information

IMPACT FACTOR 3.1

CITESCORE 5.8

Article

Smali^{+}: An Operational Semantics for Low-Level Code Generated from Reverse Engineering Android Applications

Marwa Ziadia, Jaouhar Fattahi, Mohamed Mejri and Emil Pricop

Special Issue

Advanced Topics in Systems Safety and Security

Edited by
Dr. Emil Pricop, Dr. Grigore Stamatescu and Dr. Jaouhar Fattahi

MDPI

https://doi.org/10.3390/info11030130

information

MDPI

Article

Smali$^+$: An Operational Semantics for Low-Level Code Generated from Reverse Engineering Android Applications

Marwa Ziadia [1,*], Jaouhar Fattahi [1], Mohamed Mejri [1] and Emil Pricop [2]

[1] Department of Computer Science and Software Engineering, Laval University, Pavillon Adrien-Pouliot 1065, avenue de la Médecine, Quebec City, QC G1V 0A6, Canada; marwa.ziadia.1@ulaval.ca (M.Z.); jaouhar.fattahi.1@ulaval.ca (J.F.); mohamed.mejri@ift.ulaval.ca (M.M.)

[2] Automatic Control, Computers and Electronics Department. Petroleum-Gas University of Ploiesti, 100680 Ploiesti, Romania; emil.pricop@upg-ploiesti.ro

* Correspondence: marwa.ziadia.1@ulaval.ca

Received: 31 January 2020; Accepted: 26 February 2020; Published: 27 February 2020

Abstract: Today, Android accounts for more than 80% of the global market share. Such a high rate makes Android applications an important topic that raises serious questions about its security, privacy, misbehavior and correctness. Application code analysis is obviously the most appropriate and natural means to address these issues. However, no analysis could be led with confidence in the absence of a solid formal foundation. In this paper, we propose a full-fledged formal approach to build the operational semantics of a given Android application by reverse-engineering its assembler-type code, called Smali. We call the new formal language Smali$^+$. Its semantics consist of two parts. The first one models a single-threaded program, in which a set of main instructions is presented. The second one presents the semantics of a multi-threaded program which is an important feature in Android that has been glossed over in the-state-of-the-art works. All multi-threading essentials such as scheduling, threads communication and synchronization are considered in these semantics. The resulting semantics, forming Smali$^+$, are intended to provide a formal basis for developing security enforcement, analysis and misbehaving detection techniques for Android applications.

Keywords: Android applications; multi-threading; operational semantics; reverse engineering; Smali$^+$

1. Introduction

A few years ago, mobile phones were used to make calls or send messages. Today, they surpass computers as the most commonly used digital device. They manage our agenda, emails, credit cards, itineraries and business documents. Android is the most popular operating system for mobiles and embedded devices, having the largest application market and 85% of all smartphones sold in 2019 were equipped with an Android OS [1]. Android is an open nature platform, which means that applications could be downloaded from sources other than the official Google play store. This is an important feature that has contributed to its unquestionable success, given the breadth of the available application that draws people to the platform, making it an ideal target for malicious application downloads.

Indeed, users are increasingly exposed to attacks targeting the Android environment via malicious applications. They thus endanger privacy information, by disclosing sensitive data (FakeNetflix malware [2]) or collecting sensitive banking information, especially with the increasing use of banking applications (Anubis trojan [3]). Furthermore, the installation of apparently legitimate malicious applications can lead to: clandestine eavesdropping on telephone conversations; tracking GPS position; exploiting pay services to cause financial losses to the user for the benefit of the attacker by calling or sending SMS messages to premium-rate numbers without the user's knowledge (SMS Trojan such as FakePlayer, AsiaHitGroup and GGTracker [4–6].

To deal with this, automated tools for analyzing, verifying and enforcing the security of Android applications are highly needed [7–10]. Nevertheless, they must be based on a formal specification of the target platform to give solid results. In this paper, we propose formal operational semantics for a subset of the low-level Android code, which we consider particularly relevant for modeling Android applications and which we call Smali$^+$. It includes the main bytecode instructions of Dalvik, and a few important API methods related to Java concurrency. Smali$^+$ is ultimately written from Smali with some essential native methods that were replaced with macro-instructions for simplification. Smali$^+$ is intended to serve as a basis for further analysis of Android applications and security implementation techniques. Android applications are mainly written in Java. The Java source code is first compiled into a Java Virtual Machine (JVM) bytecode using a standard Java compiler called *Javac*. Following this, the Java source files are converted into class files that store Java bytecode. The Java bytecode is then translated to an optimized bytecode called *Dalvik* through a tool called *dx*. At this stage, all the class files are converted and consolidated into a single DEX file called Dalvik EXecutable or simply a DEX to save memory. An Android Package Kit (APK) is essentially a zip of the DEX file accompanied by a Androidmanifest.xml file, a set of resources and potentially shared libraries. Figure 1 illustrates these steps.

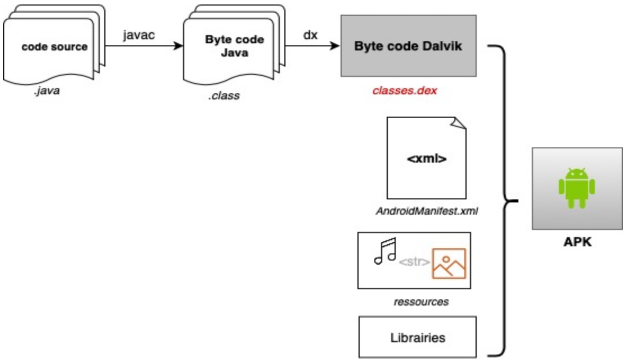

Figure 1. Compilation steps of an Android application.

In this work, we focus on the DEX format file, which contains the Dalvik binary code used even by the successor of Dalvik (since Android 5.0) called Android Runtime (ART).

Formalizing a low-level code, rather than high-level Java source or intermediate level Java bytecode, is our choice for many reasons. Firstly, Dalvik byte code is always available and it is easily obtainable from any Android application. Secondly, Dalvik bytecode is the common executable format for all Android applications and therefore the code is much closer to the code really executed. Even though decompilation from Dalvik back to Java or to Java bytecode is possible using reverse engineering tools (such as dex2jar and ded), there is no guarantee to recover the original source code since there is not a 100% robust and correct Dalvik-to-Java reverse translation tool [11]. However, even though that it is possible to retrieve source code or Java bytecode from Dalvik, editing or improving code at this level requires the user to reconvert it back to Dalvik and running the application afterward will often fail [9]. Focusing directly on Smali will avoid such problems. Hence, binary code obtained at this level, in DEX file, is illegible and requires conversion into a more understandable format prior to being analyzed, improved or edited. Reverse engineering in software makes it possible to convert a machine-readable binary file into a human-readable file, which is the case with DEX files.

Apktool [12] is a reverse engineering tool that simplifies the entire process of assembling and disassembling Android applications. It includes "*Smali*" and "*bakSmali*", which are equivalent to "assembler" and "disassembler", respectively allowing the passage from and to the DEX format. Apktool allows the user to disassemble applications to nearly original form. It uses *BakSmali* to produce, from an APK, a human-readable format akin to assembly languages called Smali (Smali is both the name of a mnemonic language for the Dalvik bytecode and its assembler version.). This code is nothing but a translation of the machine code generated by the DVM. In other words, it is a readable representation of Dalvik bytecode in an assembly-like code, with mnemonic instructions. *BakSmali* creates a Smali file for each class in the application preserving the original signature. The structure of such a file is presented in Figure 2. In addition to the code contained in the classes.dex file, Apktool generates the application decoded resources, as well as the *AndroidManifest.xml* file (in a readable version. These reverse engineering analysis techniques are still effective with the newly introduced ART environment [13].

```
.class modifiers Lsome/package/Someclass;
.super Lsome/package/Someclass;
.implements Lsome/package/Someinterface;
.source "someclass.Java"

.field modifiers fieldname : type;

.method modifiers methodname (type,...)type
    .locals ...
    instruction ...
    instruction ...
    instruction ...
    ...
.end method
...
```

Figure 2. Structure of a Smali file.

In this paper, we put forward formal semantics for Smali. Smali is an assembly-like language that runs on Android's DVM. It is obtained by 'bakSmaling' the Dalvik executable file (.dex). A syntax and semantics have been adopted to specify this low-level code. The resulting formal language is a sub-language of Smali and a simpler language, called Smali^{+}. A set of the most used Dalvik instructions have been generalized into 12 semantically different instructions (see [11] for generalization process), compared to more than 200 Dalvik original instructions in Smali. In addition to this set, our semantics includes instructions related to multi-threading. We plan to use Smali^{+} in the near future to specify security properties for Android applications and this in order to protect the user from security threats that target the Android environment through downloaded applications.

The paper is organized as follows. In Section 2, we present some related work with similar ideas of bytecode formalization and we discuss their advantages as well as their drawbacks and limitations. In Section 3, we give some essential preliminaries related to Smali (registers, some adopted notations, types, etc.). In Section 4, we present the operational syntax and semantics of Smali^{+} for a single-threaded application. In Section 5, we present the operational syntax and semantics of Smali^{+} for a multi-threaded application. In Section 8, we conclude and we introduce the future avenues of our research.

2. Related Work

Mostly, the studies based on formal semantics of Android target a single well-defined goal. This can be an analysis for certification, a detection of potential vulnerabilities or malicious behavior of an application, or a verification of any aspect. It can also be a means to reveal security breaches of Android applications [14]. We will see in the studies we are presenting hereafter that formalization elements substantially differ from one objective to another. This being said, it is practically impossible to evaluate the efficiency of analyzes that are not based on the formal specification of the targeted platform.

