Alma Mater Studiorum
Università degli Studi di Bologna

Tesi di Laurea

Laurea Magistrale in Ingegneria Informatica

PRISM

Code transplantation for adversarial malware

Relatore
Chiar.ma Prof.ssa Rebecca Montanari
Università degli Studi di Bologna

Correlatore
Chiar.mo Prof. Lorenzo Cavallaro
King's College London

Candidato
Jacopo Cortellazzi

Anno Accademico 2017/2018

PRISM

Index

1. Abstract

At the beginning of the Information Age, malware has been a problem only for a very restricted circle of people who had the possibility and the knowledge to access computers. Nowadays, however, due to the large adoption of mobile devices, the Digital Revolution re-shaped everyone's life; more and more people have been brought into the cyberspace and, at the very same time, malware reached its widest attack surface: the world.

In the nefarious fight against attackers, a wide range of smart algorithms have been introduced, in order to block and even prevent new families of malware before their appearance. Machine learning, for instance, recently gained a lot of attention thanks to its ability to use generalization to possibly detect never-before-seen attacks or variants of a known one. During the past years, a lot of works have tested the strength of machine learning in the cybersecurity field, exploring its potentialities and weaknesses. In particular, various studies highlighted its robustness against adversarial attacks, proposing strategies to mitigate them [1].

Unfortunately, all these findings have focused in testing their own discoveries just operating on the dataset at feature layer space, which is the virtual data representation space, without testing the current feasibility of the attack at the problem space level, modifying the current adversarial sample [2] .

For this reason, in this dissertation, we will introduce PRISM, a framework for executing an adversarial attack operating at the problem space level. Even if this framework focuses only on Android applications, the whole methodology can be generalized on other platforms, like Windows, Mac or Linux executable files.

The main idea is to successfully evade a classifier by transplanting chunks of code, taken from a set of *goodware* to a given malware. Exactly as in medicine, we have a donor who donates organs and receivers who receive

them, in this case, *goodware* applications are our donors, the organs are the needed code and the receiver is the targeted malware.

In the following work we will discuss about concepts related to a wide variety of topics, ranging from machine learning, due to the target classifier, to static analysis, due to the possible countermeasures considered, to program analysis, due to the extraction techniques adopter, ending in mobile application, because the target operating system is Android.

The whole work idea was born thanks to constructive confrontations with the members of Systems Security Research Lab (S2LAB) [3] and the whole framework has been developed into their laboratories. All of this has been possible thank to them, for their resources and smart ideas.

The whole dissertation has been developed in five main sections. The next Section, which is Section 2, contains all the background knowledge needed for correctly understand this work, without getting in deep into PRISM. Instead, in Section 3 and 4, the whole framework will be presented. Section 3 presents the high-level ideas and strategies behind it, while Section 4 contains information on the current implementation, examples included. The last Section contains the conclusions, final reflections regarding its effectiveness and future works plus the experiments done.

2. Background

In this first paragraph we will introduce the basic concepts needed for understand **PRISM**, starting from what a malware is, followed by a brief introduction on basics static analysis concepts and the different possible usages of machine learning in malware analysis. At last, we will introduce the Android mobile framework, which is the mobile framework that this dissertation focuses on.

Indeed, this work can be summarized as an advanced Android code injector which is going to be used to successfully evade a target classifier, supposing a white-box scenario, in which the attacker has full knowledge of the target classifier to be evaded. This can be successfully accomplished by extracting benign features from *goodware* and transplant all the features environment (dependencies, invocation, variables, etc.) into the malware application in order to bypass the target classifier check.

One of the main ideas is translating the whole attack from a feature-space attack, which consist in creating a perturbation in the feature space layer, to a problem-space attack, instrumenting the adversarial application at bytecode level. This kind of contribution totally lacks in the current academic literature, bringing an interesting opportunity in proposing a new approach to the adversarial attack scenario. The whole framework focuses on the Android platform, allowing to successfully extract and inject chunks of code from different source and destination applications and successfully evading a target classifier. Of course, this has also several limitations for now and needs future development, but the same approach could be generalised an applied on different kind of binaries, suggesting an alternative way for look at adversarial attacks.

2.1. Mobile malware

First and foremost: what is a malware? Shortly, malware is malicious software created with the only purpose to harm or manipulate a target device in some ways.

Malware does the damage after it is implanted or introduced in some way into a target's computer and can take the form of executable code, scripts, active content, and other software. The "executable code" we are referring to is often labelled as computer viruses, worms, Trojan horses, ransomware, spyware, adware, and scareware, among other terms. Malware has a malicious intent, acting against the interest of the end-user; so it does not include any software that causes unintentional harm due to some deficiency, which is typically described as a software bug.

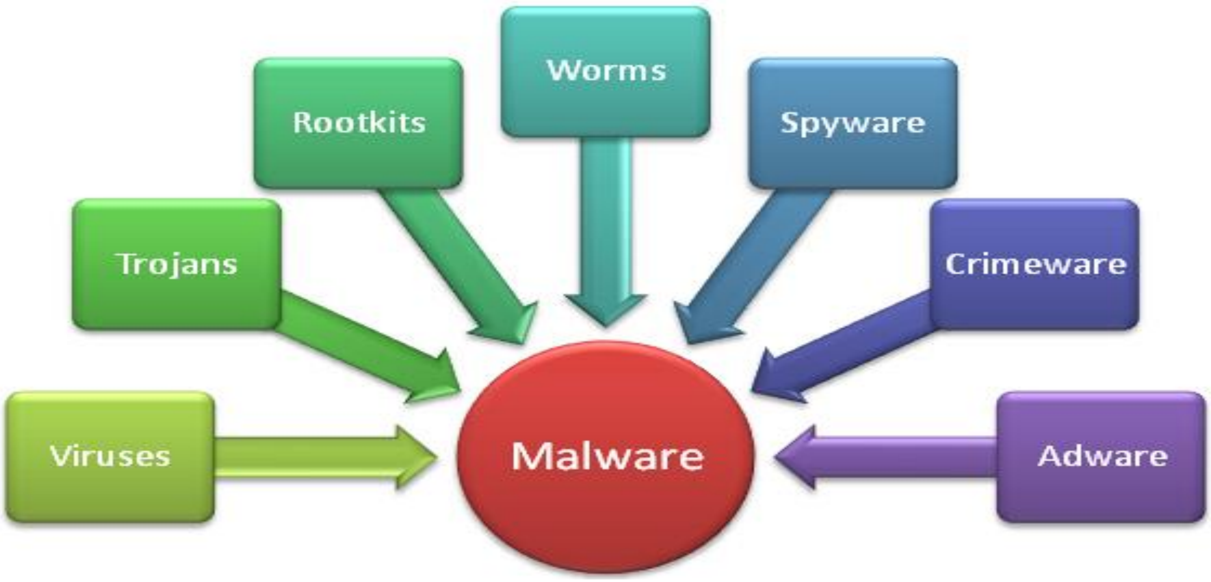


Figure 2.1

The number of these malicious programs is rising over and over during the years, reaching alarming peaks: a noteworthy example is WannaCry, which outbreak is reported as one of the most concerning episodes of malware infection during the last years [4].

The only way to protect from these kind of threats is using proper antivirus software, which try to cope with the constant virus evolution by leveraging novel analysis techniques: machine learning algorithms, for instance, can spot malware by analysing their behaviour, without solely relying on known fingerprints. Even if this kind of protection has been consolidated for standard devices, like laptop and computers, there is substantial lack of protection in the mobile world, thus leading virus creators and hackers to focus on deploying mobile malware. This also depends on the role that these mobile devices have in our lives and how they are integrated in the all-day routine.

The mobile OS which count the highest number of malicious application is Android, reaching almost the 3.5 million during the last year [5]. Indeed, it is more exposed to malicious application than iOS because it is possible to download and install applications from unofficial stores, which usually offers paid applications for free or applications which are not listed on Google Play, and also because the security checks done by Google Play Store are lighter if compared with the ones done by the Apple Store. The latter uses different dynamic and static analysis techniques in order to deeply understand the application execution and evaluate its compatibility with their policies. Moreover, Apple devices do not allow to install applications which are not approved by the Apple Store, further limiting the final user freedom, protecting him at the same time.

For all those reasons, Android OS is targeted by malicious applications. So, what specifically is a mobile malware? There are different forms of mobile malware, not all of which are the same as those affecting desktop operating systems, even if sharing one common goal: taking control of the target device.

