Spying by taking photos, recording videos or audios, retrieving the history of the application, recording phone conversations, and tracking user location are among a large range of threats that jeopardize Android users through rogue applications. Several research initiatives have been recently put forward to handle these concerns. Their goal is mainly to detect misbehaving applications and to enforce the security within them. Nevertheless, it is virtually impossible to assess efficiency or deficiency or prove the validity of a given system in the absence of a formal specification. According to Stefanescu et al. [2], which we endorse, analysis tools for a programming language should be based on the formal semantics of that language. The informal semantics are subject to interpretation by different tool developers, and there is generally no guarantee that these interpretations are consistent with the specification. However, even in the presence of a formal specification, language definitional frameworks are highly needed. They are generally provided with a guideline to formalize a given language, which allows avoiding human errors and omissions that can slip in. They also permit to produce reliable results and to test the resulting formal semantics against a set of sample programs. Nevertheless, this type of framework should meet a set of criteria. Firstly, it must be easy to use and should provide human-readable semantics. Secondly, it must be sufficiently adapted to perform formal reasoning and provide automated proofs. Moreover, it should be able to define language-related features, such as concurrency. The framework should also be generic so that it is not tied to a specific language, and modular so that it does not need to be modified if new features are added. Ideally, the framework should also include its own analyzer for the language being analyzed so that the use of an external software program is not mandatory. The $\mathbb{K}$ framework [3] is a prime example of tools satisfying most of these requirements. It provides a user-friendly rewrite-based language for defining formal operational semantics of programming languages. Figure 1 shows different modules that can be applied on any language having $\mathbb{K}$ semantics, such as LTL model checking, symbolic execution, and program verification. The first thing we can notice is the large choice of tools (e.g. compilers, interpreters, state-space explorers, and test-case generators) that can be automatically derived from *one* reference formal definition of the language. It is a wise way to eliminate the need to squander resources on designing and implementing expensive custom tools. Different approaches define multiple semantics for one language, each designed for a purpose (e.g. program verification, symbolic execution, etc.), which is an uneconomical and labor-intensive way. In previous work [4], we put forward an operational semantics for Smali that we called *Smali*$^{+}$. Smali is an assembly-like code generated from reverse-engineering Android applications. *Smali*$^{+}$ includes the most important Dalvik features such as multithreading, method invocation, and object creation. However, this formal approach used to generate formal semantics is not executable, error-prone, and lacks a semantics engineering framework with the characteristics mentioned above. Furthermore, verifying and proving the correctness of such formal semantics requires manual development of custom tools (such as interpreters and compilers), usually with no guarantee. The resulting program may end up manifesting unexpected behaviors and, in some cases, leading to irreversible consequences [3]. In this paper, we choose the $\mathbb{K}$ framework for Smali code formalization. The main goal is to provide an executable and expressive formal semantics with which program analysis, security policy enforcement, and property verification can be performed. We name the resulting semantics $\mathbb{K}$-Smali. Additionally, using $\mathbb{K}$, the obtained semantics can be used to check security properties specified as Linear temporal logic (LTL) [5, 6] formulas. These properties reflect the healthy behaviors of an application that attacks, originating from SMS Trojans or Android spyware for example, try to transgress. For complete details on Smali, reverse engineering and compilation steps of an Android application, we kindly refer the reader to [4]. Our contribution consists of full-fledged semantics for Android applications, entirely compatible with the $\mathbb{K}$ framework so that it inherits all its powerful compilation, testing, and verification tools. We have been a lot motivated by different $\mathbb{K}$ semantics for several real languages such as Java [7], PHP [8], and C [9]. We see that they have been used as trusted reference models for the defined language.

This paper is organized as follows. In Section 2, we briefly introduce the $\mathbb{K}$ framework. In Section 3, we present $\mathbb{K}$-Smali. That is, we present the general and detailed configuration, syntax, and semantics rules. In Section 4, we indicate how we verify some important security-related properties using $\mathbb{K}$ on the derived semantics. Section 5 reviews and discusses related research work. Finally, in Section 6, we draw some conclusions and trace some directions for future work.

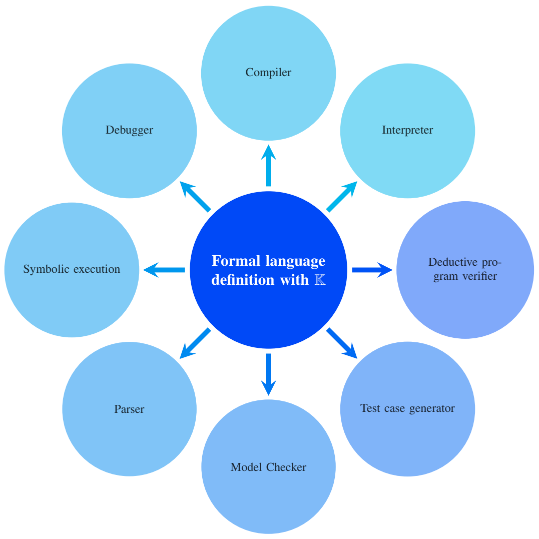


Fig. 1: $\mathbb{K}$ Framework features [10]