In [15], Payet et al. define operational semantics for a subset of Dalvik opcodes that present registers manipulation, arithmetic operations, object creation, access and method calls as well as

Android activities. Semantics rules were relatively complex. An Android program was modeled as a graph of blocks where each block has one or more instructions among the selected instructions. Blocks are linked in a way that they express control flow passing from one block to another. They require that invoke and return instructions only occur at the beginning and the end of a block, respectively. Blocks of semantics integrate instruction semantics for those that are different from a call or a return. Call instruction semantics allow passing from the caller method block to the callee method block. Activity semantics depend on the activity state, method callback, activity life cycle and external events. These semantics are defined to be the basis of static analyses that take into account the life-cycle of the activities. Despite the importance of thread-activity connection in Android semantics, threading was detached from activities semantics and concurrency was ignored in this work.

In [16,17], the authors propose a formal operational semantics for the Dalvik bytecode. The formalization was accompanied by a control flow analysis to detect potential malicious actions. Although the results highlight threading as the most often used language features with a (90.18%), this feature was omitted in both analyses and semantics to focus, instead, on reflection, exceptions and dynamic dispatch with 73.00% and 19.53%, respectively, which we find somewhat awkward. This motivates us to pay a special attention to the mutli-threading aspect modeling for Android.

In [11], the authors present SymDroid as a Dalvik bytecode interpreter for eventual security vulnerabilities detection. It is a symbolic execution for a simplified intermediate language of a fraction of Dalvik opcodes, named *µ-Dalvik*. SymDroid receives the Dalvik bytecode (the .dex file) as input. The opcode is first translated to *µ-Dalvik*, which one is based on 16 instructions considered as the most relevant ones to perform code analysis. Then, it is processed by a symbolic execution core using the SMT solver to generate traces as an intermediate result. Finally, the post-analyzer inspects the output traces and determines the final result. Entry points and all possible events affecting the application's behavior were developed according to a client-oriented specification (it is up to the user to model it) to drive the application under test as desired. Although this work's models, in addition to modeling bytecode instructions, the system libraries including Bundle and Intent, Android components life cycle, services and views; it ignores the system's concurrent nature, either in the selected bytecode instructions or at the program symbolic execution level, which is considered as being sequential.

In the same vein, Julia presented in [18] is a static analyzer for Java bytecode based on abstract interpretation. It was extended in [19] and adapted to analyze Dalvik bytecode and handle specific features of Android such as event-driven nature, potentially concurrent entry points and dynamic inflation of graphical views. It applies several static analyses for Android applications' classcast, nullness, dead code and termination analysis, but does not track information flow. Multi-threaded applications were not included in this work and event handlers are executed by a single thread.

Gunadi et al. [20,21] propose an operational semantics of DEX bytecode for certifying non-interference properties through type system. This study includes a translating process from Java bytecode semantics developed in [22] to Dalvik bytecode, concluding that if the first type system guarantees non-interference then its translation into Dalvik bytecode is also typable. Therefore existing bytecode verifiers for Java could certify non-interference properties of Dalvik bytecode.

Multi-threading programming semantics in applications have lately drawn increasing attention. Some combine it with event handling [23–25], others consider the main API methods relating to it [26]. In [24], Kanade proposes a semantic of a combined concurrency model of threads and events. All the focus in this work goes to the event-driven nature of Android and its relationship with the application's threads. As a consequence, all other states that semantics could reach, such as those resulting from basic instruction execution (method call, jump, return instruction, etc.), have not been treated. The semantics proposed in [26] were the closest to ours. They cover the main important Dalvik instructions and handle multi-threading. This paper could be seen an extension of [27], with the obviously major change of the semantics needed for the concurrent setting and exception handling. However, thread scheduling was not discussed and thread spawning is left to the virtual machine to execute in an unpredictable point in time.

In the same stream of thought, in [28], Chaudhuri presents a formal security study on Android using operational semantics and a system of types for specific Android constructs. However, semantics ignore all Java constructs that may appear in Android applications (no class and method modeling), to focus instead on Android components, intents and all Android-specific features related to it (binding a service, sending an intent, etc.). This can be seen as a unified formal understanding of security for users and developers of Android applications to deal with their security concerns.

Some works have focused on other issues of Android such as multi-tasking. For instance, ASM presented in [29] is a formal model that formalizes all Android elements related to multi-tasking, such as activities, back stacks and tasks. An Android application is somewhat seen as a collection of activities with different types that interact with the user through a back stack. ASM has recently been extended in [30] to capture all the core elements of the multi-tasking mechanism used in inter-component communication.

Over time, formalization has included the permissions system as well [31–33]. For example, Bagheri et al. propose in [31] a formal specification for Android application's permission system through an ad-hoc specification language called Alloy. It aims to formally specify the behavior of Android applications, in particular, the mutual interaction between applications based on permissions and security consequences caused by it or what authors call inter-app permission leakage vulnerabilities. Almost all Android elements related to inter-app permissions were taken into account in the formalism. Every application is modeled as a set of components, permissions, intent filters and vulnerable paths. Similarly, in [33], a formal model of the Android's permissions is specified in the theorem prover Coq syntax.

Acteve++ [34] is an automated testing tool for Android Apps. It is based on Acteve [35] but is improved to support input events and broadcast events in order to achieve higher coverage. Authors use a non-standard operational semantics that describes the concolic execution of the program. Semantics describe program execution in response to a sequence of events generated automatically from an external environment. All other features and instructions that Android handles were neglected to focus instead on the event-driven paradigm, which we found not expressive enough to model an Android application. Our operational semantics consider, besides the concurrent feature, a variety of instructions that models methods invocations, object creation and the whole tree structure of an application (class, method and fields).

In [36], the authors focused on the low-level interactions with the operating system, by recording the system calls (syscalls) invoked. To benefit from two levels, the analysis uses generic low-level syscall traces to reconstruct the high-level semantics. While syscalls analysis offers more security guarantees, it, in our opinion, complicates the task more. Especially, this information is extracted from internal interfaces between the Android libraries and the kernel, which may change in the next versions of Android without notice. In our work, we propose a rich semantics that covers all API calls at a high level and we consider that it is sufficient to enforce security policies later.

Some studies like those conducted by Stowway [8] and Comdroid [37] for flow analysis directly analyze the disassembled DEX file for a given application to identify potential component and/or communication vulnerabilities. Despite the promising results of both tools in analyzing Dalvik bytecode and Android's API, proving its soundness and evaluating its efficiency or deficiency is practically impossible in the absence of formal specification and proof.

Concurrent programming concepts and techniques are widely used in Android in order to manage different tasks and threads. Our formalism Smali$^+$ consider this important feature that was neglected before given its complexity. Overall, none of the aforementioned studies, including those considering multi-threading, offer complete semantics covering all the states that a thread can reach nor representing all multi-threading essentials. Most of the studies formalizing Dalvik byte code and handling multi-threading include only the two Dalvik instructions related to monitor use, *monitor-enter* and *monitor-exit*, since Dalvik opcodes encompass only these two instructions with regard to threading. However, a semantic for an Android program should not be limited to these instructions and must also consider instructions related to threads communication, signaling and scheduling. In this paper, we fill this gap by proposing semantics that incorporate, in addition to Dalvik instructions, a wide range of API methods covering multi-threading essentials formulated in macro instructions for the sake of simplicity. In comparison with all test-based approaches, Smali$^+$ is based on formal methods with their foundation in mathematical logic, allowing us to achieve rigorous and unambiguous reasoning in the system specification and proofs, ensuring the system proprieties, while test-based approaches can only ensure that systems satisfy the requirements for test cases. In sum, the proposed formal language is expressive enough to enforce security proprieties and to detect security critical APIs (i.e., those related to sensitive data access such as camera, SMS, telephony and contact list). Its syntax includes the class fully qualified name for each invoked method facilitating to localize such APIs.

3. Preliminaries

In this section, we present the most essential information for Smali. First, we present the DVM architecture and how it affects Smali syntax. Then, we present method invocation and how it affects Smali registers. Finally, we present Smali special notations for types.

3.1. Registers

Being optimized to run on devices on which resources and processor speed are scarce and the DVM architecture is register-based. Local variables are assigned to any of the 2^{12} available registers. A register is used to hold any data value, except for *double* and *long* values where each one requires two registers (64 bits). The Dalvik opcodes operate on the register's content instead of operating directly on values and accesses elements on a program stack such as stack-based virtual machines. Hence, registers allow the DVM to keep track of program evolution while it executes bytecode [38]. Each method in Smali has its own set of registers for each method's arguments, local variables and a special register for its return value. We will see later that most of the instructions include source and destination registers. Smali language denotes each set of registers differently, which allows us to visually distinguish between the method's local and argument registers.

The alternate *.locals* directive specifies the number of local registers used by the method (non-parameter registers) which is statically known. Local registers in Smali are denoted with $v_0, v_1, v_2, ..., v_n$, where v_0 is the first local register, v_1 the second and so on until the last register. This includes a special register for a method return value that allows passing return values from the callee back to the caller, which one is denoted by *ret*.

$$LocalRegisters = \mathbb{N} \cup \{ret\}$$

Parameter registers in Smali are denoted by $p_0, p_1, p_2, ..., p_n$. The first parameter for non-static methods is always the object that the method is being invoked on, in this case p_0 holds the object reference and p_1 the second parameter register. For a static method invocation p_1 is the first parameter register. For more details, please see the Method invocation subsection.

The *.registers* directive specifies the total number of registers in the method. This includes the registers needed to hold the method parameters, which are stored in the last registers in the method.

$$Registers = LocalRegisters \cup ParameterRegisters$$

3.2. Method Invocation

The DVM conforms to the ARM's calling convention which is used for low-level code where parameters, return values, return addresses and scope links are placed in registers. It dictates how these elements are shared between the caller and the callee. In fact, these two share a part of their register array so that the caller passes arguments to the callee by setting its parameter registers in the right order. As for class methods, a lookup procedure starts by searching in the list of all static methods that belong to the named class, where classes have distinct names and locating the invoked method through its signature (i.e., name, argument types and number, and return type). Then, its parameter registers array is set according to ARM's calling convention, so that the first argument leads to the first parameter register p_1 and so on until the last argument which identifies the last register for arguments (n arguments lead to n parameter registers).