As following, some examples of malware categories:

- Trojans: provide a backdoor, enabling an attacker to remotely execute code or control a device.
- Keyloggers: which also sometimes include screenscrapers, sit on a user's device, logging all keystrokes in an attempt to capture valuable information.

- Bank trojans: this type of malware is particularly attractive to mobile attackers, as it combines a trojan with a keylogger. Attackers either intercept a user's legitimate banking app information or trick users into downloading fraudulent banking apps.
- Ransomware: a type of malware that will encrypt a user's data and hold it for "ransom" until a payment is sent to the attacker.
- Ghost push: a malware form that can target Android devices, getting root access and then pushing software updates or malicious ads onto a user device.
- Adware: even if not always defined or identified as malware, ads can sometimes be laced with tracking components (sometimes called *spyware*) that will collect information on user activity.

There are currently some vendors that propose antivirus applications for mobile, but their effectiveness is very limited since they are less robust than the standard ones on desktop pc.

2.2. Malware static analysis

Basic static analysis consists of examining the executable file without running the executable code. Basic static analysis can confirm whether a file is malicious, providing information about its functionalities. Static analysis may be represented by alternative abstractions built atop of a program, which could lead to different kind of resource consumption and complexity, allowing to analyse distinct characteristics of a program from different point of views, as deeply explained later.

On the other side dynamic analysis is the evaluation of a program or technology using real-time data. This method of analysis can be done on a virtual processor or on a real processor. Instead of taking code offline, vulnerabilities and program behaviour can be monitored while the program is running, providing visibility into its real-world behaviour. A dynamic test will monitor system memory, functional behaviour, response time, and overall performance of the system. Further details are not covered since dynamic analysis is out of the scope of this dissertation.

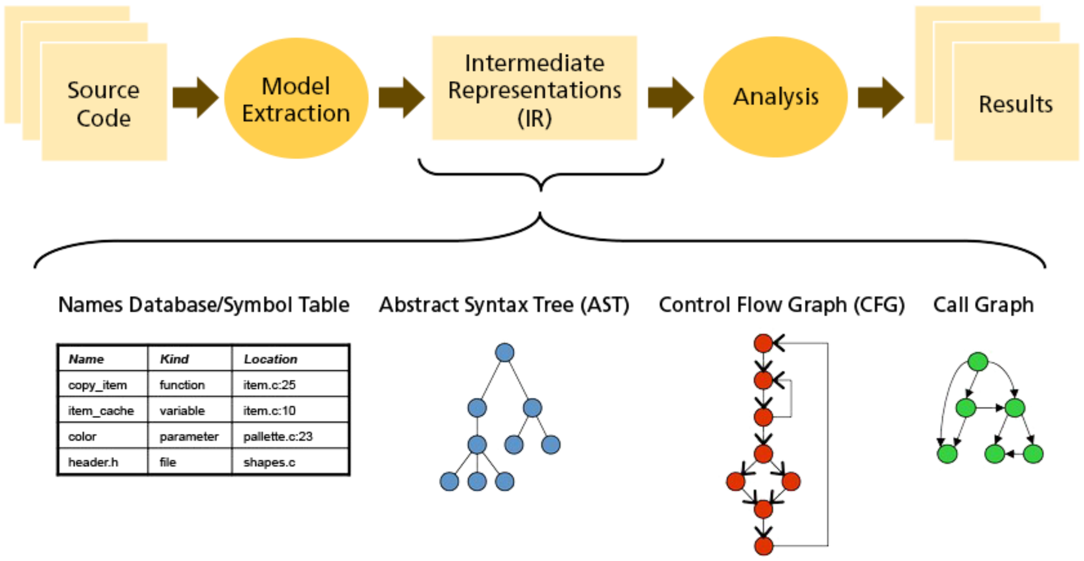
One of the major benefits of static-based detection is that it can be performed before the file is executed (also referred as *pre-execution*). This is obviously useful because it is much easier to remediate malware if it is never allowed to execute. An ounce of prevention is worth a pound of cure. A corollary of this benefit is that even corrupt and malformed executables which will not execute can still be detected statically. An example of basic static analysis indeed relies on the pipeline showed in Figure 1.8: given a program to analyse, the first thing is trying to extract strings and to compare the signatures of the file, searching for some malicious entry.

As for disadvantages of static detection, it is more difficult to detect completely new and novel threats that are sufficiently unlike any previously analysed sample. One reason for this is that it is much easier to manipulate the structure of an executable than it is to alter its behaviour. Consider the behaviour of ransomware: a legitimate app can be modified in such a way that it does not appear to be malicious, yet it acts like a ransomware. Looking just at the file structure, it is possible to figure out which functions it imports, but there is no way to know if and when they are called or in what order. Since most of the app’s code and structure is legitimate, it may be hard to detect a particular file depending on the sophistication of the static analysis technique that is been used. On the other hand, a trivial dynamic analysis tool can raise suspicion on a program opening many files, calling cryptography functions, writing new files, and deleting existing ones soon after starting. This behaviour looks highly suspicious and is mostly endemic to ransomware.

2.2.1. Graphs

Static analysis strongly relies on the evaluation of the following representational trees:

- Abstract Syntax Tree: is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.
- Control Flow Graph: is the graphical representation of the execution flow of a program, i.e. the order in which instructions and functions are executed. They are mostly used either in static analysis or in compiler applications.
- Call Graph: is an artefact produced by program analysis tools to record the relationships between a function and the functions it calls. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so static call graph algorithms are generally over-approximations. That is,


Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Database/Symbol Table
Abstract Syntax Tree (AST)
Control Flow Graph (CFG)
Call Graph

Name	*Kind*	*Location*
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

Figure 2. 2

every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

A basic static analysis is shown in Figure 2.2. Anyway, by static analysis it is also possible to classify each reference to a variable in a program as a definition or a use, focusing then on Data Flow Information. Indeed, by inspecting each statement in a program, we can identify the definitions and uses in that statement. This identification is called local data flow analysis. Although local data flow information can be useful for activities such as local optimization, many important compiler and software engineering tasks require information about the flow of data across statements. Data flow analysis that computes information across statements is called global data flow analysis, or equivalently, intraprocedural data flow analysis. Another kind of static analysis is the one that focus on program dependencies, extracting which are the set of dependencies for a specific program or more specifically a code part. A first example is the Control Dependence Graph, which encodes control dependencies and assumes that nodes do not postdominate [6, p. 3] themselves. Into this kind of graph the nodes represent statements, or regions of code that have common control dependencies. Another important program representation is the Program Dependence Graph, which represents both control dependencies and data dependencies for a program. A PDG consists of a control dependence, to which data dependence edges have been added.

2.2.2. Program slicing

The concept of a program slice was originally developed by Weiser [7] for debugging purposes. Based on his original definition, informally, a static program slice S consists of all statements in program P that may affect the value of variable v in a statement x. The slice is defined for a slicing criterion $C = (x, v)$, where x is a statement in program P and v is variable in x.

A static slice includes all the statements that can affect the value of variable v at statement x for any possible input. Static slices are computed by backtracking dependencies between statements. More specifically, to compute the static slice for (x, v), we first have to find all the statements that can directly affect the value of v before statement x is encountered. Recursively, for each statement y which can affect the value of v in statement x, we compute the slices for all variables z in y that affect the value of v. The union of all those slices is the static slice for (x, v).

This is called backward slice and it is represented in the Figure 2.3 as example.

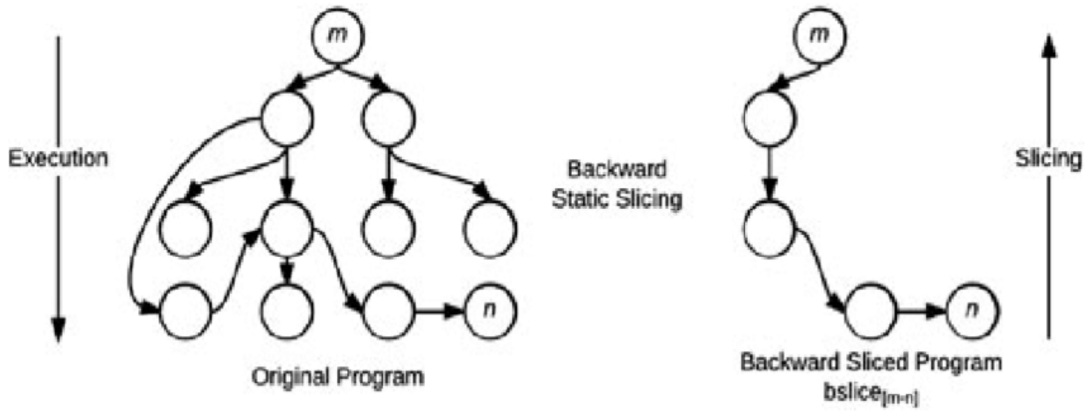


Figure 2.3

There have been several useful extensions to slicing algorithms. One problem with the static backward slice that was developed by Weiser is that it contains all the statements that may affect a given statement during any program execution. A refinement of this static slice that eliminates the problem of additional statements is a dynamic slice [8]. Whereas a static slice is computed for all possible executions of the program, a dynamic slice contains only those statements that affect a given statement during one execution of the program. Thus, a dynamic slice is associated with each test case for the program. Another useful type is the forward slice [9]. A forward slice for a program point P and a variable v consists of all those statements and predicates in the program that might be affected by the value of v at P. A forward slice is computed by taking the transitive closure of the statements directly affected by the value of v at P.