2 Overview on $\mathbb{K}$ framework

$\mathbb{K}$ is a rewriting-based definitional semantic framework for programming languages. It provides a complete methodology for their design and specification. The $\mathbb{K}$ specification consists of three main steps: a syntax definition, a configuration definition, and a semantic definition. Once these steps are completed and saved in files with *k* extension, the command *kompile* is used to compile each definition. The command *krun* invokes an interpreter with which program models can be simulated and tested. Several options can be added to this command to generate models on which formal verification tools for parsing, interpretation, deductive formal verification, symbolic execution, and model checking can be applied. Figure 1 illustrates all these features. Syntax in $\mathbb{K}$

is written with the conventional Backus-Naur Form (BNF) notation. Listing 1.1 represents an example of a $\mathbb{K}$ source file containing a program *P* syntax definition. As shown, non-terminals are starting with uppercase letters and preceded by the keyword *syntax*, whereas terminals are represented inside two quotes. For example, in line 2, a program *Pgm* is defined as a list of semicolon-separated instructions. Syntax declaration can be tagged with attributes. These attributes are specified in square brackets at the end of a given definition and are meant to provide additional information to the parser. The strictness constraint, for example, specifies how the arguments of the language construct should be evaluated. In line 3, the "strict(2)" attribute indicates to the parser that only the second argument (i.e. Exp) should be evaluated. When no number is provided with this attribute, such as in line 4, all argument positions are considered strict (i.e. they are evaluated in any fully interleaved order). $\mathbb{K}$ framework offers some basic types such as *Bool*, *Int*, *String*, *Float*, etc. as well as the *Id* type (Identifier), which facilitates the language specification.

```
1 module P-Syntax
2   syntax Pgm  ::= List "{" Inst ," ; " "}"
3   syntax Inst ::= Id " " Exp [strict (2)]
4   syntax Exp  ::= "mul" "(" Int Int")" [strict]
5 end module
```

Listing 1.1: An example of $\mathbb{K}$ syntax definition

Before defining semantic rules, $\mathbb{K}$ requires to set the structure of the program state by setting its configuration. It provides additional information (besides the syntax) about the definite language to better understand its semantics. Program states in $\mathbb{K}$ configurations are organized in units called cells. Cells are labeled and possibly nested. Each cell contains semantic information about the program, such as its context, memory, environment, etc. The cell content differs according to this information and can hold several algebraic data types such as maps, lists, sets, and trees. Figure 2 shows the generated graphical representation for a program *P* configuration. The notation inside the cells represents their initial state. Configuration consists of a top cell labeled $\top$, holding two sub-cells: a ***$ PGM*** variable cell of type *k*, used, by convention always for computation, a ***Memory*** cell holding a mapping form the program variables to values, initially empty. The asterisk symbol "*" used with the *Inst* sub-cell specifies its multiplicity.

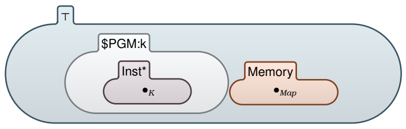


Fig. 2: $\mathbb{K}$ configuration example

Once syntax and configuration are defined, semantics rules should be set. Defining semantics for the language consists of a set of $\mathbb{K}$ rewrite rules that drive the execution of programs. One can describe a $\mathbb{K}$ rewrite rule as a transition over configurations, that starts with a configuration holding the original program and ends with a new one maintaining the result. Each rule in $\mathbb{K}$ is preceded by the keyword *rule* and has the following form:

$$\text{rule}\ \ lhs \Rightarrow rhs$$

where lhs represents the left-hand side of the rule and rhs is the right-hand side. Listing 1.2 provides an example of semantics definition. The module *P-semanitcs* represents the semantics definition of the *P* program presented in Listing 1.1. Line 3 represents a rewrite rule definition for multiplication. Notice the ellipses "..." in the rewrite rule definition. It is actually used for a volatile part of the term, which corresponds to the part that the current rule does not take into account.

```
module P-Semantics
imports P-Syntax
 rule <T> <PGM><Inst> I1:Int * I2:Int ⇒ I1 *Int I2 </Inst></PGM>... </T>
end module
```

Listing 1.2: An example of $\mathbb{K}$ semantics definition

The $\mathbb{K}$ rewrite rule defined in Listing 1.2 affects one cell in the program *P* configuration (i.e. *Inst* cell) as follows:

$$\text{rule}\ \left\langle \frac{\text{I1:}\mathit{Int} * \text{I2:}\mathit{Int}}{I1\ *_{Int}\ I2} \right\rangle_{Inst}$$

The line expresses a rewrite. Terms above and below the line represent the left-hand side (lhs) and the right-hand side (rhs) of the rule, respectively. The rest of the configuration context is inferred automatically.