In the dynamic invocation case, the class of the object whose method is being called (or recipient object's class) is statically unknown, so it is first retrieved from the heap through its reference (see the semantics section for more details). Then, a lookup procedure searches among the class method list upwards to its super-class chain, for a method matching the given method signature. Registers comprise an additional register for the object reference called p_0 in Smali code. Hence, the actual number of parameter registers is $p + 1$.

Local register contents are initially undefined (registers are untyped in Dalvik), however, its number is statically known.

3.3. Types in Smali

Smali code has two major classes of types, primitive types and reference types.

A primitive notation in Smali is particular where a single letter specifies each type, for example V is used for a void type.

Reference types are objects (i.e., class type) and arrays. A class type takes the form *Lpackagename/ClassName;* where the leading L indicates that it is a class type, *packagename* is the package name path where class *ClassName* belongs to, whereas *ClassName* refers to the class name. For example, a thread object in Smali has the following type: *LJava/lang/Thread;* which is equivalent to *Java.lang.Thread* in Java. Arrays take the form $[$ *Type* (*Type* which could obviously be a primitive or a reference). Arrays with multiple dimensions are presented by corresponding number of "[" characters. For example, a two-dimension arrays of int(s) is presented as follow $[[I$ which is equivalent to $int[][]$ in Java. Table 1 summarizes different types in Smali.

Table 1. Types in Smali.

Primitive Types	
B	byte
C	char
F	float
I	int
J	long
S	short
V	void
Z	boolean
Reference Types	
Lpackagename/Classname;	Object
[... Object or Primitives	Array

4. Operational Semantics for a Single-Threaded Application

4.1. Notations

Throughout the paper, we use the following notations:

- $A :: B :: C$ to designate a stack, where A is the top-most value of the stack, B is the underlying element and C is the remaining portion of the stack. An empty stack is presented by ϵ.
- $\perp$ to denote any undefined value.
- $dom(f)$ is domain of a function f. The notation $dom[f \mapsto x]$ expresses the domain dom where the value of a function$f$ is updated to x.
- $f[x \mapsto y]$ expresses the function f where value x maps to y so $f(x) = y$.

4.2. Syntax

Table 2 provides basic syntactic categories as well as the selected instructions syntax.

A package of a disassembled DEX bytecode format is specified by a name *pck* and sequences of classes. In our formal model, we consider that a package consists only of classes that correspond to *.Smali* files (Androidmanifest file and the rest of XML files are not considered in our formalization).

Table 2. Smali$^+$: sequential execution.

(Package)	$Pckg$	::= **.package** pck $\{Cl^*\}$	
(Class definition)	Cl	::= **.class** $(Acc\text{-}flg^*)$ Cfn **.super** Sc **.implements** $Intf^*$ $\{Fld^*, Mtd^*\}$	
(Super class)	Sc	::= $Cfn \mid \top$	
(Interface definition)	$Intf$	::= **.interface** $(Acc\text{-}flg^*)$ Inf **.super** $Sinf^*$ $\{CstFld^*, MtdSign^*\}$	
(Super interface)	$Sinf$	::= Inf	
(Field definition)	Fld	::= **.field** $(Acc\text{-}flg)^*$ $f : \tau$	
(Constant Field definition)	$CstFld$	::= **.field** **public** **final** **static** $f : \tau$	
(Method definition)	Mtd	::= **.method** $(Acc\text{-}flg)^*$ $MtdSign$ **.locals** loc $\{LabelInst^*\}$	
(Method signature)	$MtdSign$	::= $m\ (\tau_1, ...\tau_n)\ ret\tau$	
(Access flags)	$Acc\text{-}flg$	::= **public** | **private** | **protected** | **final** | **...**	
(Labeled Instruction)	$LabelInst$	::= $i\ Inst$	
(Label)	i	::= **int**	
(Instructions)	$Inst$	::= **goto** i	(unconditional jump)
		| **move** $Des\ Src$	(move from source to destination)
		| **binop**$_{\oplus}$ $v\ v_1\ v_2$	(binary operation)
		| **unop**$_{\odot}$ $v\ v_1$	(unary operation)
		| **if**$_{\olessthan}$ $v_1\ v_2\ \ i$	(conditional jump)
		| **new-instance** $v\ Cfn$	(object creation)
		| **invoke-static** $Cfn\ MtdSig\ v^*$	(static method invocation)
		| **invoke-instance** $v_{ref}\ MtdSig\ v^*$	(instance method invocation)
		| **return** v	(retrun from non-void method)
		| **return-void**	(retrun from a void method)
(Destination register)	Des	::= v	(register name)
		| $v_{ref}.f$	(instance field)
		| $Cfn.f$	(static field)
(Source register)	Src	::= $Des \mid Cst$	(des or constant)
(Operators)	$\oplus$	::= **+** | **-** |**...**	(binary operator)
	$\odot$	::= ¬ |++|...	(unary operator)
	$\olessthan$	::= **<**|**>**|...	(comparison operator)
(Program counter)	i	::= **int**	
(Num. of loc. registers)	loc	::= **int**	
(Local registers name)	v	::= **string**	
(Parameter registers name)	p	::= **string**	
(Constant)	Cst	::= $Single$	(constant)
(Type)	τ	::= $Prim \mid Ref$	
	$Prim$	::= $Single \mid Double$	
	Ref	::= $Cfn \mid ArrayType$	
	$ArrayType$	::= $ArrayTSingle \mid ArrayTDouble$	
	$ArrayTSingle$	::= **array** $(Single \mid Ref)$	
	$ArrayTDouble$	::= **array** $Double$	
	$Single$	::= **boolean** | **char** | **byte** | **short** | **int** | **float**	
	$Double$	::= **long** | **double**	
(ReturnType)	$ret\tau$	::= τ | **V**	
(Names)	Cfn	::= **L**$packagename/c$	(class full name)
	Inf	::= **L**$packagename/itf$	(interface full name)
	pck, c, itf, f, m	::= **String**	(package, class, interface, field and method names)

A class Cl definition includes its access flags *Acc-flg*, which is a keyword defining the class visibility, a fully qualified class name Cfn that indicates the class package path name followed by the class name c (we assume an unlimited supply of distinct names). This includes also its direct super-class fully qualified name (a single inheritance). $\top$ is applied to classes without super-classes such as the Object class and the Thread class, and finally a set of implemented interfaces $Intf$, fields Fld and methods Mtd.

An interface is specified by its fully qualified name Inf, access flags *Acc-flg*, a set of super-interfaces $Sinf$, its abstract methods (which consist of their method signatures) and constant fields. A field definition comprised its name f, its access flags and a type τ (which could be a primitive for static fields or a class type for instance fields). A method definition includes a set of access flags that determines its scope, the method signature, the number of local registers it operates on denoted by loc and a sequence of labeled instructions $Inst$ that present the method body. A method signature consists of the method name m, argument(s) type τ and a return type $ret\tau$ which might be a void, primitive or a class type. In Smali$^+$, we consider a subset of Dalvik instructions being selected based on results of a study of 1700 Android applications, carried out to determine what instructions and language features are most often used in typical applications [16,17]. In fact, Dalvik bytecode comprises 218 instructions [39]. We bring some modification to the selected instructions that does not affect the expressive power of Dalvik language. In contrast, it simplifies the representation of our semantics. For example, in Dalvik we find 13 variants of the *move* instruction that are semantically similar, we model this group of instructions by only one *move* instruction.

In our formal model, we consider instructions expressing the unconditional and conditional jump with, respectively, *goto* and $if_{\otimes}$ instructions. A *move* instruction to move values from source Src to destination Des. A destination may be a register name v, an instance field $v_{ref}.f$ or a static field $Cfn.f$, whereas a source Src may be any of these elements beside constants cst. We consider also instructions expressing the creation of a new object of a class Cfn, a return from a void and non-void method with *new-instance*, *return-void* and *return* instructions, respectively. Method invocation refers to the method name, argument types and number, return type and registers. For methods class that are dynamically dispatched, it includes in addition to that a register holding the recipient object reference.

4.3. Semantics

Table 3 defines the domains used by our operational semantics. In fact, each application has at least one thread that defines the code path of execution and all of the code will be processed along the same code path if there is no other created thread. Hereafter, we suppose a single-threaded execution, a simple programming model with deterministic execution order, which means that an instruction has to wait for all preceding instructions to finish prior to being processed. We model such execution with a local configuration denoted by σ. It models the full state of a single-threaded program. It includes a call stack C_s, a heap H and a static heap S. A call stack allows keeping track of all information concerning methods invoked in the program. It is initially empty and presented as a sequence of method frames. A method frame F_m is a triplet consisting of a method name m, a program counter i for execution progress, both determine the program point in the invoked method and finally a register array R mapping register names (parameters, locals and return) to values. We adopt the same notations for registers used in Smali, as explained in the Registers subsection. Therefore, we have a set of registers for the method parameters and a set for the method local variables. Local registers content are initially undefined denoted by $\perp$. The top of the call stack represents the currently executing method's frame. Values can be either primitives or heap locations. A heap H map locations (we suppose an arbitrary number of unique locations) to objects Obj or arrays Arr. Objects record their class and a mapping from (class) fields to values, whereas arrays record the array type and its values. Finally, the static heap S is a mapping from static (class) field names to their values. Fields are annotated with their type used for initialization, to determine the default values of each primitive type (see Table 4). This annotation is omitted when it is unneeded.