2.2.3 Drebin Model

This approach can be successfully adopted also for the Android platform. The model that we have adopted for PRISM is Drebin [10]. As the first step, Drebin performs a lightweight static analysis of a given Android application. Accordingly, it becomes essential to select features which can be extracted efficiently. It indeed focuses on the manifest and the disassembled dex code of the application, which both can be obtained by a linear sweep over the application's content. To allow for a generic and extensible analysis, we represent all extracted features as sets of strings, such as permissions, intents and API calls. In particular it distinguishes between features extracted from the Manifest and the ones extracted from the dex.

Every application developed for Android must include a manifest file called AndroidManifest.xml which provides data supporting the installation and later execution of the application. The information stored in this file can be efficiently retrieved on the device using the Android Asset Packaging Tool, which enables us to extract the following sets:

- S1 Hardware components: This first feature set contains requested hardware components. If an application requests access to the camera, touchscreen or the GPS module of the smartphone, these features need to be declared in the manifest file. Requesting access to specific hardware has clearly security implications, as the use of certain combinations of hardware often reflects harmful behaviour.
- S2 Requested permissions: One of the most important security mechanisms introduced in Android is the permission system. Permissions are actively granted by the user at installation time – and from Android 6.0 also at runtime - and allow an application to access security-relevant resources. For example, malicious software tends to request certain permissions more often than innocuous applications.
- S3 App components: There exist four different types of components in an application, each defining different interfaces to the system: activities, services, content providers and broadcast receivers. Every application can declare several components of each type in the manifest. The names of these components are also collected in a

feature set, as the names may help to identify well-known components of malware.

- S4 Filtered intents: Inter-process and intra-process communication on Android is mainly performed through *intents*: passive data structures exchanged as asynchronous messages and allowing information about events to be shared between different components and applications. We collect all intents listed in the manifest as another feature set, as malware often listens to specific intents.

Another set of features can be extracted from the disassembled code of Android applications: the bytecode can be efficiently disassembled and provides Drebin with information about API calls and data used in an application. The framework uses this information to construct the following feature sets:

- S5 Restricted API calls: The Android permission system restricts access to a series of critical API calls. Our method searches for the occurrence of these calls in the disassembled code in order to gain a deeper understanding of the functionality of an application.
- S6 Used permissions: The complete set of calls extracted in S5 is used as a foundation to determine the subset of permissions that are both requested and actually used.
- S7 Suspicious API calls: Certain API calls allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. As these calls can specially lead to malicious behaviour, they are extracted and gathered in a separated feature set.
- S8 Network addresses: Malware regularly establishes network connections to retrieve commands or exfiltrate data collected from the device. Therefore, all IP addresses, hostnames and URLs found in the disassembled code are included in the last set of features.

2.2.4 Soot framework

In order to understand all the operations done in this dissertation, we will introduce Soot, which is the main framework on which all the modules are based on.

Soot is the core framework of both the Extractor and Injector module. Soot is a product of the Sable research group from McGill University, whose objective is to provide tools leading to the better understanding and faster execution of Java programs [11]. One of the main benefits of Soot is that it provides four different Intermediate Representations (IR) for analysis purposes. Each of the IRs has different levels of abstraction that give different benefits when analysing, they are: Baf, Grimp, Jimple and Shimple. During this thesis work we have mostly used the Jimple IR which is going to be deeply explained later. Soot builds data structures to represent:

- `Scene`. The `Scene` class represents the complete environment the analysis takes place in. Through it, it is possible to set e.g., the application classes (the ones supplied to Soot for analysis), the main class (the one that contains the main method) and access information regarding interprocedural analysis (e.g., points-to information and call graphs).
- SootClass. Represents a single class loaded into Soot or created using Soot.
- SootMethod. Represents a single method of a class.
- SootField. Represents a member field of a class. Body. Represents a method body and comes in different flavours, corresponding to different IRs (e.g., JimpleBody).

These data structures are implemented using Object-Oriented techniques and designed to be easy to use and generic where possible. A statement in Soot is represented by the interface Unit, which there are different implementations of, one for each IR (e.g. Jimple uses Stmt). Through a Unit we can retrieve values used, values defined or both.

Additionally, we can get at the units jumping to a specific unit and units a units is jumping to, by jumping we mean control flow other than falling through.

Unit also provides various methods of querying about branching behaviour. A single datum is represented as a Value. Examples of values are: locals (Local), constants (in Jimple Constant), expressions (in Jimple Expr), and many more. An expression has various implementations, e.g. BinopExpr and InvokeExpr, but in general can be thought of as carrying out some action on one or more Values and returns another Value.

References in Soot are called boxes, which can be distinguished in ValueBox and UnitBox. UnitBoxes refer to Units. Used when a single unit can have multiple successors, i.e. when branching. ValueBoxes refer to Values. As previously described, each unit has a notion of values used and defined in it, this can be very useful for replacing use or def boxes in units, for instance when performing constant folding. All these elements are summarized in Figure 2.4.

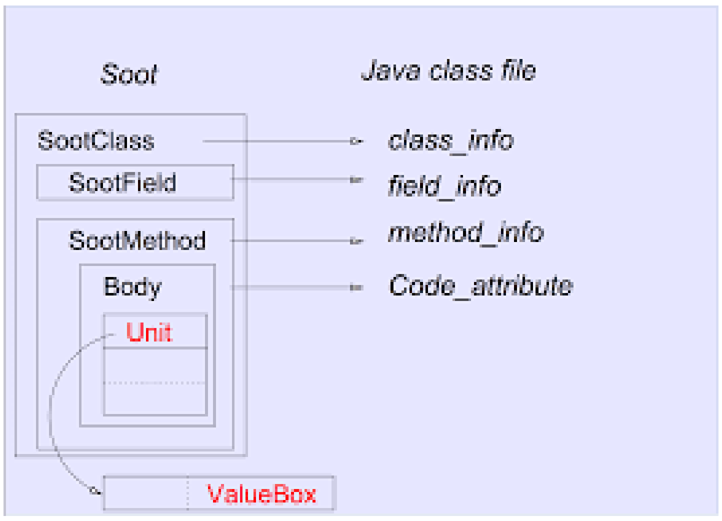


Figure 2.4

As we already mentioned, the Soot framework provides four intermediate representations for code: Baf, Jimple, Shimple and Grimp. The representations provide different levels of abstraction on the represented code and are targeted at different uses. Because this work has focused only on the Jimple IR, we will explain only this one with further details.

Jimple is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation. Jimple representations can be created directly in Soot or based on Java source code and Java bytecode/Java class files. The translation from bytecode to Jimple is performed using a naïve translation from bytecode to untyped Jimple, by introducing new local variables for implicit stack locations and using subroutine elimination to remove jump-to-subroutine instructions. Types are inferred for the local variables in the untyped Jimple and added. The Jimple code is cleaned for redundant code, like unused variables or assignments. An important step in the transformation to Jimple is the linearization (and naming) of expressions: this makes statements only reference at most 3 local variables or constants, resulting in a more regular and very convenient representation for performing optimizations. In Jimple an analysis only has to handle the 15 statements in the Jimple representation compared to the more than 200 possible instructions in Java bytecode.

An example of Jimple transformation is the following, shown in Figure 2.5.