3 $\mathbb{K}$-Smali

3.1 Syntax

As previously mentioned, formal modeling Smali code was the subject of earlier work [4]. To make this paper self-contained, this subsection details just the definitions, instructions, and terms that are newly considered in $\mathbb{K}$-Smali. Listing 1.3 corresponds to a $\mathbb{K}$ source file used to define $\mathbb{K}$-Smali. It provides basic syntactic categories and the syntax of selected instructions. Following the disassembly process, all internal source Java classes are separated from their including class, each class in a *.smali* file. The Manifest file allows the identification of the application's entry point. We suppose that its syntax consists only of the keyword *.manifest* followed by a method reference *MethodRef* referring to the method's full name as well as the fully qualified name of its including class. This method represents the entry point from which the program starts execution (line 78). Each class in the *.smali* file is defined by a class header *ClassHeader* indicating all information about the class: possible comments; its fully qualified name (starting always by "L" and ending by ";" line 33), its direct super-class fully qualified name (if exists), access flags indicating its visibility; its corresponding Java source class (identified by the *.source* keyword) and finally a set of implemented interfaces.

```
 1 module SMALI–SYNTAX
 2   syntax Program      ::= SmaliFiles ManifestFile
 3   syntax SmaliFiles  ::= List{SmaliFile," "}
 4   syntax SmaliFile   ::= Class
 5   syntax Class       ::= ClassHeader Fields Methods
 6   syntax ClassHeader::= Comments ".class" AccessFlags ClassName SuperClass
       SourceClass Interfaces
 7   syntax SuperClass ::= Comments ".super" SuperClassName | Empty
 8   syntax SourceClass::= Comments ".source" String | Empty
 9   syntax Comments ::= List{Comment," "}
10   syntax Comment   ::= r"\\#.*"                    [token]
11   syntax Fields ::= List{Field," "}
12   syntax Field ::= Comments ".field" AccessFlags FieldName ":" Type ValueOp
13   syntax ValueOp ::= Value | Empty
14   syntax Methods ::= List{Method," "}
15   syntax Method ::= Comments ".method" AccessFlags  MethodNameSign MethodBody ".
       end method"
16   syntax MethodNameSign ::= MethodName MethodSignature
17   syntax MethodSignature ::= MethodInTypes MethodRetType
18   syntax MethodInTypes ::= "(" Types ")" | "(" ")"
19   syntax MethodRetType ::= Type  | VoidType
20   syntax Type::= PrimitiveType|ObjectType|ArrayType
21   syntax PrimitiveType::="Z"|"B"|"C"|"D"|"F"|"I"|"J"|"S"
22   syntax VoidType    ::= "V" /* void type*/
23   syntax ObjectType ::= LName  /* Object reference*/
24   syntax ArrayType ::="["PrimitiveType|"["ObjectType|"[ArrayType"
25   syntax Value ::= Bool | Int | Float | String
26   syntax AccessFlags ::= List{AccessFlag," "}
27   syntax AccessFlag::= "public"|"private"|"protected" |"final"|"abstract"|"static"
28   syntax ClassName ::= LName
29   syntax SuperClassName::= LName
30   syntax MethodName ::= Name | "constructor" "<init>"
31   syntax FieldName::=Name
32   syntax Name ::= Id
33   syntax LName ::= r"L[_a-zA-Z0-9]*[_a-zA-Z0-9]*;"[token]
34   syntax MethodRef ::= ClassName"->" MethodNameSignature
35   syntax FieldRef  ::= ClassName"->" FieldName
36   syntax Parameters::= List{Parameter,","}
37   syntax Parameter ::= RegName
38   syntax MethodBody ::= List{Statement," "}
39   syntax Statement  ::= Instruction | Directive
40   syntax Instruction ::= "goto" ":" Label
41                         |":" Label
42                         |"nop"
43                         |"sparse-switch" RegName "," ":" Switchtab
44                         |"const" RegName "," Val
45                         |"const-string" RegName "," String
46                         |"move" RegName "," RegName
47                         |"new-instance" RegName "," ClassName
48                         |"new-array" RegName "," RegName "," ArrayType
49                         |Sget RegName "," FieldRef
50                         |Sput RegName "," FieldReference
51                         |"iget" RegName "," RegName "," FieldRef
52                         |"iput" RegName "," RegName "," FieldRef
53                         |"aget" RegName "," RegName "," RegName
54                         |"aput" RegName "," RegName "," RegName
55                         |"if-eq" RegName "," RegName "," ":" Label
56                         |"if-lt" RegName "," RegName "," ":" Label
57                         |BinOp RegName "," RegName "," RegName              [left]
58                         |UnOp RegName "," RegName
59                         |"invoke-static" "{"Parameters"}" "," MethodRef
60                         |"invoke-virtual" "{"Parameters"}""," MethodRef
61                         |"move-result" RegName
62                         |"retrun-void"
63                         |"return" RegName
64                         |"monitor-enter" RegName
65                         |"monitor-exit" RegName
66   syntax Sput  ::= "sput" |"sput-object"
67   syntax Sget  ::= "sget" | "sget-object"
68   syntax Binop ::= "add" | "sub" | "mul" | "div" |...
69   syntax Unop  ::= "neg" | "not" | "int-to-long" |...
70   syntax Val  ::= Int
71   syntax Switchtab ::= ".sparse-switch" Tablecases ".end sparse-switch"
72   syntax Tablecases ::= List {Tablecase," "}
73   syntax Tablecase ::= Value "→" ":" Label
74   syntax StringId , Label ::= Id
75   syntax Empty ::= " "
76   syntax ManifestFile::=".manifest" MethodRef
77 end module
```

Listing 1.3: 𝕂 source file for 𝕂-Smali syntax

A *Comment* is a regular expression r"<regExp>" that starts with # and followed by any character (.) zero or many times (*). Notice that the attribute [token] used when defining a comment and the fully qualified name of a class (lines 10 and 33) signals that the associated sort will be occupied by domain values, which is a set of literal values (string and integer). A class definition includes its fields and methods as well. A method is defined by a set of access flags that determines its scope, a full name, a signature, and a body. A method name signature consists of the method input *MethodIntypes* and output *MethodRetTypes* types. Fields are a list of *field* identified by the keyword *.field*, access flags, a name, a type, and a value (if exists). The method body is a list of blank-separated statements. Statements are either directives or instructions. A directive could be *.locals* followed by an integer, indicating the number of the local register in the method. The directive *.registers* specifies the total number of registers in the method (including local and parameter registers). Considered instructions include unconditional and conditional jumps with, respectively, *goto*, *if-eq, if-lt* and *sparse-switch* instructions. All jumping to a given label (*:Label*) identifying the concerned instruction. We also consider instructions of moving a constant string and constant integer to a destination register with, respectively *const-string* and *const* instructions. Exchange between registers is modeled with *move* instruction from source to a destination register. Objects and arrays creation, arithmetic and subroutine instructions as method invocation and return (void and non-void) instructions are also part of the $\mathbb{K}$-Smali language. Notice that the attribute [left] can be used for binary operations like addition which is left-associative (line 57). $\mathbb{K}$-Smali includes as well read/write static fields (*sget, sput*), instance fields (*iget, iput*), and array elements (*aget, aput*) instructions. Finally, threads synchronization for shared objects instructions are modeled by *monitor-enter* and *monitor-exit* followed by the register name *RegName*, which actually holds the object to be reserved reference. For more details, such as interface definition, primitive types notations in Smali, we invite the reader to see [4].