The relation $\sigma \xrightarrow{m(i)} \sigma'$ models evolution of a starting configuration σ into a new σ' as the result of a computation step. $m(i)$ represents the program point, which corresponds to the instruction at a position i in a specified method m, always for the top-most method frame of the call stack in σ.

To illustrate the semantics, we present in Table 5 the semantic rules for instructions presented in Table 2.

Table 3. Semantic domains.

(Local configuration)	σ	$::= \ < C_s, H, S >$
(Call stack)	C_s	$::= \ \epsilon \mid F_m \mid C_s :: C_s$
(Method frame)	F_m	$::= \ < m, i, R >$
(Registers array)	R	$::= \ (Rg \to Val)^*$
(Registers names)	Rg	$::= \ v^* \vee p^* \vee ret$
(Heap)	H	$::= \ \epsilon \mid \ (l \to (Obj \mid Arr))^*$
(Object)	Obj	$::= \ \{\mid Cfn; (f_\tau \to Val)^* \mid\}$
(Array)	Arr	$::= \ ArrayType\,[^*\ Val$
(Static Heap)	S	$::= \ \epsilon \mid (Cfn.f_\tau \to Val)^*$
(Values)	Val	$::= \ \tau \mid l \mid \perp$
(Local register)	v	$::=$ **string**
(Parameter register)	p	$::=$ **string**
(Return register)	ret	$::=$ **string**
(location)	l	$::=$ **heap locations** $\mid$ **null**

Table 4. Default values of primitive types.

int	**0**
long	0|
short	0
char	’$\backslash u$0000’
byte	(byte) 0
float	0.0f
double	0.0d
object	null
boolean(int)	false (0)

Table 5. Single-threaded semantics.

R_{goto}	$\dfrac{m(i) = \textbf{goto}\ i'}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i',R>::C_s\ ,H,S>}$	$R_{mv\text{-}reg}$	$\dfrac{\begin{array}{c} m(i) = \textbf{move}\ v\ Src \\ [\![Src]\!]=\text{Val} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R[v\mapsto Val]>::C_s,H,S>}$
$R_{mv\text{-}sttf}$	$\dfrac{\begin{array}{c} m(i) = \textbf{move}\ Cfn.f\ \ v \\ R(v) = Val\ \ [\![Cfn.f\ ]\!]=S(Cfn.f) \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,H,S[Cfn.f\mapsto Val]>}$	$R_{mv\text{-}instf}$	$\dfrac{\begin{array}{c} m(i) = \textbf{move}\ v_{ref}.f'\ \ v \\ R(v) = Val\ \ R(v_{ref}) = l\ \ H(l) = o \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,H[l\mapsto o[f'\mapsto Val]],S>}$
$R_{mv\text{-}cst}$	$\dfrac{\begin{array}{c} m(i) = \textbf{move}\ v\ Cst \\ [\![Cst]\!]= \text{Cst} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R[v\mapsto cst]>::C_s,H,S>}$	$R_{new\text{-}ins}$	$\dfrac{\begin{array}{c} m(i) =\textbf{new-instance}\ v\ \ Cfn \\ o' = \{\vert Cfn; (f_\tau \mapsto 0_\tau)^* \vert\}\ \ l' \notin dom(H) \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R[v\mapsto l']>::C_s,H[l'\mapsto o'],S>}$
$R_{b\text{-}op}$	$\dfrac{\begin{array}{c} m(i) = \textbf{binop}_{\oplus}\ \ v\ v_1\ v_2 \\ (R(v_1) \oplus R(v_2)) = Val \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R[v\mapsto Val]>::C_s,H,S>}$	$R_{u\text{-}op}$	$\dfrac{\begin{array}{c} m(i) = \textbf{unop}_{\odot}\ v\ \ v_1 \\ \odot(R(v_1)) = Val \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R[v\mapsto Val]>::C_s,H,S>}$
$R_{if\text{-}true}$	$\dfrac{\begin{array}{c} m(i) =\textbf{if}_{\olessthan}\ v_1\ v_2\ \ \ i' \\ [\![\text{R}(v_1)\ \olessthan\ \text{R}(v_2)]\!] = \textbf{true} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i',R>::C_s,H,S>}$	$R_{if\text{-}false}$	$\dfrac{\begin{array}{c} m(i) = \textbf{if}_{\olessthan}\ v_1\ v_2\ \ i' \\ [\![\text{R}(v_1)\ \olessthan\ \text{R}(v_2)]\!] = \textbf{false} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,H,S>}$
$R_{inv\text{-}st}$	$\dfrac{\begin{array}{c} m(i) =\textbf{invoke-static}\ Cfn\ \ m'(\tau_1,...,\tau_n)ret\tau\ \ v_1,...,v_n \\ lookup(m'(\tau_1,..,.\tau_n)ret\tau, Cfn) = m'(\tau_1,...,\tau_n)ret\tau\ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, p_1 \mapsto R(v_0),...,p_n \mapsto R(v_n)\} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m',0,R'>::<m,\mathbf{i}+\mathbf{1},R>::C_s,H,S>}$	$R_{inv\text{-}inst}$	$\dfrac{\begin{array}{c} m(i) =\textbf{invoke-instance}\ v_{ref}\ \ m'(\tau_1,...,\tau_n)ret\tau\ \ v_1,...,v_n \\ R(v_{ref}) = l\ \ \ H(l) = \{\vert Cfn; (f_\tau \mapsto Val)^* \vert\} \\ lookup(m'(\tau_1,...,\tau_n)ret\tau, Cfn) = m'(\tau_1,...,\tau_n)ret\tau\ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, p_0 \mapsto \mathbf{l}, p_1 \mapsto R(v_0),...,p_n \mapsto R(v_n)\} \end{array}}{<<m,i,R>::C_s,H,S> \xrightarrow{m(i)} <<m',0,R'>::<m,i+1,R>::C_s,H,S>}$
$R_{ret\text{-}nv}$	$\dfrac{m(i) = \textbf{return}\ v}{<<m,i,R>::<m',i',R'>::C_s,H,S> \xrightarrow{m(i)} <<m',i',R'[ret\mapsto R(v)]>::C_s,H,S>}$	$R_{ret\text{-}v}$	$\dfrac{m(i) =\textbf{return-void}}{<<m,i,R>::<m',i',R'>::C_s,H,S> \xrightarrow{m(i)} <<m',i',R'[ret\mapsto R(ret)]>::C_s,H,S>}$

These rules are as follows. The rule R_{goto} updates the program counter to the specified one unconditionally. Rules related to a *move* instruction from source to destination use an evaluation function $[\![\text{-}]\!]$ that evaluates a destination or a source under the current configuration σ, except for registers. In this case, for the sake of being simple, we use directly $R(v)$ always from the top-most method frame of the call stack in σ since $[\![v]\!]$ is equivalent to $R(v)$. Constants are evaluated to themselves whereas static and instance fields are evaluated based on static S and dynamic H heaps, respectively, obviously under the current configuration σ. The rule $R_{mv\text{-}reg}$ evaluates the source sub-expression and then updates the destination register content in the register array. Rules $R_{mv\text{-}insf}$ and $R_{mv\text{-}Sf}$ update instance and static field, respectively, by the content of the source register. Rule $R_{mv\text{-}cst}$ is quite straightforward. That is, after evaluating the source to constant, it updates the destination register content by the constant value.

Rule $R_{new\text{-}ins}$ creates a new object in the heap by reserving a memory with a new fresh location l, loading the class that is instantiated from and initialing its static fields, each by its default value according to Table 4. Once created, it returns the newly allocated object by pushing its heap location in a destination register v.

Rules $R_{b\text{-}op}$ and $R_{u\text{-}op}$ compute a binary or unary expression, respectively, and store the results in the destination register. Rules $R_{if\text{-}true}$ and $R_{if\text{-}false}$ models conditional jump. If the guard is evaluated to true, it branches to the targeted program counter ($R_{if\text{-}true}$), otherwise the program counter is advanced to the next instruction ($R_{if\text{-}false}$). In rules $R_{inv\text{-}st}$ and $R_{inv\text{-}dy}$, a lookup function is called to look up for the appropriate method. In the dynamic case, the method class is retrieved from the heap through object location l which is passed to the register v_{ref}. In both rules, a new method frame structure is pushed on the top of the call stack. It includes the method name, a count program set to 0 and a register array R' set as explained in the subsection Method invocation. Notice that here we increment the program counter of the caller by one to restart from the correct instruction once the callee returns.

A lookup method searches for a method matching the given method signature ($m(\tau_1, ...\tau_n) \xrightarrow{loc} \tau$) in the given class full name and upwards to its super-class chain. Once located, it returns the method signature with the number of its local registers. We assume that the identified class and method exist in the package and class ancestry, respectively, with an array of local registers. Moreover, we admit that all verification checks are performed by the DVM. For instance it is verified that the method can be legally accessed by the class. Thus, the invoke instructions $R_{inv\text{-}st}$ and $R_{inv\text{-}dy}$ are safe to execute.

$$lookup(MtdSign, Cfn) = \begin{cases} m(\tau_1, ...\tau_n) ret\tau \;\; loc & if \;\; m \in Cfn \\ lookup(MtdSign, Cfn.Sc) & else \end{cases}$$

Rules $R_{ret\text{-}nv}$ and $R_{ret\text{-}v}$ pop the top frame from the call stack and pass on the return value from the callee back to the caller through its return register *ret*. Notice that, in the case of a void method, the return value must be moved to *ret* by the callee before the *return-void* instruction.

5. Operational Semantics for a Multi-Threaded Program

Results shown in [17] have highlighted multi-threading as a widely used feature in Android applications with 90.18% including a reference to Java/lang/Thread and 88% using monitors. An important rate that motivates us to take this feature into account in our formalization in order to develop a complete semantic.