Java	**Jimple**
1 static int factorial (int x){	1 static int factorial (int) {
2 int result = 1;	2 int x, result, i, temp$0, temp$1, temp$2, temp$3;
3 int i = 2;	3 x := @parameter0: int; /*1*/
4 while (i <= x) {	4 result = 1; /*2*/
5 result *= i;	5 i = 2; /*3*/
6 i++;	6
7 }	7 label0:
8 return result;	8 nop; /*3*/
9 }	9 if i <= x goto label1; /*4*/
	10 goto label2;/*4*/
	11
	12 label1:
	13 nop; /*4*/
	14 temp$0 = result; /*4*/
	15 temp$1 = temp$0 * i; /*4*/
	16 result = temp$1; /*5*/
	17 temp$2 = i; /*5*/
	18 temp$3 = temp$2 + 1; /*6*/
	19 i = temp$3; /*6*/
	20 goto label0; /*4*/
	21
	22 label2:
	23 nop; /*4*/
	24 return result; /*8*/
	25 }

Figure 2.5

Soot also provides several different control flow graphs (CFG), defining methods to get:

- entry and exit points to the graph;
- successors and predecessors of a given node;
- an iterator to iterate over the graph in some undefined order and the graphs size (number of nodes).

The following implementations are those that represent a CFG in which the nodes are Soot Units. Furthermore, we will only describe those that represent an intraprocedural flow. The base class for these kinds of graphs is UnitGraph, an abstract class that provides facilities to build CFGs. There are three different implementations of it: BriefUnitGraph, ExceptionalUnitGraph and TrapUnitGraph. The one adopted in **PRISM** is the ExceptionalUnitGraph which includes all the basic blocks and also edges from throw clauses to their handler (catch block, referred to in Soot as Trap), that is if the trap is local to the method body. Additionally, this graph takes into account exceptions that might be implicitly thrown by the VM (e.g. ArrayIndexOutOfBoundsException). For every unit that might throw an implicit exception, there will be an edge from each of that unit predecessors to the respective trap handler's first unit. Furthermore, in case the excepting unit would contain side effectsm an edge will also be added from it to the trap handler.

Over the Control Flow Graph, Soot allows to build a Call Graph, which is crucial for interprocedural analysis. A call graph in Soot is a collection of edges representing all known method invocations. This includes:

- explicit method invocations
- implicit invocations of static initializers
- implicit calls of Thread.run()
- implicit calls of finalizers
- implicit calls by AccessController

- and many more

Each edge in the call graph contains four elements: source method, source statement (if applicable), target method and the kind of edge. The different kinds of edges are e.g. for static invocation, virtual invocation and interface invocation. The call graph has methods to query for the edges coming into a method, edges coming out of method and edges coming from a particular statement. Each of these methods return an Iterator over Edge constructs. Soot provides three so-called adapters for iterating over specific parts of an edge:

- Sources iterates over source methods of edges;
- Units iterates over source statements of edges;
- Targets iterates over target methods of edges.

2.3 Machine Learning in malware analysis

Machine learning is a set of methods that gives “computers the ability to learn without being explicitly programmed[1]”.

In other words, a machine learning algorithm discovers and formalises the principles that underlie the data it sees. With this knowledge, the algorithm can reason about the properties of previously unseen samples. In malware detection, a previously unseen sample could be a new file. Its hidden property could be either a malicious or a benign one. A mathematically formalised set of principles underlying data properties is called the model. Machine learning has a broad variety of approaches that it takes to a solution rather than a single method. These approaches have different capacities and different tasks that they suit best. We have two possible approaches to this, unsupervised learning and supervised learning. In the first setting scenario, we are given only a dataset without the right answers for the task: the goal is to discover the structure of the data or the law of data generation. One important example is clustering. Clustering is a task that includes splitting a data set into groups of similar objects. Another task is representation learning –this includes building an informative feature set for objects based on their low-level description. Large unlabelled datasets are available to cybersecurity vendors and the cost of their manual labelling by experts is high – this makes unsupervised learning valuable for threat detection. Clustering can help with optimizing efforts for the manual labelling of new samples. With informative embedding, we can decrease the number of labelled objects needed for the usage of the next machine learning approach in our pipeline: supervised learning.

This other kid of setting is used when both the data and the right answers for each object are available. The goal is to fit the model that will produce the right answers for new objects. Supervised learning consists of two stages:

[1] Arthur Samuel, 1959

- Training a model and fitting a model to available training data. This mainly consists on using some amount of data to teach a method on how to estimate F, which is a function that roughly fits the data. Our goal is to apply a statistical learning method to the training data in order to estimate the unknown function f. In other words, we want to find a function F such that $Y \approx F(X)$ for any observation (X, Y).
- Applying the trained model to new samples and obtaining predictions.

The task is that, given a set of objects, in such a way that each one is represented with feature set X and mapped to right answer or labelled as Y, we want to create the best possible model that will produce the correct label Y' for previously unseen objects given the feature set X'. In the case of malware detection, X could be some features of file content or behaviour, for instance, file statistics and a list of used API functions. Labels Y could be "malware" or "benign", or even a more fine-grained classification, such as a virus, Trojan-Downloader or adware.

In the "training" phase, we need to select some family of models, for example, neural networks or decision trees. Usually each model in a family is determined by its parameters. Training means that we search for the model from the selected family with a particular set of parameters that gives the most accurate answers for train objects according to some metric. In other words, we "learn" the optimal parameters that define valid mapping from X to Y. After we have trained a model and verified its soundness, we are ready for the next phase – applying the model to new objects. In this phase, the type of the model and its parameters do not change: the model only produces predictions. In the case of malware detection, this is the protection phase. Vendors often deliver a trained model to users where the product makes decisions based on model predictions autonomously. Mistakes can cause devastating consequences for a user – for example, removing an OS driver. It is crucial for the vendor to select a model family properly. The vendor must use an efficient training procedure to find the model with a high detection rate and a low false positive rate.

It is important to emphasise the data-driven nature of this approach. A created model depends heavily on the data it has seen during the training phase to determine which features are statistically relevant to predict the correct label.

We will explain why making a representative data set is so important. Imagine we collect a training set, and we overlook the fact that occasionally all files larger than 15 MB are all malware and not benign, which is certainly not true for real world files. While training, the model will exploit this property of the dataset, and will learn that any files larger than 15 MB are malware. It will use

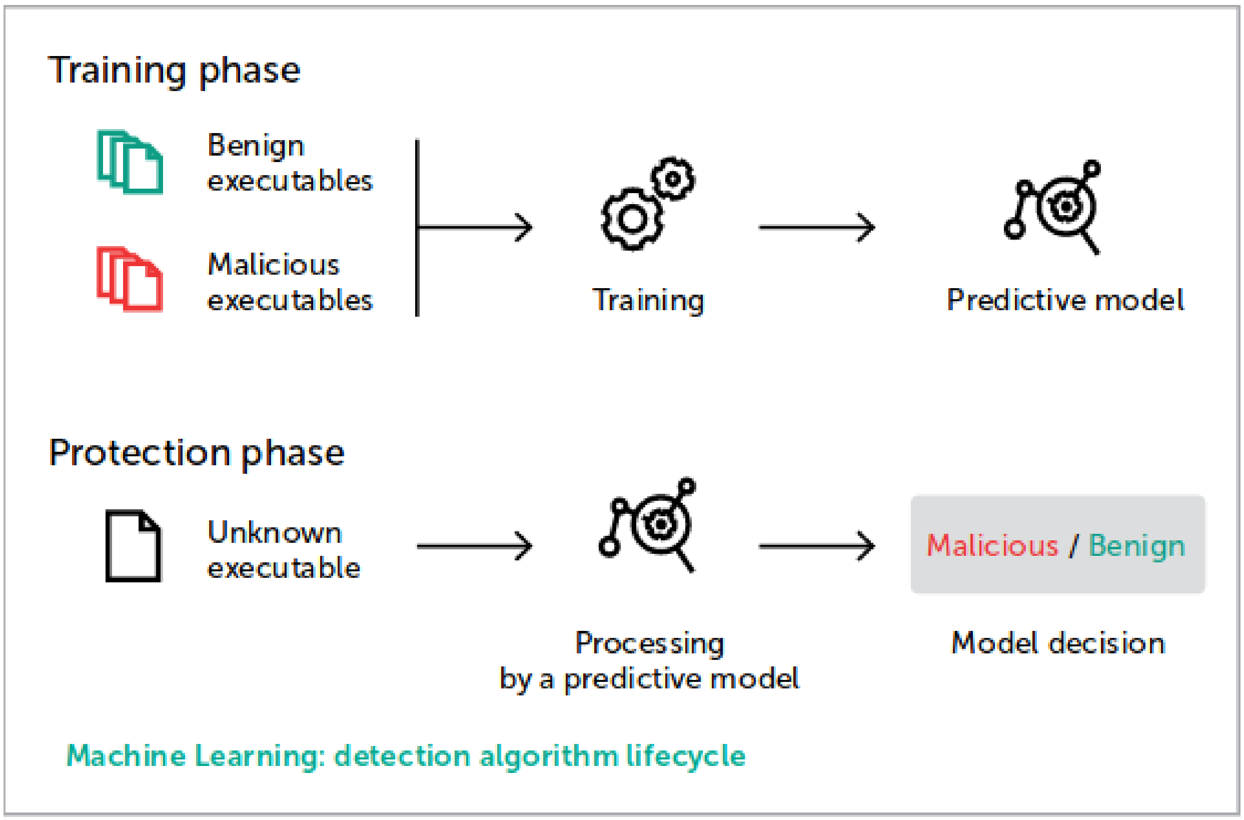


Figure 2. 6

this property for detection. When this model is applied to real world data, it will produce many false positives.