3.2 Configuration

Figure 3 illustrates the configuration of a disassembled DEX file in a high-level overview. A Smali program configuration consists of a top level cell $\top$ holding four main cells: ***Threads***, ***Classes***, ***RegisterMethods***, and ***Heap***. The ***Threads*** cell represents the concurrent behavior of the program. It consists of the executing thread represented by the *Thread* sub-cell and a list of runnable threads in the *Scheduler* sub-cell. All information required for multithreading (synchronization, scheduling and communication), including the currently executing details, are in this sub-cell. Each thread is identified by an identifier *id*, a *RunTime* field computing each executed instruction, and a *status* representing its state. A thread state can be "run" for a running thread, a "runnable" for a thread waiting to be selected by the scheduler, or an object reference "*Ref*" for a blocked thread waiting for the release of this object. ***Classes*** cell is harboring one or multiple class(es). ***RegisterMethods*** cell is an independent cell (since registers are reserved and released each time a method returns). The ***Heap*** cell corresponds to a shared memory used to store the dynamically created objects and arrays.

Figure 4 provides the detailed configuration for sub-cells. A running thread is identified by an identifier *Id*, a *k* cell for the execution context (i.e. the computation to be executed), and a *ReturnResult* cell for its return value. Each class in *Class* cell is defined by its fully qualified class name, its direct super-class fully qualified name, an access flag indicating its visibility, and a map cell *Fieldsclass* mapping the class fields names to values. The *Class* cell includes either a *Methods* cell for all methods (zero or more) in the class. A *method* cell includes its full name, access flags, and a body, which is denoted by *code* cell and consists of a mapping from *Ids* (identifiers) to corresponding

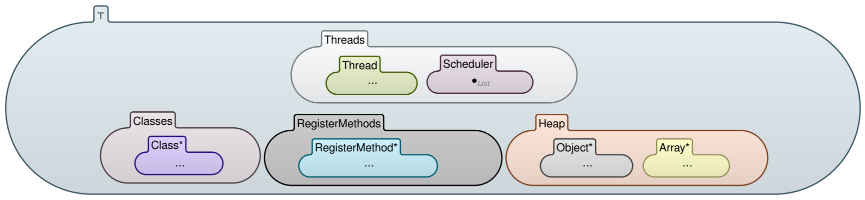


Fig. 3: 𝕂-Smali global configuration

statements. *RegisterMethod* cell holds two sub-cells, the register reference and a mapping register names to values. An *Object* cell records the object reference in the heap, its class full name, a mapping (class) fields to values, and a *Reservedobject* indicator (an integer) cell used for threads synchronization. An "undefined" value indicates a free object (i.e. its associated monitor is not acquired by any thread), whereas a thread identifier *id* value designates a reserved one by the specified thread reference. The *Array* cell records the array name and size, and a mapping form indexes names to values.

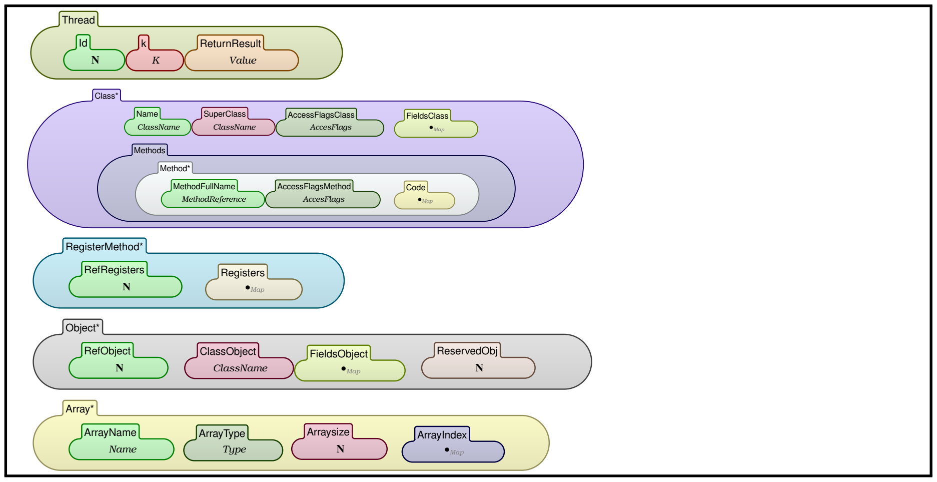


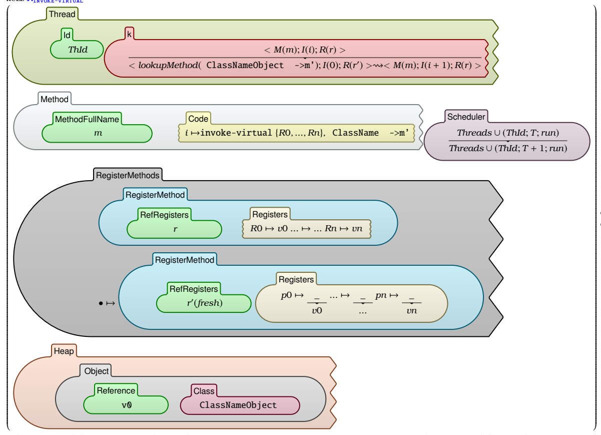
Fig. 4: 𝕂-Smali sub-cells configuration

3.3 Semantics

Hereafter, we present the operational semantics of Smali in 𝕂. It is represented as a set of independent rewrite rules. As our semantics is quite vast (it encompasses more than 50 rules), we will present only rules expressing the most important features. In each rule, we can capture three main repetitive execution phases :(1) the execution of the selected statement in the *code* sub-cell, (2) the selection of next statement to execute in the sub-cell K, (3) the thread executing the current instruction T must be selected by the scheduler. Which means that it must have the state "*run*" and the identifier *id*. This condition is checked by the side condition of each rule. In addition to rewrite rules, $\mathbb{K}$ definitions include functions. Most of these functions are used to manage the side-conditions of rewrite rules, in particular, for logical predicates.

The rule $R_{invoke-virtual}$ invokes an instance method.