5.1. Syntax

Here, we consider multi-threaded programs. Multi-threading semantics include single-threaded semantics for each running thread separately. Threads in the same DVM interact and synchronize using shared objects and monitors associated with these objects. In order to give a full account of Java concurrency, we consider instructions related to this aspect. We define macro-instructions that cover methods of the Java Thread API [40] which are *start* for thread spawning and *join* for joining a referenced thread. We also define macro-instructions that cover several methods of the Java Object API [41] related to thread signaling such as *notify*, *notifyAll* and to synchronization such as *wait*. We also give the semantics of Dalvik instructions related to threads synchronization and monitors with the instructions *monitor-enter* and *monitor-exit*. All instructions syntax are illustrated in Table 6.

Table 6. Smali$^+$: concurrent instructions.

$Inst$	::=	**start** v_{ref}	(start the thread in v_{ref})
	|	**monitor-enter** v_{ref}	(acquire the monitor for object in v)
	|	**monitor-exit** v_{ref}	(release the monitor for object in v)
	|	**join** v_{ref}	(join the thread in v_{ref})
	|	**wait** v_{ref}	(release object's monitor in v_{ref} and suspend current thread)
	|	**notify** v_{ref}	(notify one thread from those waiting on object's monitor in v_{ref})
	|	**notifyAll** v_{ref}	(notify all threads waiting on object's monitor in v_{ref})

5.2. Semantics

An overall configuration $\Sigma =< C_s, S_{rbl}, H, S >$ models the full state of an Android application in its low-level implementation. It presents a multi-threading program configuration including as first attribute a running thread's call stack C_s, a set of runnable threads S_{rbl}, a heap H and a static heap S.

- Each thread in the program has a call stack C_s for methods being invoked, their arguments and local variables, with the same syntax used in Table 3.
- S_{rbl} is a set of pending threads. Each thread is presented by its call stack for method invoked information, plus a special register p_0 holding the thread reference. Threads in this set are in a "runnable" state (i.e., waiting to be selected by the scheduler).
- H and S are dynamic and static heaps which are shared between all threads in the program and have the same semantics domain used for the single-threaded program in Table 3.

A new semantic domain for multi-threaded program is provided in Table 7. Some changes are applied to the object definition. It includes a new fields *acq* which indicates if the object's monitor is acquired by another thread. If this is the case, *acq* will contain this thread's reference, otherwise it will contain an undefined value $\perp$ since an object cannot be reserved by more than one thread at once, at a given time. S_{blck} is a set of blocked threads waiting for the object's monitor to be released. S_{wait} is a set of threads pending notification (threads that executed the *wait* instruction). The initial state of a new instance object, in a multi-threading context, will be initialized as seen in the single-threaded environment (with default values). New attributes are initialized as follows:

- *acq* $\mapsto \perp$ initialized to an undefined value, which means that initially the object is in a free state and could be acquired by a given thread.
- $S_{blck} \mapsto \emptyset$, an empty set of blocked threads, which means that initially there is no thread waiting for the monitor to be released.
- $S_{wait} \mapsto \emptyset$, an empty set of waiting to be notified threads.

A class *Cl* is a Thread class if and only if it is an instance of a Thread class ($\perp$=Thread), which means that its super class *Sc* is either the Thread top class path ($Cfn =$ LJava/lang/Thread) or another class that it is extended from this class. Each thread object has a Boolean finished field indicating whether the thread has completed its execution or not, a mapping from a group of threads to a set of threads call stacks, it contains a set of threads waiting to join this thread and an attribute called *state* indicating the current state of the thread. Each thread has a run method. Thread attributes are initialized as follows:

- $finished \mapsto false$.

- $S_{join} \mapsto \emptyset$, an empty set of join threads, which means that initially there is no thread waiting to join the current thread.
- $state = \bot$.

Table 7. Semantic domains for a multi-threaded program.

(Global configuration)	Σ	$::= < C_s, S_{rbl}, H, S >$	
(Set of runnable threads)	S_{rbl}	$::= \emptyset \mid C_s \mid \{S_{rbl}, S_{rbl}\}$	
(Object)	Obj	$::= \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ acq \mapsto Val;\ S_{blck} \mapsto S_b;\ S_{wait} \mapsto S_w \vert\}$	
(A thread Object)	th	$::= \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto booelan;\ S_{join} \mapsto S_j \vert\}$	
(Set of blocked threads)	S_b	$::= \emptyset \mid C_s \mid \{S_b, S_b\}$	
(Set of waiting threads)	S_w	$::= \emptyset \mid C_s \mid \{S_w, S_w\}$	
(Set of join threads)	S_j	$::= \emptyset \mid C_s \mid \{S_j, S_j\}$	
(Acquiring field)	acq	$::\ f$	(field name)
(finished field)	$finished$	$::\ f$	(field name)
(Groups names)	$S_{wait}, S_{blck}, S_{join}$	$::=$ **String**	

Table 8 provides the semantics of spawning and scheduling threads. Rule R_{start} starts a new thread, which reference is stored in the register v_{ref}. It internally calls the referenced thread's $run()$ method that will be executed in this thread separately, once selected. Therefore, a $lookup()$ procedure for its run method is performed and a separate call stack for a new thread is created with one frame comprising all information about the thread's $run()$ method returned by the $lookup()$ function. This thread moves to a "runnable" state in S_{rbl}. When it gets a chance to execute, its target run() method will be executed. The actual execution of the launched thread will be managed with the rule R_{select}. Notice that, as expressed by the rule R_{start}, the reference of the launched thread is always stored in the register p_0 and we assume that it will remain there for all semantics rules and for all method's frames in the thread's call stack.

Table 8. Multi-threaded semantics: scheduling.

$$R_{start}\quad \dfrac{\begin{array}{c} m(i) = \textbf{start}\ v_{ref} \\ R(v_{ref}) = l \quad H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false;\ \ S_{join} \mapsto S_j \vert\} \\ lookup(run()V, Cfn) = run()V\ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, \mathbf{p_0} \mapsto \mathbf{l}\} \quad F_m =< @run, 0, R' > \end{array}}{<<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,S_{rbl} \cup \{F_m\},H,S>}$$

$$R_{select}\quad \dfrac{\begin{array}{c} selectFrom(S_{rbl}) = [F_m :: C_s, t_s]\ F_m =< m, i, R > \\ R(p_o) = l\ \ H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false; state \mapsto -\ S_{join} \mapsto S_j \vert\} \\ th' = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false; state \mapsto Running(t_s)\ S_{join} \mapsto S_j \vert\} \end{array}}{<\epsilon,S_{rbl},H,S> \xrightarrow{\tau} <F_m::C_s,S_{rbl} \backslash \{F_m::C_s\},H[l \mapsto th'],S>}$$

$$R_{stop}\quad \dfrac{\begin{array}{c} R(p_o) = l\ \ H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false; state \mapsto Running(t_s)\ S_{join} \mapsto S_j \vert\} \\ clock() > t_s \end{array}}{<<m,i,R>::C_s,S_{rbl},H\ S>_{\mathbf{M}} \xrightarrow{\tau} <\epsilon,S_{rbl} \cup \{C_s\},H,S>_{\mathbf{M}}}$$

Rules R_{select} and R_{stop} manage threads scheduling. Rule R_{select} selects from S_{rbl} one thread to be executed for a time slice t_s. The selected thread's state will be updated to a "Running(t_s)" state. The thread's call stack will be removed from the runnable set and placed at the first position of configuration Σ to start execution. The $select(S_{rbl})$ function will be based on a CFS scheduler's algorithm for scheduling threads in S_{rbl}. It takes into account the thread's nice values and returns the selected thread's local state presented in its current call stack as well as the time slice allocated to it for execution.

Rule R_{stop} stops, in a monitoring mode (i.e., a mode that monitors the execution time given to each thread), a thread whose allocated time slice to execute a task has expired. We model the timing aspect in our formalism by the function $clock()$ which represents the scheduler timer to control running threads.

Synchronization in *Dalvik* is modeled by the use of monitors with instructions *monitor-enter* and *monitor-exit*. That actually corresponds to the *synchronized* keyword in Java. A monitor is attached to an object and could be acquired and released by threads.

The semantics of these two instructions must fulfill two conditions. The first is related to the mutual exclusive access to shared objects in the heap by different threads. The second relates to the cooperation between these threads. Cooperation is modeled by a set of threads waiting for notification when the object is released by another thread. The sole thread running and owning the monitor is in a critical section. Table 9 presents rules related to synchronization. *Monitor-enter* semantics represent a thread trying to access the critical section by acquiring monitor for the object, whose reference is stored in a register v_{ref}. It first checks if the object is acquired by any other thread. If this is the case, the current thread will be blocked (mutual exclusive access condition) and added to the object blocking set S_{blck} to join other threads (if any) with the same situation (cooperation condition). This case is modeled by the rule R_{block}). Otherwise, the current thread can take ownership of the monitor. The *acq* attribute is then updated with this thread's reference. This thread could resume its execution in the critical section. This case is modeled with the rule $R_{acq-mnt}$.