To prevent this outcome, we needed to add benign files with larger sizes to the training set. Then, the model would not rely on an erroneous data set property. Generalising this, we must train our models on a dataset that correctly represents the conditions where the model will be working in the real world. This makes the task of collecting a representative dataset crucial for machine learning to be successful.

False positives happen when an algorithm mistakes a malicious label for a benign file. Our aim is to make the false positive rate as low as possible, or "zero". This is untypical for machine learning application. It is important, because even one false positive in a million benign files can create serious consequences for users. This is complicated because there are lots of clean files in the world, and they keep appearing. To address this problem, it is important to impose high requirements for both machine learning models and metrics that will be optimized during training, with the clear focus on low false positive rate (FPR) models. This is still not enough, because new benign files that go unseen earlier may occasionally be falsely-detected. We take this into account and implement a flexible design of a model that allows us to fix false-positives on the fly, without completely retraining the model. Examples of this are implemented in our pre- and post-execution models, which are described in following sections.

Of course, this approach has also its own limits and can be tricked by different kind of adversarial examples. In order to make it understandable, we will start to explain the issue from the images.

Indeed, several machine learning models consistently misclassify inputs formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed input results in the model outputting an incorrect answer with high confidence.

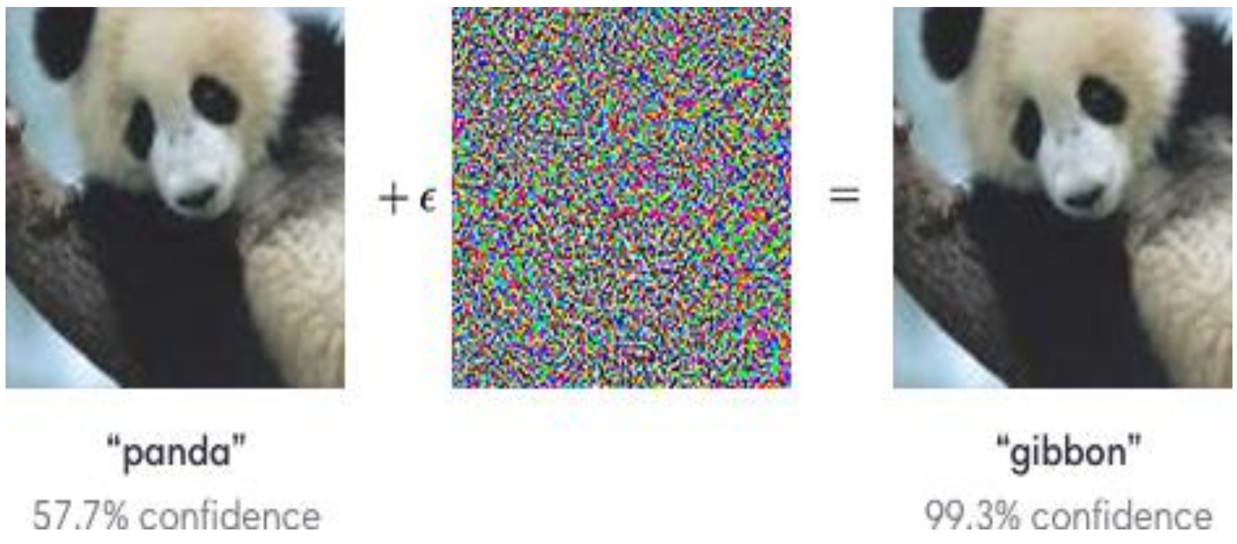


Figure 2. 7

For example, starting with an image of a panda Fig 2,7, the attacker adds a small perturbation that has been calculated to make the image be recognised as a gibbon with high confidence. The challenge for the adversary is figuring out how to generate an input with the desired output, as in the source-target-misclassification attack. In such an attack, the adversary starts with a sample that is legitimate (such as a Panda) and modifies it through a perturbation process to attempt to cause the model to classify it in a chosen target class (which in our case is a gibbon).

For an attack to be worth studying, from a machine learning point of view, it is necessary to impose constraints that ensure that the adversary is not able to truly change the class of the input. For example, if the adversary could physically replace a stop sign with a yield sign or physically paint a yield symbol onto a stop sign, a machine learning algorithm must be able to still recognize it as a yield sign [12]. In the context of computer vision, we generally consider only modifications of an object's appearance that do not interfere with a human observer's ability to recognize the object. The search for misclassified inputs is thus done with the constraint that these inputs should be visually very similar to a legitimate input. Consider the images in Fig 2.8 , potentially consumed by an autonomous vehicle. To the human eye, they appear to be the same, and our biological classifiers (vision) identify each one as a stop sign. The image on the left is indeed an ordinary image of a stop sign. We produced the image on the right by adding a small, precise perturbation that forces a particular image classification deep neural network to classify it as a yield sign. Here, the adversary could potentially use the altered image to cause the car to behave dangerously, especially if the car lacks of fail-safes (such as maps of known stop-sign locations). In other application domains, the constraint differs. When targeting machine learning models used for malware detection, the constraint becomes that the input—or malware software—misclassified by the model must still be in a legitimate executable format and execute its malicious logic when executed.

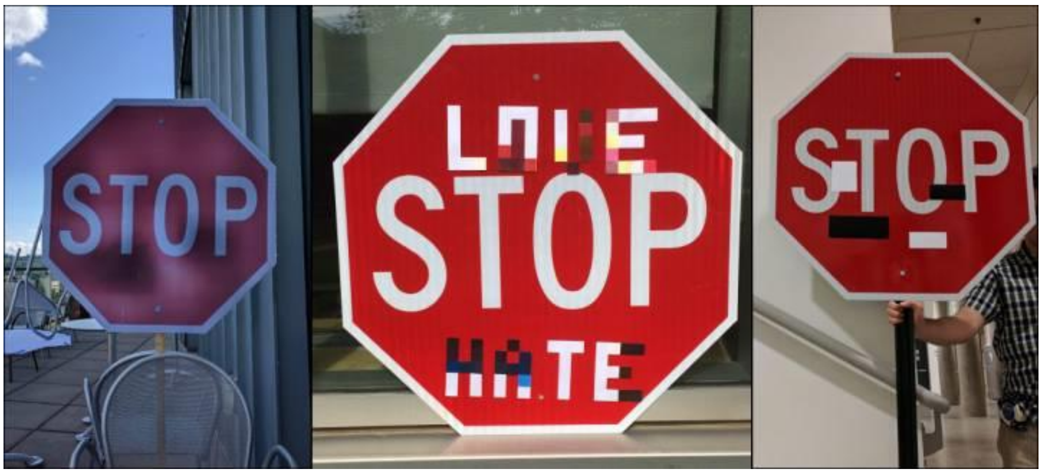


Figure 2.8

The same approach could exactly be adopted for malwares: if we apply a string perturbation to a malware we could achieve the misclassification of it for a single classifier, evading it. This is a challenging problem and we can suppose two different scenarios: the white-box one and the black-box one. One way to characterise an adversary's strength is the amount of access the adversary has to the model. In a white-box scenario, the adversary has full access to the model whereby the adversary knows what machine learning algorithm is being used and the values of the model's parameters. In this case, we show in the following paragraphs that constructing an adversarial example can be formulated as a straightforward optimization problem. In a black-box scenario, the attacker must rely on guesswork, because the machine learning algorithm used by the defender and the parameters of the defender's model are not known. Even in this scenario, where the attacker's strength is limited by incomplete knowledge, the attacker might still succeed. We now describe the white-box techniques first because they form the basis for the more difficult black-box attacks.

We will keep clarifying a bit more on the white-box case because it is the one on which we have focused on during the PRISM implementation.