RULE $R_{\text{INVOKE-VIRTUAL}}$

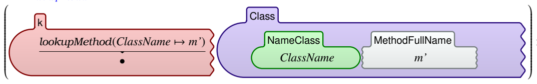


Thread
Id
ThId
k
$< M(m); I(i); R(r) >$
$< lookupMethod($ ClassNameObject ->m'$); I(0); R(r') > \rightsquigarrow < M(m); I(i+1); R(r) >$
Method
MethodFullName
m
Code
$i \mapsto$invoke-virtual $\{R0, ..., Rn\}$, ClassName ->m'
Scheduler
$Threads \cup (ThId; T; run)$
$Threads \cup (ThId; T+1; run)$
RegisterMethods
RegisterMethod
RefRegisters
r
Registers
$R0 \mapsto v0 \ldots \mapsto \ldots Rn \mapsto vn$
$\bullet \mapsto$
RegisterMethod
RefRegisters
$r'(fresh)$
Registers
$p0 \mapsto \frac{\_}{v0} \ldots \mapsto \frac{\_}{\ldots}\ pn \mapsto \frac{\_}{vn}$
Heap
Object
Reference
v0
Class
ClassNameObject

requires
Scheduler($Threads \cup (ThId; T; run)) = ThId$
and
 ClassName ->m'$\neq$Ljava/lang/Thread;->start()V

The caller method passes arguments to the callee by setting its parameter registers. The class of the object whose method is being called (or the receiving object's class) is first retrieved from the heap through its reference. The rule $R_{lookupMethod}$ is called to search the method *m'* in the class *ClassName* and upwards to its super-class chain. .

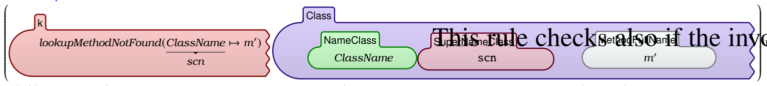
RULE $R_{lookupMethod}$



requires *m'* is in the class *ClassName* .

If the method is not in the class, then lookup in the super-class *scn*.

RULE $R_{lookupMethod}$

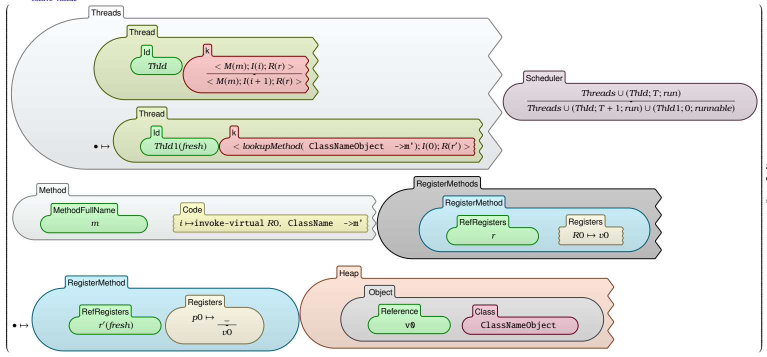


requires *m'* is not the class ClassName .

This rule checks also if the invoked method is different from *start()* method of the class *Thread*, which is used to start a thread and treated separately with the rule $R_{Create-thread}$.

The rule $R_{\texttt{create-thread}}$ creates a new thread object and adds it to the scheduler list.

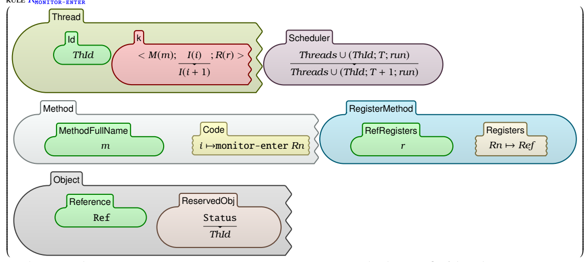
RULE $R_{\text{CREATE-THREAD}}$



requires
Scheduler($Threads \cup (ThId; T; run)) = ThId$
and
 ClassName ->m'
=Ljava/lang/Thread;->start()V

The rule $R_{\texttt{monitor-enter}}$ expresses a successful detention of the monitor associated with the object *Ref* since its status equals to "undefined". The reserved object reference is updated by the owner thread reference *ThId*.

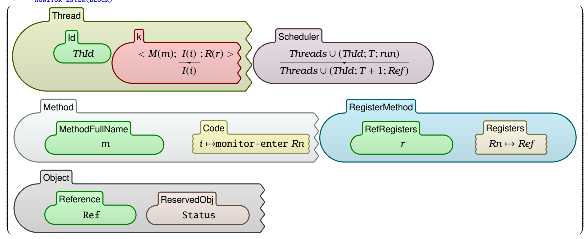
RULE $R_{\text{MONITOR-ENTER}}$



requires $Scheduler(Threads \cup (ThId; T; run)) = ThId$
and $(Status = \text{undefined}\ ) or (Status \neq ThId))$

The rule $R_{\texttt{monitor-enter(block)}}$ models a failed attempt to a shared object (the object status equals to another thread reference). The thread is blocked until the object's monitor is released.

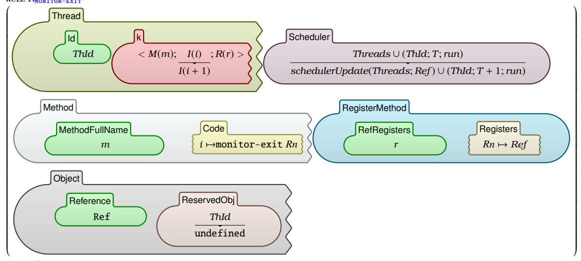
RULE $R_{\text{MONITOR-ENTER(BLOCK)}}$



requires $Scheduler(Threads \cup (ThId; T; run)) = ThId$
and $Status \neq \text{undefined}$
and $Status \neq ThId$

The rule $R_{\texttt{monitor-exit}}$ represents a thread that releases the owned monitor for the object in *Rn* (the status is rewritten by the "undefined" value).