Table 9. Multi-threaded semantics: synchronization.

$$R_{Acq-mntr}\ \frac{\begin{array}{c} m(i) = \textbf{monitor-enter}\ \ v_{ref} \\ R(v_{ref}) = l \quad H(l) = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto \bot; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w|\!\} \\ o' = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto R(p_0); S_{blck} \mapsto S_b; S_{wait} \mapsto S_w|\!\} \end{array}}{<<m,i,R>::C_s,S_{rbl}\ ,H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,S_{rbl},H[l \mapsto o'],S>}$$

$$R_{block}\ \frac{\begin{array}{c} m(i) = \textbf{monitor-enter}\ \ v_{ref} \\ R(v_{ref}) = l \quad H(l) = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l'; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w|\!\} \\ o' = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l'; S'_{blck} \mapsto S_b \cup \{< m, i, R >:: C_s\}; S_{wait} \mapsto S_w|\!\} \end{array}}{<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i,R>::C_s,S_{rbl},H[l \mapsto o'],S>}$$

$$R_{Rls-mntr}\ \frac{\begin{array}{c} m(i) = \textbf{monitor-exit}\ \ v_{ref} \\ R(p_0) = l' \quad R(v_{ref}) = l \quad H(l) = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l'; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w|\!\} \\ o' = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto \bot; S'_{blck} \mapsto \emptyset; S_{wait} \mapsto S_w|\!\} \end{array}}{<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,S_{rbl} \cup S_b,H[l \mapsto o'],S>}$$

$$R_{wait}\ \frac{\begin{array}{c} m(i) = \textbf{wait}\ v_{ref} \\ R(p_0) = l'\ R(v_{ref}) = l \quad H(l) == \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l'; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w|\!\} \\ o' = \{\!|Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto \bot; S_{blck} \mapsto \emptyset; S'_{wait} \mapsto S_w \cup \{< m, i, R >:: C_s\}\ |\!\} \end{array}}{<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i,R>::C_s,S_{rbl} \cup S_b,H[l \mapsto o'],S>}$$

Monitor-exit semantics represents a thread that reaches the end of the critical section by releasing the owned monitor for another thread to take ownership, which perfectly fulfills the cooperation condition. Rule $R_{Rls-mntr}$ provides this semantics, the current thread must first own this object's monitor, once this condition is satisfied, the *acq* attribute is updated to an undefined value (object is free). Then, all waiting threads in S_{blck} are removed to the runnable set S_{rbl}. It is up to the scheduler to select which thread to execute (there is no ordering among the blocked threads).

A thread could voluntarily give up ownership of the monitor before reaching the end of the critical section by calling the $wait()$ method or by executing the *wait* instruction. This thread releases ownership of this monitor and remains in a waiting state (i.e., suspended or inactive until be notified by another thread). Rule R_{wait} provides the semantics of wait instruction. The calling thread must own this object's monitor (i.e., must executing *wait* from inside a synchronized block) then relinquish it. Once the monitor associated with this object is released, the current thread is placed in the wait set for this object.

Table 10 presents rules R_{notify} and $R_{notifyAll}$ expressing the signaling mechanism. Rule R_{notify} represents the semantics for waking up a single thread that is waiting for this object's monitor in the waiting set S_{wait}. One thread among the set will be chosen randomly by the function $random()$. This thread will be moved from the waiting set to the runnable set to be selected later on by the scheduler and then processed. The rule $R_{notifyAll}$ is similar to the rule R_{notify}, with the exception that it wakes all threads in the waiting set, which ones will be moved to the runnable set S_{rbl}. Notice that, rules R_{notify} and $R_{notifyAll}$ release in addition to waiting thread(s) set S_{wait} all blocked threads in S_{blck}. The two sets have the same privileges with regards to acquiring monitor. In other words, waiting threads have no precedence over potentially blocked threads that also want to synchronize on this object.

Table 10. Multi-threaded semantics: signaling.

R_{notify}	$\dfrac{\begin{array}{c} m(i) = \textbf{notify}\ v_{ref} \\ R(p_0) = l \quad R(v_{ref}) = l' \quad H(l') = \{\vert Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w \vert\} \\ random(S_w) = C'_s \quad o' = \{\vert Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto \bot; S'_{blck} \mapsto \emptyset; S'_{wait} \mapsto S_w \setminus \{C'_s\} \vert\} \end{array}}{<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,S_{rbl} \cup \{C'_s\} \cup S_b,H[l' \mapsto o'],S>}$
$R_{notifyAll}$	$\dfrac{\begin{array}{c} m(i) = \textbf{notifyAll}\ v_{ref} \\ R(p_0) = l \quad R(v_{ref}) = l' \quad H(l') = \{\vert Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto l; S_{blck} \mapsto S_b; S_{wait} \mapsto S_w \vert\} \\ o' = \{\vert Cfn; (f_\tau \rightarrow Val)^*; acq \mapsto \bot; S'_{blck} \mapsto \emptyset; S'_{wait} \mapsto \emptyset \vert\}\} \end{array}}{<<m,i,R>:::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,S_{rbl} \cup S_w \cup S_b,H[l' \mapsto o'],S>}$

Table 11 presents semantics of finishing thread and joining instructions. Rules $R_{Join\text{-}exec}$ and $R_{Join\text{-}wait}$ check if the joined thread has finished its execution, if so, the current thread resumes execution ($R_{Join\text{-}exec}$). Otherwise, the rule $R_{Join\text{-}wait}$ is applied. The current running thread is removed into S_{join} for threads waiting for the same thread to complete its execution (no release by the monitor of the object is acquired by the running thread here). The rule R_{finish} ensures that when a thread completes its execution (i.e., its run() method returns) and releases all waiting threads in S_{join} by moving them to the runnable set S_{rbl}.

Table 11. Multi-threaded semantics: join.

$R_{Join\text{-}exec}$	$\dfrac{\begin{array}{c} m(i) = \textbf{join}\ v_{ref} \\ R(v_{ref}) = l \quad H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto true;\ \ S_{join} \mapsto S_j \vert\} \end{array}}{<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,\mathbf{i+1},R>::C_s,S_{rbl},H,S>}$
$R_{join\text{-}wait}$	$\dfrac{\begin{array}{c} m(i) = \textbf{join}\ v_{ref} \\ R(v_{ref}) = l \quad H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false;\ \ S_{join} \mapsto S_j \vert\} \\ o' = \{\vert\ Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false;\ \ S'_{join} \mapsto S_j \cup \{< m, i, R >:: C_s\}\ \vert\} \end{array}}{<<m,i,R>::C_s,S_{rbl},H,S> \xrightarrow{m(i)} <<m,i,R>::C_s,S_{rbl},H[l \mapsto o'],S>}$
R_{finish}	$\dfrac{\begin{array}{c} @run(i) = \textbf{return-void} \\ R(p_0) = l \quad H(l) = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto false;\ \ S_{join} \mapsto S_j \vert\} \\ o' = \{\vert Cfn; (f_\tau \rightarrow Val)^*;\ finished \mapsto true;\ \ S'_{join} \mapsto \emptyset \vert\} \end{array}}{<<@run,i,R>::\epsilon,S_{rbl},H,S> \xrightarrow{@run(i)} <\epsilon,S_{rbl} \cup S_j,H[l \mapsto o'],S>}$

6. Practical Aspects

We give, hereafter, some practical aspects of Smali+ through an example. For the sake of simplicity and due to the space limitation, we only present an illustration of a single-threaded program in Smali$^+$ that includes various important instructions such as method call, return, static and instance field update, etc. As shown in Table 12, the program is sequential and consists of two classes $c1$ and $c2$ belonging to the same package called p. Figure 3 shows the initial configuration. We show in detail, through this example, how the rules are applied and how the configuration evolves in every step. Each rule is followed by the resulting configuration.

Table 12. Smali$^+$ program.

```
     .class public Lp/c2 .super c1 {
     .field public x: int
     .field public y: char
     .method public static m1()V  .locals 3 {
       ...
5      move v1 30                              ...
6      goto 10
...    ...
10     invoke-static Lp/c1 m2(int,char)char v0,v1
11     move c2.x v0
12     new-instance v2 Lp/c1                   ...
13     move v2.b v1                            18
...  ...
     }

     .class public Lp/c1 .super ⊥ {
     .field public a: LJava/lang/String
     .field public b: int
     .field private final c: char
     .method public static m2(int,char)char .locals
     2 {
     ...
       ...
     return v1
     }
```

$\mathbf{C_s}$
$< m_1, 5, R >$

H

S			
Lp/c2	public	x	0
	public	y	'\\$u$0000'
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	'\\$u$0000'

R	v_0	v_1	v_2	*ret*
	5	⊥	⊥	⊥

R′	v_0	v_1	*ret*
	⊥	⊥	⊥

Figure 3. Initial configuration.

The first table corresponds to the call stack C_s, which is the current method frame. The second table corresponds to an empty heap H and the last two tables correspond to the register arrays for methods m_1 and m_2, respectively.

The first Smali$^+$ instruction to execute is the move instruction labeled with 5. It is a constant displacement, so the rule $R_{mv\text{-}cst}$ applies. Since constants are evaluated to themselves, the register v_1 for m_1 locals registers is updated by the constant value and the program counter is incremented.

$$R_{mv\text{-}cst}\ \frac{m_1(5) = \textbf{move}\ v_1\ 30 \qquad [\![30]\!]{=}\ 30}{<<m_1,5,R>::C_s,H,S> \xrightarrow{m_1(5)} <<m_1,6,R[v_1 \mapsto 30]>::C_s,H,S>}$$

$\mathbf{C_s}$
$< m_1, 6, R >$

H

S			
Lp/c2	public	x	0
	public	y	'\\$u$0000'
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	'\\$u$0000'

R	v_0	v_1	v_2	*ret*
	5	**30**	⊥	⊥

R'	v_0	v_1	*ret*
	⊥	⊥	⊥

The next instruction corresponds to the unconditional jump *goto*. The rule R_{goto} so applies to update the program counter by the instruction labeled with 10.

$$R_{goto}\ \frac{m_1(6) = \textbf{goto}\ 10}{<<m_1,6,R>::C_s,H,S> \xrightarrow{m_1(6)} <<m_1,10,R>::C_s\ ,H,S>}$$

$m_1(10)$ is an invocation of a static method. Rule $R_{inv\text{-}st}$ so applies. A new frame for the called method is pushed on top of C_s and the counter program in the caller method frame is incremented.