An adversarial example x^* is found by perturbing an originally correctly classified input x. To find x^*, one solves a constrained optimization problem. One very generic approach, applicable to essentially all machine learning paradigms, is to solve for the x^* that causes the most expected loss (a metric reflecting the model's error), subject to a constraint on the maximum allowable deviation from the original input x; in the case of machine learning models solving classification tasks, the loss of a model can be understood as its prediction error. Another approach, specialised to classifiers, is to impose a constraint that the perturbation must cause a misclassification and solve for the smallest possible perturbation

$$x' = x + argmin\{||z|| : f(x+z) = t\}$$

where x is an input originally correctly classified, $|| \bullet ||$ a norm that appropriately quantifies the similarity constraints discussed earlier, and t is the target class chosen by the adversary. In the case of "untargeted attacks," t can be any class different from the correct class $f(x)$. For example, for a malware the adversary might use the ℓ_0 "norm" to force the attack to modify very few pixels, or the ℓ_∞ norm to force the attack to make only very small changes to each pixel. All of these different ways of formulating the optimization problem search for an x^* that should be classified the same as x (because it is very similar to x) yet is classified differently by the model. These optimization problems are typically intractable, so most adversarial example-generation algorithms are based on tractable approximations.

The limitations of existing defences point to the lack of theory and practice of verification and testing of machine learning models. To design reliable systems, engineers engage in both testing and verification. By "testing", we mean evaluating the system under various conditions and observing its behaviour, watching for defects. By "verification", we mean producing a compelling argument that the system will not misbehave under a broad range of circumstances.

Machine learning practitioners have traditionally relied primarily on testing. A classifier is usually evaluated by applying the classifier to several examples drawn from a test set and measuring its accuracy on these examples.

To provide security guarantees, it is necessary to ensure properties of the model besides its accuracy on naturally occurring test-set examples. One well-studied property is robustness to adversarial examples. The natural way to test robustness to adversarial examples is simply to evaluate the accuracy of the model on a test set that has been adversarial perturbed to create adversarial examples.

Unfortunately, testing is insufficient to provide security guarantees, as an attacker can send inputs that differ from the inputs used for the testing process. In general, testing is insufficient because it provides a "lower bound" on the failure rate of the system when an "upper bound" is necessary to provide security guarantees. Testing identifies *n* inputs that cause failure, so the engineer can conclude that at least *n* inputs cause failure; the engineer would prefer to have a means of becoming reasonably confident that at most *n* inputs cause failure.

Putting this in terms of security, a defence should provide a measurable guarantee that characterises the space of inputs that cause failures. Conversely, the common practice of testing can only provide instances that cause error and is thus of limited value in understanding the robustness of a machine learning system. Development of an input-characterizing guarantee is central to the future of machine learning in adversarial settings and will almost certainly be grounded in formal verification.

Adversarial machine learning is at a turning point. In the context of adversarial inputs at test time, we have several effective attack algorithms but few strong countermeasures. Can we expect this situation to continue indefinitely? Can we expect an arms race with attackers and defenders repeatedly seizing the upper hand in turn? Or can we expect the defender to eventually gain a fundamental advantage?

We can explain adversarial examples in current machine learning models as the result of unreasonably linear extrapolation but do not know what will happen when we fix this particular problem; it may simply be replaced by another equally vexing category of vulnerabilities. The vastness of the set of all possible inputs to a machine learning model seems to be cause for pessimism. Even for a relatively small binary vector, there are far more possible input vectors than there are atoms in the universe, and it seems highly improbable that a machine learning algorithm would be able to process all of them acceptably. On the other hand, one may hope that as classifiers become more robust, it could become impractical for an attacker to find input points that are reliably misclassified by the target model, particularly in the black-box setting.

These questions may be addressed empirically, by actually playing out the arms race as new attacks and new countermeasures are developed. We may also be able to address these questions theoretically, by proving the arms race must converge to some asymptote. All these endeavours are difficult, and we hope many will be inspired to join the effort.

2.4 Android mobile framework

The mobile OS on which we are focusing on is Android. The whole Android stack is show in the Figure 2.9, and can be divided in the following sections:

- Linux kernel: At the bottom of the layers is Linux - Linux 3.6 with approximately 115 patches. This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware.

- Libraries: On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

- Android libraries: This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

- Android Runtime: This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called Dalvik Virtual Machine which is a kind of Java Virtual Machine specially designed and optimized for Android. The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine. The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

- Application Framework: The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.
- Applications: You will find all the Android application at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

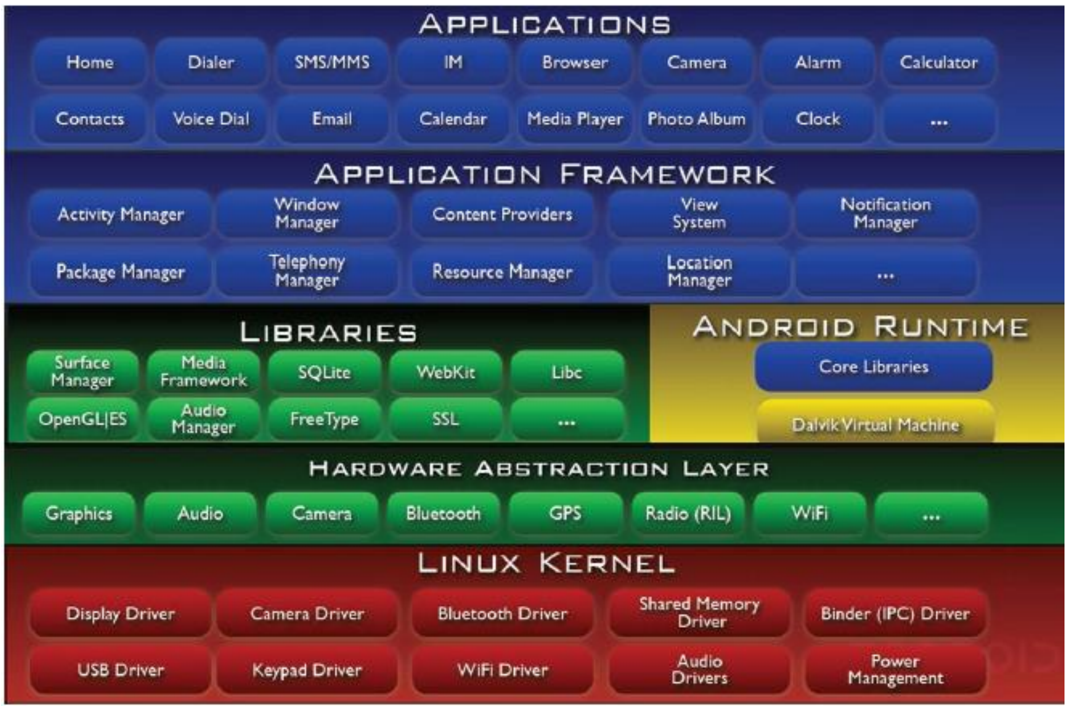


Figure 2.9

It is a complex architecture as we could notice, including different modules and components.

The main Android components are the following four:

- Activity which is the component associated to the UI. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI. The lifecycle of this component is shown in Fig 2.10.

- Broadcast Receivers, components that Android apps can use for send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow.
- Content Provider, which can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate data and provide mechanisms for defining data security.
- Services, application components that can perform long-running operations in the background, and that do not provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application.

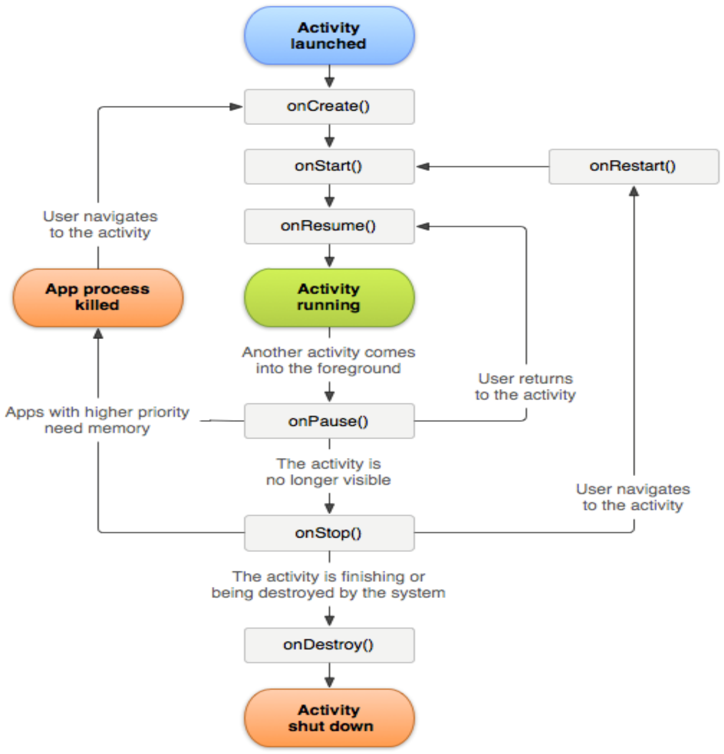

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2. 10

Another important entity of the Android framework are Intents. An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

- Starting an activity: one can start a new instance of an Activity by passing an Intent to `startActivity()`. The Intent describes the activity to start and carries any necessary data.
- Starting a service: to start a service and perform a one-time operation (such as downloading a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data. If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService(). For more information, see the Services guide.
- Delivering a broadcast: i.e. a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. It is possible to deliver a broadcast to other apps by passing an Intent to sendBroadcast() or sendOrderedBroadcast().