RULE $R_{\text{MONITOR-EXIT}}$



requires $Scheduler(Threads \cup (ThId; T; run)) = ThId$
and $(Status = \text{undefined}\ )\ or\ (Status \neq ThId))$

where:
$schedulerUpdate(\{\}; Ref) = \{\}$
$schedulerUpdate(\{(ThId; T; Ref)\} \cup Threads; Ref) = \{(ThId; T; run)\} \cup schedulerUpdate(Threads; Ref)$
$schedulerUpdate(\{(ThId; T; Status)\} \cup Threads; Ref) = \{(ThId; T; Status)\} \cup schedulerUpdate(Threads; Ref)\ \ If\ Status \neq Ref$

The Function *schedularUpdate()* releases all blocked threads waiting for this object since it is now free.

4 Program verification

In addition to defining an executable formal semantics of Smali, our second objective is to formally verify Smali programs using $\mathbb{K}$ and the built-in tools for parsing and program verification. To verify properties on a given Android application, we need a $\mathbb{K}$-Smali *program P*, a *property S* to be proved, and finally testing if *P* satisfies or not the property *S* using the command *krun* and the appropriate option. For property specification, $\mathbb{K}$ offers a wide range of options. In sum, the logical foundation of the $\mathbb{K}$

framework's verification infrastructure is matching logic for static properties [11] and reachability logic for the dynamic ones (from version 4 and up). Therefore, properties can be specified as reachability logic assertions using $\mathbb{K}$ rewrite rules. They can also be written as preconditions and post-conditions in Hoare triples [2], temporal logic formulas LTL, Modal logic, formulas in first-order logic, or any other logical formalism. $\mathbb{K}$ offers Linear Temporal Logic (LTL) model checking via compilation into Maude programs through its Maude [12, 13] backend available in version 3.5 and down. Security-related properties such as confidentiality, access control, information flow, etc. can be checked in general. For Android, we may need more fine-grained properties specific to its typical features and security-sensitive services. Most existing approaches rely on the analysis of API calls to detect malicious behaviors of a given Android application (e.g. in [14–18]). Executing sensitive operations, such as sending SMS messages, recording audios and videos, tracking the geographical position of the user are all performed through calls to API methods. Verifying properties for each time these APIs are used will certainly increase the false positive rate. Instead, it would be more judicious to express the temporal order in which these APIs are invoked. In this sense, model-checking is the most suitable technique for verifying temporal properties [14]. Many spyware exploit the system services to collect or disclose private data. As a result, they become able to track the geographic location of the device, eavesdrop on conversations, take photos, and record videos without the user's knowledge. The SMS sending APIs are also among the most sensitive APIs in Android. Such a feature can be mislead by attackers to send SMSs to premium-rate numbers without the user's consent. DogWars [19], for instance, is an application containing a Trojan that sends SMSs to all contacts on the device. Similarly, the telephony-related APIs can be a way to use paid services in Android and call premium-rate numbers without notifying the user.

4.1 Spyware

Among the properties that we want to check is if a given Android application tries to spy on the user. Taking photos, recording audios and videos without the user's knowledge are among behaviors that characterize spyware on Android.

- *Program to be verified:* Given a program *P* that allows to take a picture with the instruction *invoke-virtual* of the method *TakePicture* from the fully-qualified class name *Landroid/hardware/Camera;*. The invocation of this API can only be exploited to take a picture without the user's knowledge. Except that, invoking the method *setPreviewDisplay* or *setPreviewTexture* from the same class before allows to display the camera preview. In this way, the user will be aware that the camera is open and tries to take a photo. APIs representing these two features are mainly invoked by as illustrated in the following example.

```
invoke-virtual{v1,v2},Landroid/hardware/Camera;↦setPreviewDisplay(Landroid/
    view/SurfaceHolder;)V
invoke-virtual{v0,v2,v2,v1},Landroid/hardware/Camera;↦takePicture(Landroid/
    hardware/Camera$ShutterCallback;Landroid/hardware/Camera$PictureCallback;
    Landroid/hardware/Camera$PictureCallback;)V
```

Listing 1.4: Taking a picture and displaying camera preview APIs

- *Property specification:* An LTL formula can express the desired behavior and requires the order of having the API *SetPreviewDisplay* or *SetPreviewTexture* **before** the API *TakePicture*. This way we can check if the program can spy on the user or not. To define this property, we need to express the past logic using both past LTL (ptLTL) and future LTL logic (LTL) modalities. Temporal logic (LT) gathers LTL and ptLTL modalities [20]. An LTL formula representing this behavior could be defined as follows:

  $$\Box(\mathit{takePicture} \rightarrow \odot(\mathit{setPreviewDisplay} \vee \mathit{setPreviewTexture}))$$

  The LTL formula starts with the LTL operator $\Box$ which means "*always*". The operator $\odot$ represents the past logic (ptLTL) and means "*previously*". The operator $\vee$ expresses disjunction. Intuitively, the formula states that "*If takepicture happens now, setPreviewDisplay or setPreviewTexture must (always) have happened (previously)*". Similarly, by not using the method *setPreviewDisplay* from *Landroid/-media/MediaRecorder;*, the user will not be warned when the application attempts to record video or audio surreptitiously. The following LTL formula expresses this behavior :

  $$\Box(\mathit{setVideoSource} \vee \mathit{setAudioSource} \rightarrow \odot\mathit{setPreviewDisplay})$$

4.2 SMS Trojan

SMS trojans cause financial losses to users by sending SMS messages to premium-rate numbers without the user's consent. Hiding of received SMS messages is possible by aborting broadcast intents. In fact, after invoking the API *sendTextMessage* with a premium number, the attacker intercepts and calls the *abortBroadcast* function to remove billing-related notification messages from respective service providers. This way, the attacker can make sure that the user will not be able to detect that an SMS has been sent.