$\mathbf{C_s}$
$< m_1, \mathbf{10}, R >$

H

S			
Lp/c2	public	x	0
	public	y	$'\backslash u0000'$
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	ret
	5	30	$\bot$	$\bot$

R'	v_0	v_1	ret
	$\bot$	$\bot$	$\bot$

$$R_{\textit{inv-st}}\quad \frac{\begin{array}{c} m_1(10) = \textbf{invoke-static}\ Lp/c1\ m_2(int,char)char\ v_0,v_1 \\ lookup(m_2(int,char)char, Lp/c1) = m_2(int,char)char\ 2 \\ R' = \{v_0 \mapsto \bot, v_1 \mapsto \bot, p_1 \mapsto R(v_0), p_2 \mapsto R(v_1)\} \end{array}}{<<m_1,10,R_1>::C_s,H,S> \xrightarrow{m_1(10)} <<m_2,0,R'>::<m_1,\mathbf{11},R>::C_s,H,S>}$$

$\mathbf{C_s}$
$< \mathbf{m_2}, \mathbf{0}, \mathbf{R'} >$
$< m_1, \mathbf{11}, R >$

H

S			
Lp/c2	public	x	0
	public	y	$'\backslash u0000'$
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	ret
	5	30	$\bot$	$\bot$

R'	v_0	v_1	p_1	p_2	ret
	$\bot$	$\bot$	5	30	$\bot$

After some execution steps, we suppose that the register v_1 in m_2 is updated by a new value "CA" and the current instruction to execute is labeled with 18 in m_2.

$\mathbf{C_s}$
$< \mathbf{m_2}, \mathbf{18}, \mathbf{R'} >$
$< m_1, \mathbf{11}, R >$

H

S			
Lp/c2	public	x	0
	public	y	$'\backslash u0000'$
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	ret
	5	30	$\bot$	$\bot$

R'	v_0	v_1	p_1	p_2	ret
	$\bot$	CA	5	30	$\bot$

The instruction $m_2(18)$ is a return from a non-void method m_2, so the rule $R_{\textit{ret-nv}}$ applies. The top frame of C_s is popped and the return value is passed from the callee back to the caller through its return register *ret*.

$$R_{\textit{ret-nv}}\quad \frac{m_2(18) = \textbf{return}\ v_1}{<<m_2,18,R'>::<m_1,11,R>::C_s,H,S> \xrightarrow{m_2(18)} <<m_1,11,R[ret \mapsto R'(v_1)]>::C_s,H,S>}$$

$\mathbf{C_s}$
$< m_1, 11, R >$

H

S			
c2	public	x	0
	public	y	$'\backslash u0000'$
c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	ret
	5	30	$\bot$	CA

R'	v_0	v_1	p_1	p_2	ret
	$\bot$	CA	5	30	$\bot$

The instruction $m_1(11)$ is a static field update. So the rule $R_{\textit{mv-sttf}}$ so applies to update the indicated field in the static heap S by the register v_0 content.

$$R_{\textit{mv-sttf}}\quad \frac{\begin{array}{c} m_1(11) = \textbf{move}\ Lp/c2.x\ \ v_0 \\ R(v_0) = 5\quad [\![Lp/c2.x\ ]\!] = S(Lp/c2.x) \end{array}}{<<m_1,11,R>::C_s,H,S> \xrightarrow{m_1(11)} <<m_1,12,R>::C_s,H,S[Lp/c2.x \mapsto 5]>}$$

$\mathbf{C_s}$
$< m_1, 12, R >$

H

S			
Lp/c2	public	x	**5**
	public	y	$'\backslash u0000'$
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	ret
	5	30	$\bot$	CA

R'	v_0	v_1	p_1	p_2	ret
	$\bot$	$\bot$	5	30	$\bot$

The instruction $m_1(12)$ corresponds to an object creation. The rule $R_{\textit{new-instance}}$ so applies to create a new instance from the class $c1$ in the heap H and all fields are initialized according to their types.

$$R_{\textit{new-ins}}\quad \frac{\begin{array}{c} m_1(12) = \textbf{new-instance}\ v_2\ \ Lp/c1 \\ o' = \{|Lp/c1; (a \mapsto null, b \mapsto \backslash u0000', c \mapsto \backslash u0000')\}\quad l' \notin dom(H) \end{array}}{<<m_1,12,R>::C_s,H,S> \xrightarrow{m_1(12)} <<m_1,13,R[v_2 \mapsto l']>::C_s,H[l' \mapsto o'],S>}$$

The instruction $m_1(13)$ is an instance field update. So the rule $R_{\textit{mv-instf}}$ applies. The register v_2 holds the instance location o' in H. The instance field in o' is updated with the source register v_1 content.

C
$< m_1, \mathbf{13}, R >$

H			
l'	**o′**		
	Lp/c1	a	*null*
		b	0
		c	$'\backslash u0000'$

S			
Lp/c2	public	x	5
	public	y	$'\backslash u0000'$
Lp/c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	*ret*
	5	30	l′	⊥

R'	v_0	v_1	p_1	p_2	*ret*
	⊥	⊥	5	30	⊥

$$R_{mv\text{-}instf}\ \frac{\begin{array}{c} m_1(13) = \textbf{move}\ v_2.b\ v_1 \\ R(v_2) = l' \ \ R(v_1) = 30 \quad H(l') = o' \end{array}}{<<m_1{,}13{,}i{,}R>::C_s{,}H{,}S> \xrightarrow{m_1(13)} <<m{,}14{,}R>::C_s{,}H[l' \mapsto o'[b \mapsto 30]]{,}S>}$$

C
$< m_1, \mathbf{13}, R >$

H			
l'	**o′**		
	Lp/c1	a	*null*
		b	30
		c	$'\backslash u0000'$

S			
c2	public	x	5
	public	y	$'\backslash u0000'$
c1	public	a	*null*
	private	b	0
	pv/final	c	$'\backslash u0000'$

R	v_0	v_1	v_2	*ret*
	5	30	l′	⊥

R'	v_0	v_1	p_1	p_2	*ret*
	⊥	⊥	5	30	⊥

7. Discussion

So far, we have proposed a formal language for Android programs called Smali$^+$. Presented in a BNF notation, $Smali^+$ is a simple language that remains faithful to the original Smali notations and the .Smali file structure. It contains 12 generalized instructions from 218 Dalvik instructions [39] and some macros instructions modeling concurrency aspect. These 12 instructions were selected carefully to highlight Dalvik's characteristics, such as register-based architecture, assembly-like code for Smali, methods invocations, monitors, etc. Macro instructions were used for the sake of simplification as well as to model multi-threading in Android. All the important API methods that affect a thread life-cycle were considered in Smali$^+$ semantics.

Another important feature that lacks so far in Android application semantics is thread scheduling. This important aspect, in general, consists in picking a thread for execution and allocating an execution time to it, depending on its priority, before selecting a new thread to execute and switching the context. Android applications including their threads adhere to the Linux execution environment. So, threads are scheduled using the standard scheduler of the Linux kernel, known as a *completely fair scheduler* (CFS). On Linux, the thread priority is called a "nice value". A low nice value corresponds to a high priority and vice versa. In Android, a Linux thread has niceness values in the range of −20 (most prioritized) to 19 (least prioritized), with a default niceness of 0 [42]. We exhibited in this work two rules related to scheduling feature in Android, R_{select} and R_{stop}. In the first rule, we presented a function $select()$ that plays the same role as the CFS, meaning it selects from runnable threads the most prioritized thread based on nice values comparison and allocates to it an amount of time for execution. The second rule stops a thread when the allocated time expires, prior to picking a new one through R_{select}. We mean by "monitoring mode" mentioned in threads scheduling, a monitor that is based on the CFS algorithm that monitors each thread for each task executed, and we suppose that each rule in the concurrent context is executing under a monitoring mode. This mode was presented just for R_{stop} and omitted in other rules for simplification reasons.

The operational semantics are mainly created to secure Android applications. In fact, we intend to use these semantics in an upcoming work to check a number of security proprieties to protect users from rogue applications. Our ultimate goal is to formally reinforce security policies on Android applications. That is to say, starting from a Smali$^+$ program and a formal specification of a security policy, we automatically generate a new equivalent secure version of the original program that respects the security policy. Formally, the approach takes, as input, a Smali$^+$ program P and a formal specification of a security policy ϕ and generates, as output, a new version P' that respects ϕ. The new version of the program preserves all the behavior of the original version, except in cases where the security policy is on the verge of being violated. This is equivalent to saying that the traces of P' are the intersection of traces accepted by ϕ and traces of P. It is formally modeled by (1).

$$P' = P \cap \phi \tag{1}$$

Security policies will be enforced through a program-rewriting approach that combines static and dynamic approaches. It rewrites the program statically, according to a given security property, then generates a new executable version that satisfies this property. Security modifications or tests are added at well-calculated points in the program to force the latter to conform to the security property during execution. In other words, the untrusted code will be transformed into a self-monitoring code that will be exploded at specific points in the program. The rewritten version should be equivalent but more restrictive than the original so that it will be able to avoid potentially dangerous operations before they occur.

Reinforced security properties will obviously be specific to malware and attacks threatening Android applications, such as sensitive information leakage, which could be SMS contents, call logs, contact information or geographical location or Android financial malware, which exploit the premium services to incur financial loss to the user for the benefit of the attacker, for example, by calling or texting to premium-rate numbers without the user's consent and privilege escalation attacks [43]. Therefore, all mediums that could be exploited for this kind of malware, such as Internet access, system services access including SMS, contact, telephony, Bluetooth, Global Positioning System (GPS) as well as APIs resulted from inter-application communication, will be checked through security policies. Such APIs will be easily located in Smali$^+$, since it provides for each invocation the class fully qualified name.