We will deal with Intents during the Extraction phase, when we will search for eligible Activity invocations, which consist into a `startActivity()` call.

Android is nowadays the most spread and adopted mobile OS in the world, counting now over 2 billion monthly active devices all over the globe. Android and iOS have created a duopoly in the smartphone market, accounting for more than 95% of the 3.1 billion active smartphone devices in the world. In terms of the sheer volume of devices in use, Android dominates iOS by a large margin with a 75.9% market share in November of 2017 or 2.3 billion smartphones in use. [13]

This gap keep increasing over and over the years, rising also the attentions of virus creator. Indeed, due to this mass adoption and its openness to unofficial applications, Android owns the largest number of malicious applications in the mobile world, counting, in the first half of 2018, 2,040,293 new malware samples recorded.

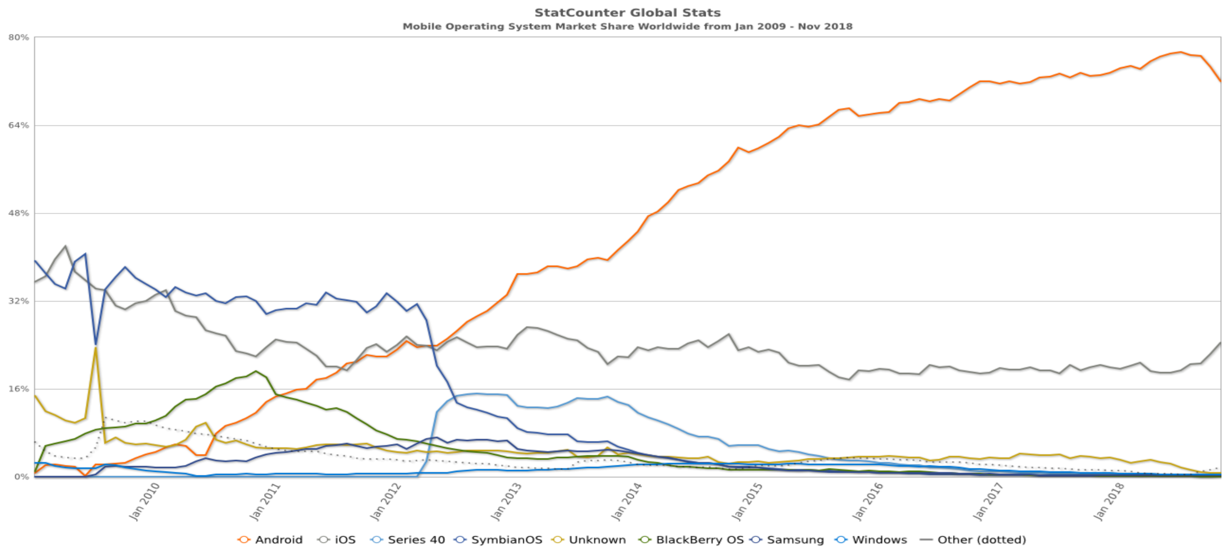


Figure 2.11

The forecasts are not even better: Android malware on the march, with ransomware an ever-increasing threat [14]. According to a SophosLabs analysis, the number of attacks on Sophos customers using Android devices increased almost every month in 2017 as shown in the Fig 2.12.

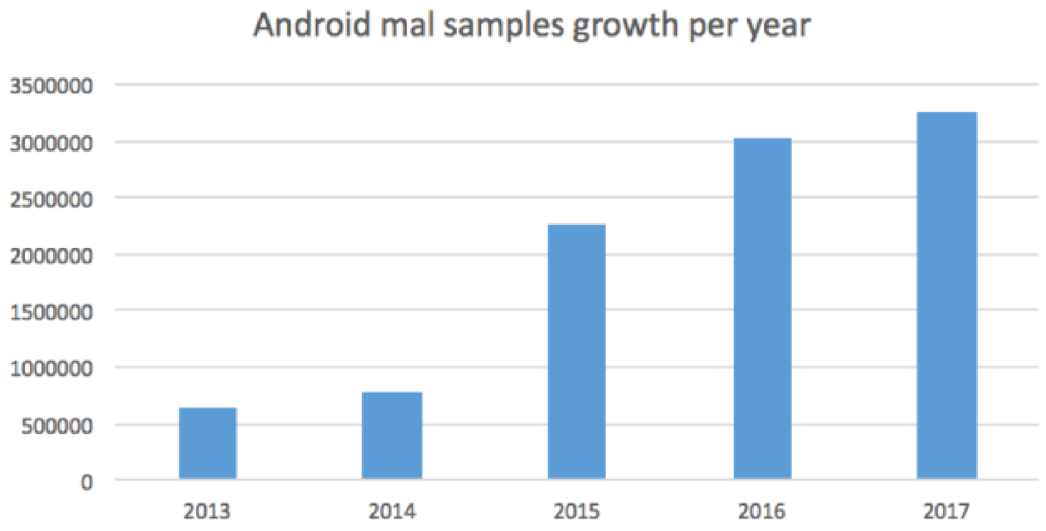


Figure 2. 12

Creating a malicious apk is not even hard, there are several ways to accomplish that. On the web it is possible to find various tutorial that explain how to create a simple malicious application by just injecting a malicious reverse shell into an arbitrary application.[2] This can be accomplished by just decompiling the apk in smali code and choose the best injection point for being sure that the malicious code is always executed; then Metasploit framework will handle the rest.

[2] Null-Byte, "How to do a persistent backdoor into an Android apk," [Online]. Available: https://null-byte.wonderhowto.com/how-to/create-persistent-back-door-android-using-kali-linux-0161280/

3 PRISM framework overview

As previously introduced, PRISM is a program synthesis framework which allows to successfully inject chunks of code accurately picked from a set of benign application into a desired malware, in order to successfully misclassify it against a specific classifier.

We decide to transpose the feature layer attacks to the problem space, operating directly on the bytecode instrumentation of the desired application to verify the feasibility of these kind of attacks.

The core characteristics that must be provided are:

- Successfully extract and transplant the whole dependency chain of the desired feature
- Being the more realistic as possible, in order to adapt to real world scenario
- Being stealthy
- Being flexible and robust for support different kind of applications

The whole deployment strategies focus on the satisfaction of all these requirements, in order to create a tool suitable for real-world usage scenarios. During the development we took advantage of various techniques from different computer science fields, like program analysis instruments for extract the dependencies, inject realistic invocation and use opaque predicates for shield our injected code. This grants this framework a high degree of robustness to static analysis tool, leaving the dynamic part for future expansion.

During this chapter we will discuss in detail which is the intuition behind the proposed work, starting from the need of operating at problem space level. Then we will explain the components of the overall framework and how they interoperate.

3.1 From feature space to problem space

In the current literature the almost entire set of proposed adversarial attacks work on feature layer level. It means that these attacks verify their effectiveness only operating on modifying the features in the data domain, without reflecting those changes also on the analysed object. They just focus on altering the value or the presence of a specific feature at dataset level, without testing the feasibility of this action.

A clear example of this is the SecSVM evasion algorithm [1]. It mainly consists in ordering by absolute weight the features of a specific classifier and decide to remove or add some to a given malicious application in order to misclassify it, operating only at feature layer space. The limitation of this study lies on the fact that they have not tested its feasibility in real world scenarios, but they limited the scope of the tests at the feature layer space, modifying the chosen features only in the dataset.