- *Property specification:* This property can be expressed by the formula below:

  $$\Box(\neg\mathit{abortBraodcast} \rightarrow \Diamond\mathit{sendTextMessage})$$

  The ptLTL operator $\Diamond$ means "*eventually*" in the past. In order to detect the possibility of an SMS Trojan, the formula ensures that "*each time the abortBroadcast function is preceded with a sendTextmessage* method, this action will not be permitted". Intuitively, it ensures that the user will be notified each time he receives an SMS.
- *Krun command: Krun* command is used to execute a program having the $\mathbb{K}$ semantics of the language. LTL formulas can also be verified through LTL model checking with this command plus the option "−−ltlmc" as follows:

```
krun P.smali --ltlmc LTLformula
```

The option "−−ltlmc" is used with the command *Krun* to indicate that the specified program (*P.smali*) is model-checked with the following LTL formula (*LTLformula*). The outcome is *True* if the property holds. Otherwise, a counter-example representing an execution violating the property is exhibited.

5 Comparison with related work

Android application analysis tools can be grouped into two categories of approaches: test-based and formal semantics-based approaches, hereafter discussed.

Test-based approaches Several efforts have focused on the Android security issue without a formal foundation at both the specification and verification levels. For example, Porter Felt et al. [21] proposed a tool, called Stowaway, to capture overprivilege in compiled Android applications to ascertain whether Android developers follow the least privilege rule. This tool collects the API calls that an application uses and associates them with permissions. They used dedicated testing tools to build the permissions map in order to spot privilege escalation. In [22], Chin et al. proposed a tool, called ComDroid, to analyze the interaction between applications in order to detect vulnerabilities and security risks in their components. In [23], Arzt et al. proposed Flowdroid, a static taint-analysis tool for Android applications. This tool models important aspects specific to Android such as application lifecycle and callbacks, which results in reducing missed leaks and false positives. What these tools all have in common is that they do not all produce formal proof that an application is secure or not, which undermines their reliability and raises questions about their validity. Another line of test-based Android malware detection is using machine learning and deep learning. The overall idea consists of building a dataset holding both malicious and benign Android application samples, from which features are extracted. Based on these inputs, classification algorithms are used for malware detection. Feng et al. [24] propose a pre-installed solution called MobiTive. They divide their system functionality into a server-side and mobile side. The first part provides a trained deep learning model and a feature dictionary built from the extracted API calls and manifest properties. In the second part, as soon as an application is downloaded, MobiTive extracts features of the API calls and manifest properties from the *classes.dex* and *manifest* files. Although the authors insist on the benefits of extracting features directly from the APK, without wasting time on converting it into a human-readable format, they use a third-party decoder library and an API parser for that, which is also time-consuming. In our approach, we use the reverse-engineering tool Apktool to retrieve the Smali code. The tool generates immediately a human-readable code (Smali), from which several features can be more easily extracted and parsed. In the same stream of thought, Kumar et al. [25] use a deep learning model to analyze and detect malware in Android Internet of Things (IoT) devices. Although their claimed high accuracy scores, none of the cited pieces of work are based on a formal specification to detect malware. Therefore, none of them can be proven correct. Moreover, the shown results are tritely bound to the given scenarios.

Formal approaches In addition to test-based approaches, there have been several efforts to use formal methods to analyze the code of Android applications. In [26], Khan et al. put forward a formal model to analyze data flows between Android applications using the theorem prover Coq. For that, a programming language-based security was formalized in mechanical Coq. Applications were modeled as simple terms and the system correctness comes down then to data-flow safety. Coq offers mechanical support for building and checking proof of correctness. In [27,28], Betarte et al. suggested a formal specification of Android's permission model allowing to state and prove security proprieties and enforce permission-based access control policies. Properties were proved using the Coq proof assistant. Compared to Coq, $\mathbb{K}$ supports an interpreter enabling to test and to run testing programs (executable semantics), a symbolic execution engine for the language, and parsers generated automatically from the specification. Moreover, the program definition with $\mathbb{K}$ is clearer and more concise with BNF notations, against inductive purely syntactic definitions with Coq. In sum, the $\mathbb{K}$ framework is better suited for specifying languages and verifying programs. This task is far more expensive when using Coq. This being said, Coq remains more suited to model math-oriented problems and to prove theorems. Other important studies [29–31] have been proposed. They were based on the formal semantics of Dalvik bytecode for the analysis, detection of potential vulnerabilities, or malicious behavior. Despite promising results and the power of formal methods to identify problems at an earlier stage and produce more robust languages, none of these pieces of work have been based on a language definitional framework for defining formal semantics. On the other hand, none of the aforementioned formal languages for Android application is considering the concurrent nature of the language. $\mathbb{K}$-Smali fully supports multithreading. Pegasus [14] model checks temporal logic formulas, expressing an application behavior as expected by the user, against an abstraction called permission event graph (PEG). The PEG is then verified using a verification tool for compliance with the specification. Other model checkers are also proposed in [32,33]. [34] is the closest work to ours. Instead of parsing a single application, the approach is applied to a set of applications (APK) since the checked property is related to collusion between different applications. The Maude model checker checks the property for the input set of Android applications. The resulting semantics was only used to verify the collusion property. In our work, the semantics of one Android application enables us to verify many properties related to its API calls. This feature is a key static metric that enables to identify malicious behaviors, such as SMS Trojans, spyware, and many other malware. In [35], we used the $\mathbb{K}$-*Smali* semantics presented in this paper as a formal basis to enforce security policies. LTL formulas expressing several properties are defined and then transformed to $\mathbb{K}$-*Smali* programs, then injected into untrusted programs, compelling them to abide by the policy. The security policy enforcement process has been automated in [36] using the $\mathbb{K}$ framework once again. In this work, all the enforcement steps were made by $\mathbb{K}$ through a syntax, a configuration and rewrite rules. It generates, from a formula defined in $\mathbb{K}$ and a $\mathbb{K}$-*Smali* program, a new version of the program that behaves according to the introduced formula. We used the interpreter offered by $\mathbb{K}$ to confirm this result.