8. Conclusions

In this paper, we have proposed a formal operational semantics for Smali, an assembly-like code generated form reverse engineering Android applications. We called the new formal language Smali$^+$. Smali$^+$ covers the semantics of a large subset of the main Dalvik instructions as well as many important aspects related to multi-threading programming which are rarely considered in the state-of-the-art works of Android applications. This formal model is meant to be an environment to run formal verification of applications. Broader work consisting in techniques to reinforce the security of Android applications using this formalism is currently underway. We are deeply convinced that this will be of great help in analyzing the security of Android applications and verifying their hidden functions affecting users' privacy as well as protecting users from malicious actions.

Author Contributions: Conceptualization, M.Z., J.F. and M.M.; methodology, M.Z., M.M. and J.F; validation, J.F., M.M. and E.P.; formal analysis, M.Z., M.M. and J.F.; investigation, M.Z., J.F., M.M. and E.P.; resources, M.Z., J.F., M.M. and E.P.; writing–original draft preparation, M.Z. and J.F.; writing–review and editing, M.Z. and J.F.; supervision, M.M., J.F. and E.P.; project administration, M.M.; funding acquisition, M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. IDC Corporation. Smartphone Market Share. Available online: https://www.idc.com/promo/smartphone-market-share/os (accessed on 19 February 2020).
2. Zhou, Y.; Jiang, X. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012. [CrossRef]
3. Sergiu Gatlan. Anubis Android Trojan Spotted with Almost Functional Ransomware Module. Available online: https://www.bleepingcomputer.com/news/security/anubis-android-trojan-spotted-with-almost-functional-ransomware-module/ (accessed on 20 February 2020).
4. Barrett, L. SMS-Sending Trojan Targets Android Smartphones. Available online: https://www.esecurityplanet.com/trends/article.php/3898041/SMSSending-Trojan-Targets-Android-Smartphones.htm/ (accessed on 2 January 2020).

5. Collier, N. New Android Trojan Malware Discovered in Google Play. Available online: https://blog.malwarebytes.com/cybercrime/2017/11/new-trojan-malware-discovered-google-play// (accessed on 2 January 2020).
6. F-Secure. Trojan:Android/GGTracker Available online: https://www.f-secure.com/v-descs/trojan_android_ggtracker.shtml (accessed on 2 January 2020).
7. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; McDaniel, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* **2014**, *49*, 259–269. [CrossRef]
8. Felt, A.P.; Chin, E.; Hanna, S.; Song, D.; Wagner, D. Android Permissions Demystified. In Proceedings of the 18th ACM Conference on Computer and Communications Security, New York, NY, USA, October 2011; doi:10.1145/2046707.2046779. [CrossRef]
9. Davis, B.; Sanders, B.; Khodaverdian, A.; Chen, H. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In Proceedings of the Mobile Security Technologies 2012, San Francisco, CA, USA, May 2012. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.7191&rep=rep1&type=pdf (accessed on 2 January 2020).
10. Xu, R.; Saïdi, H.; Anderson, R.J. Aurasium: Practical Policy Enforcement for Android Applications. In Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012; pp. 539–552.
11. Jeon, J.; Micinski, K.K. *SymDroid : Symbolic Execution for Dalvik*; CS-TR-5022; University of Maryland: College Park, MD, USA, July 2012. Available online: http://www.cs.tufts.edu/~jfoster/papers/symdroid.pdf (accessed on 4 January 2020).
12. Apktool. A Tool for Reverse Engineering Android Apk Files. Available online: https://ibotpeaches.github.io/Apktool/ (accessed on 19 February 2019).
13. Na, G.; Lim, J.; Kim, K.; Yi, J.H. Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART. *J. Internet Serv. Inf. Secur.* **2016**, *6*, 27–39.
14. El-Zawawy, M.A. An Operational Semantics for Android Applications. In Proceedings of the Computational Science and Its Applications - ICCSA 2016 - 16th International Conference, Beijing, China, 4–7 July 2016; pp. 100–114.
15. Payet, E.; Spoto, F. An Operational Semantics for Android Activities. Available online: https://doi.org/10.1145/2543728.2543738 (accessed on 5 December 2019).
16. Wognsen, E.; Karlsen, S. Static Analysis of Dalvik Bytecode and Reflection in Android. Master's Thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark, 6 June 2012. Available online: https://projekter.aau.dk/projekter/files/63640573/rapport.pdf (accessed on 10 December 2019).
17. Wognsen, E.; Søndberg Karlsen, H.; Chr. Olesen, M.; Hansen, R. Formalisation and analysis of Dalvik bytecode. *Sci. Comput. Program.* **2014**, *92*, 25–55. [CrossRef]
18. Cousot, P.; Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, New York, NY, USA, January 1977; pp. 238–252. [CrossRef]
19. Payet, E.; Spoto, F. Static Analysis of Android Programs. *Inf. Softw. Technol.* **2012**, *54*, 1192–1201. [CrossRef]
20. Gunadi, H. Formal Certification of Non-interferent Android Bytecode (DEX Bytecode). In proceedings of the 2015 20th International Conference on Engineering of Complex Computer Systems ICECCS, Gold Coast, Australia, 9–12 December 2015; pp. 202–205.
21. Gunadi, H.; Tiu, A.; Gore, R. Formal Certification of Android Bytecode. *arXiv* **2015**, arXiv:1504.01842v5. Available online: https://arxiv.org/abs/1504.01842 (accessed on 19 February 2020).
22. Barthe, G.; Pichardie, D.; Rezk, T. A certified lightweight non-interference Java bytecode verifier. *Math. Struct. Comput. Sci.* **2013**, *23*, 1032–1081. [CrossRef]
23. Maiya, P.; Kanade, A.; Majumdar, R. Race detection for Android applications. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, UK, 9–11 June 2014; pp. 316–325.

24. Kanade, A. Chapter Seven - Event-Based Concurrency: Applications, Abstractions, and Analyses. *Adv. Comput.* **2019**, *112*, 379–412.
25. Bouajjani, A.; Emmi, M.; Enea, C.; Ozkan, B.K.; Tasiran, S. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. In Proceedings of the Programming Languages and Systems 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, Uppsala, Sweden, 22–29 April 2017; pp. 170–200.
26. Calzavara, S.; Grishchenko, I.; Koutsos, A.; Maffei, M. A Sound Flow-Sensitive Heap Abstraction for the Static Analysis of Android Applications. *arXiv* **2017**, arXiv:1705.10482v2. Avaiable online: https://arxiv.org/pdf/1705.10482.pdf (accessed on 15 December 2019).
27. Calzavara, S.; Grishchenko, I.; Maffei, M. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroSP), aarbrucken, Germany, 21–24 March 2016. [CrossRef]
28. Chaudhuri, A. Language-based security on Android. In Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, Dublin, Ireland, 15–21 June 2009. [CrossRef]
29. Chen, T.; He, J.; Song, F.; Wang, G.; Wu, Z.; Yan, J. Android Stack Machine. Computer Aided Verification. In Proceedings of the 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, Oxford, UK, 14–17 July 2018; pp. 487–504. [CrossRef]
30. He, J.; Chen, T.; Wang, P.; Wu, Z.; Yan, J. Android Multitasking Mechanism: Formal Semantics and Static Analysis of Apps. In Proceedings of the Programming Languages and Systems - 17th Asian Symposium, Nusa Dua, Bali, Indonesia, 1–4 December 2019; pp. 291–312. [CrossRef]
31. Bagheri, H.; Kang, E.; Malek, S.; Jackson, D. Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In Proceedings of the FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, 24–26 June 2015; pp. 73–89. [CrossRef]
32. Ren, L.; Chang, R.; Yin, Q.; Man, Y. A Formal Android Permission Model Based on the B Method. In Proceedings of the Security, Privacy, and Anonymity in Computation, Communication, and Storage 10th International Conference, Guangzhou, China, 12–15 December 2017; pp. 381–394. [CrossRef]
33. Khan, W.; Kamran, M.; Ahmad, A.; Khan, F.A.; Derhab, A. Formal Analysis of Language-Based Android Security Using Theorem Proving Approach. *IEEE Access* **2019**, *7*, 16550–16560. [CrossRef]
34. Qin, J.; Zhang, H.; Wang, S.; Geng, Z.; Chen, T. Acteve++: An Improved Android Application Automatic Tester Based on Acteve. *IEEE Access* **2019**, *7*, 31358–31363. [CrossRef]
35. Anand, S.; Naik, M.; Harrold, M.J.; Yang, H. Automated concolic testing of smartphone apps. In Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), Cary, NC, USA, 11–16 November 2012; p. 59. [CrossRef]
36. Nisi, D.; Bianchi, A.; Fratantonio, Y. Exploring Syscall-Based Semantics Reconstruction of Android Applications. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses, Beijing, China, 23–25 September 2019; pp. 517–531.
37. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing Inter-application Communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, New York, NY, USA, June 2011; pp. 239–252. [CrossRef]
38. Drake, J.J.; Lanier, Z.; Mulliner, C.; Fora, P.O.; Ridley, S.A.; Wicherski, G. *Android Hacker's Handbook*; Wiley Publishing: Hoboken, NJ, USA, 2014.
39. Android Open Source Project (AOSP). Dalvik Bytecode. Available online: https://source.android.com/devices/tech/dalvik/dalvik-bytecode (accessed on 30 January 2020).
40. Oracle Corporation. Java Documentation on Thread. Available online: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html (accessed on 2 October 2019).
41. Oracle Corporation. Java Documentation on Object. Available online: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html (accessed on 2 October 2019).

42. Göransson, A. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications*, 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2014; ISBN 978-1449364137.
43. Davi, L.; Dmitrienko, A.; Sadeghi, A.; Winandy, M. Privilege Escalation Attacks on Android. In Proceedings of the Information Security 13th International Conference, Boca Raton, FL, USA, 25–28 October 2010; pp. 346–360.