6 Conclusion

In this work, we have presented $\mathbb{K}$-Smali, which we believe is the most complete formal semantics of the Smali language. Using the $\mathbb{K}$ framework, we have been able to improve several uncovered points in *Smali*$^{+}$, such as the program entry point, the initialization step, and other missing details discovered when compiling the language definition. Execution, semantics debugging are all taken care of by the framework. The interpreter allows executing sample programs and debugging the semantics, which increases the reliability of the generated formal model. $\mathbb{K}$-Smali includes an important feature that has been largely neglected in the state of the art, which is multithreading. This allows testing the behavior of any multi-threaded Android program. Moreover, owing to its built-in tools, $\mathbb{K}$ makes it easy to verify properties on Smali programs.

References

1. Mcafee mobile threat report 2020. `https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf`.
2. Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. Semantics-based program verifiers for all languages. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 74–91. ACM, 2016.
3. Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.
4. Marwa Ziadia, Jaouhar Fattahi, Mohamed Mejri, and Emil Pricop. Smali+: An Operational Semantics for Low-level Code Generated from Reverse Engineering Android Applications. *Information*, 11(3), 2020.
5. Kyungmin Bae and José Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 99:193–234, 2015.
6. Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020.
7. Denis Bogdanas and Grigore Rosu. K-java: A complete semantics of Java. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 445–456. ACM, 2015.
8. Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 567–592. Springer, 2014.
9. Chris Hathhorn, Chucky Ellison, and Grigore Rosu. Defining the undefinedness of C. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 336–345. ACM, 2015.
10. Grigore Rosu. $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 186–206. IOS Press, 2017.

11. Grigore Rosu and Xiaohong Chen. Matching logic: the foundation of the K framework (invited talk). In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, page 1. ACM, 2020.
12. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
13. Traian-Florin Serbanuta and Grigore Rosu. K-maude: A rewriting based tool for semantics of programming languages. In Peter Csaba Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2010.
14. Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.
15. M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-an, and X. Luo. Detecting Vulnerable Android Inter-App Communication in Dynamically Loaded Code. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 550–558, April 2019.
16. Manel Jerbi, Zaineb Chelly Dagdia, Slim Bechikh, and Lamjed Ben Said. On the use of artificial malicious patterns for android malware detection. *Computers and Security*, 92:101743, 2020.
17. Han Gao, Shaoyin Cheng, and Weiming Zhang. Gdroid: Android malware detection and classification with graph convolutional network. *Computers and Security*, 106:102264, 2021.
18. Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. Towards Model Checking Android Applications. *IEEE Trans. Software Eng.*, 44(6):595–612, 2018.
19. Elinor Mills. Dog wars app for android is trojanized. `https://www.cnet.com/news/dog-wars-app-for-android-is-trojanized/`.
20. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.
21. Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David A. Wagner. Android permissions demystified. pages 627–638, 2011.
22. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011.
23. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.
24. Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. A performance-sensitive malware detection system using deep learning on mobile devices. *IEEE Trans. Inf. Forensics Secur.*, 16:1563–1578, 2021.
25. Rajesh Kumar, Wenyong Wang, Jay Kumar, Zakria, Ting Yang, Waqar Ali, and Abubackar Sharif. Iotmalware: Android iot malware detection based on deep neural network and blockchain technology. *CoRR*, abs/2102.13376, 2021.

26. Wilayat Khan, Muhammad Kamran, Aakash Ahmad, Farrukh Aslam Khan, and Abdelouahid Derhab. Formal analysis of language-based android security using theorem proving approach. *IEEE Access*, 7:16550–16560, 2019.
27. Gustavo Betarte, Juan Diego Campo, Carlos Luna, and Agustín Romano. Formal analysis of android's permission-based security model,. *Sci. Ann. Comput. Sci.*, 26(1):27–68, 2016.
28. G. Betarte, J. Campo, M. Cristiá, F. Gorostiaga, C. Luna, and C. Sanz. Towards formal model-based analysis and testing of android's security mechanisms. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–10, 2017.
29. Etienne Payet and Fausto Spoto. An Operational Semantics for Android Activities. pages 121–132, 2014.
30. Erik Wognsen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and René Hansen. Formalisation and analysis of Dalvik bytecode. *Science oßf Computer Programming*, pages 25–55, 2014.
31. Jinseong Jeon and Kristopher K. Micinski. Symdroid : Symbolic Execution for Dalvik. 2012.
32. Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Android Collusion: Detecting Malicious Applications Inter-communication through Shared-Preferences. *Inf.*, 11(6):304, 2020.
33. Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, and Antonella Santone. Colluding android apps detection via model checking. In Leonard Barolli, Flora Amato, Francesco Moscato, Tomoya Enokido, and Makoto Takizawa, editors, *Web, Artificial Intelligence and Network Applications - Proceedings of the Workshops of the 34th International Conference on Advanced Information Networking and Applications, AINA Workshops 2020, Caserta, Italy, 15-17 April*, volume 1150 of *Advances in Intelligent Systems and Computing*, pages 776–786. Springer, 2020.
34. As Irina Mariuca, Jorge Blasco, Thomas M. Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. *Detecting Malicious Collusion Between Mobile Software Applications: The Android TM Case*, pages 55–97. Springer International Publishing, August 2017.
35. Marwa Ziadia, Mohamed Mejri, and Jaouhar Fattahi. Formal and automatic security policy enforcement on android applications by rewriting. In Hamido Fujita and Héctor Pérez-Meana, editors, *New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 20th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques, SoMeT 202, Cancun, Mexico, 21-23 September, 2021*, volume 337 of *Frontiers in Artificial Intelligence and Applications*, pages 85–98. IOS Press, 2021.
36. Marwa Ziadia, Mohamed Mejri, and Jaouhar Fattahi. K Semantics for Security Policy Enforcement on Android Applications with Practical Cases. In EAI CICom 2021, editor, *2nd EAI International Conference on Computational Intelligence and Communications November 18-19, 2021 Versailles, France*. EAI CICom 2021, 2021. Accepted.