In the proposed work, we adopt the same assumptions: PRISM uses a LinearSVM classifier, due to the simple need to classify a *goodware* from a malware, which is represented simply as a linear space, as true or false. We worked into a Perfect Knowledge scenario, which is the worst-case setting because also the targeted classifier is known to the attacker. Indeed, this setting is particularly interesting as it provides an upper bound on the performance degradation incurred by the system under attack and can be used as reference to evaluate the effectiveness of the system under the other simulated attack scenarios.

Taken the inspiration from the previous evasion attack proposed, we slightly modified it, adding some basic constraint to preserve the semantic equivalence of the malware. The adopted attack can be explained as follows: first, we extract the estimated weights $[W1, W2, W3, \dots Wn]$ from the classifier and sort them in descending order of their absolute values. Since we are only interested in adding features for maintaining the program semantic equivalence from the original malware to the instrumented one, we discard any positive weight (which are malicious features) and hold all the negative ones (which are benign). Then, for all the extracted features, we verify the current presence in the target malware and, if not, we add it. After every injection, we check the current entity of the perturbation created and so, given w_i one of the weights of the feature considered, φ_{min} the minimum perturbation needed for the misclassification and α an ad hoc constant:

$$\sum_{i=1}^{n} w_i > \varphi_{min} + \alpha$$

If the condition is satisfied we stop the process and we verify that effectively the malware is now misclassified from the SVM.

This way we are able to obtain the minimum set of features needed in order to misclassify the target malware, but we only have the list of the needed features at features space. We do not obtain the real set of features to inject, but only the seeds of them. Indeed we want to consider the whole set of dependencies of a single component containing the searched feature. Simply adding a single feature to a program implies considering all its dependency chain and not doing that would be deeply inconsistent and easy to detect.

Getting from this feature-layer attack to its corresponding problem space is not immediate as one could imagine and, most of the times, a single feature drags a large set of other entities in order to work correctly. For this reason, we need to understand if considering all this set of dependencies the gadget to be injected preserve its benign properties, which is not an assumed property.

Indeed it is possible to include by dependency also malicious feature whose weight could totally overwhelm the initial benign one and turn the all injection into a malicious injection instead of a benign one, increasing the malicious property of the final malware, which is absolutely what we want to avoid.

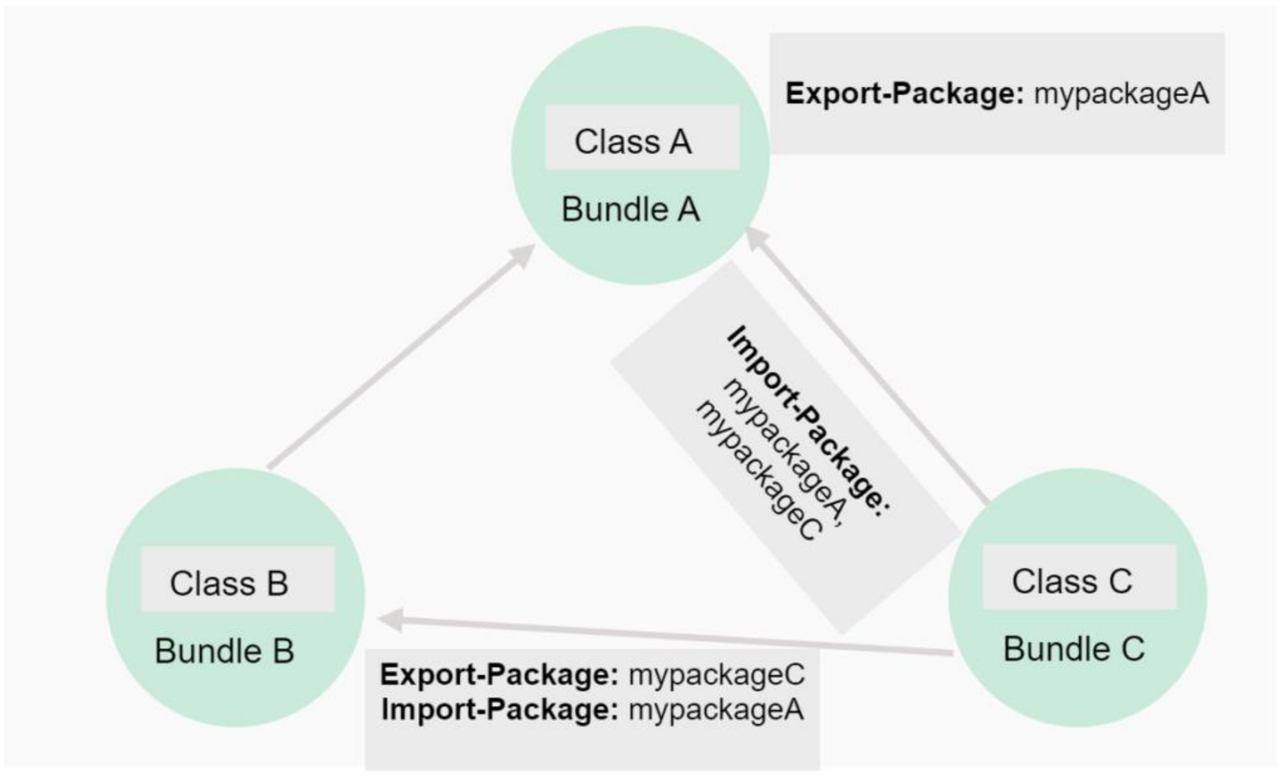

Export-Package: mypackageA
Class A
Bundle A
Import-Package: mypackageA, mypackageC
Class B
Bundle B
Class C
Bundle C
Export-Package: mypackageC
Import-Package: mypackageA

Figure 3.1

This is real for both kind of features we are considering in this project, which are Activities and URL-like features, which include URL or API calls. So, for example, in the case that a feature to inject is an Activity, PRISM will extract recursively all the classes referenced from the initial class, looking deeply inside the various class fields and variables and method invocation, gathering everything. On the other hand, if it is a URL-like feature, like an API invocation, internet call, etc., the invocation method is found and all the relative dependencies of the class containing it are extracted recursively.

Considering a relative simple *goodware* application, with few activities and classes the whole set of dependencies would consist of a total amount of 10-15 entities, while if the application is a huge one with a complex architecture scheme the final extracted gadget could easily reach 100 extra classes. For this reason, this evaluation part is critical for the correct operation of the framework, avoiding the chance to inject malicious code into a malware.

In order to evaluate the single gadgets and understanding their complex nature, we adopt a simple strategy which will be explained in detail in Section 3.2.5: the main idea is to create an empty apk, as template containing the minimum set of features possible, and using it for gadget evaluation. Every time we want to inject a gadget, we first try to inject it in the template; then evaluate the final result. This way we are sure that the final result will be in line with our goal, discarding all the malicious gadgets.

Another challenge by operating at problem space layer is how to correctly inject pieces of code without altering the correct operation of the original malware. It is necessary to identify a reasonable method for choosing the class in which injecting the code, otherwise we would risk to leave an evident trace of instrumentation, that could easily be detected by an analyser, either human or automatic. For this reason, we chose to adopt the Cyclomatic Complexity [15], which is used to indicate the logic complexity of a program. In **PRISM** we use it to retrieve a suitable list of classes, calculating the average CC of them and choosing one that fits our need. All this procedure will be explained in detail later in Section 3.2.3.

All these issues are not only limited at the feature layer, but they become a real problem when this kind of attack faces the reality at a problem space level. With this work we coped with all these issues and found an acceptable solution to them, in order to guarantee a deep correctness of all the actions taken for reaching a successful instrumentation.

Moreover, this changes the reference system of the attack, modifying also the constraints. For example, in usual adversarial attacks there is a constraint on the amount of perturbation applied at feature space level, but in our situation this limit cannot be applied, because we are in a total different reference space. Our main constraint is the size of the final application: we do not want to exceed a specific percentage of increase, in order to produce all the time a reasonable application

The main goal of this project is to create a successful framework which is able to correctly solve problem-space adversarial attacks on Android application, facing all the problems discussed before.

3.2 Infrastructure graph

The whole framework is composed by different modules that combine their own capabilities for reach the final goal. Due to the likely future use of this framework in S2LAB researches, we have decided to separate the roles in order to achieve better code modularity.

The infrastructure process is shown in Figure 3.2, containing all the different operational phases. The two main modules have been developed in the Java language, totally based on Soot framework, which has been already introduced in Section 2.2.4. We have mainly used its instrumentation capabilities, in order to detect, extract and reinject the needed code parts.

Moreover, before choosing Soot, we tried different other static analysis frameworks, like WALA and SAAF, but in the end they had not the expected characteristics (explained later in Section 5.6).

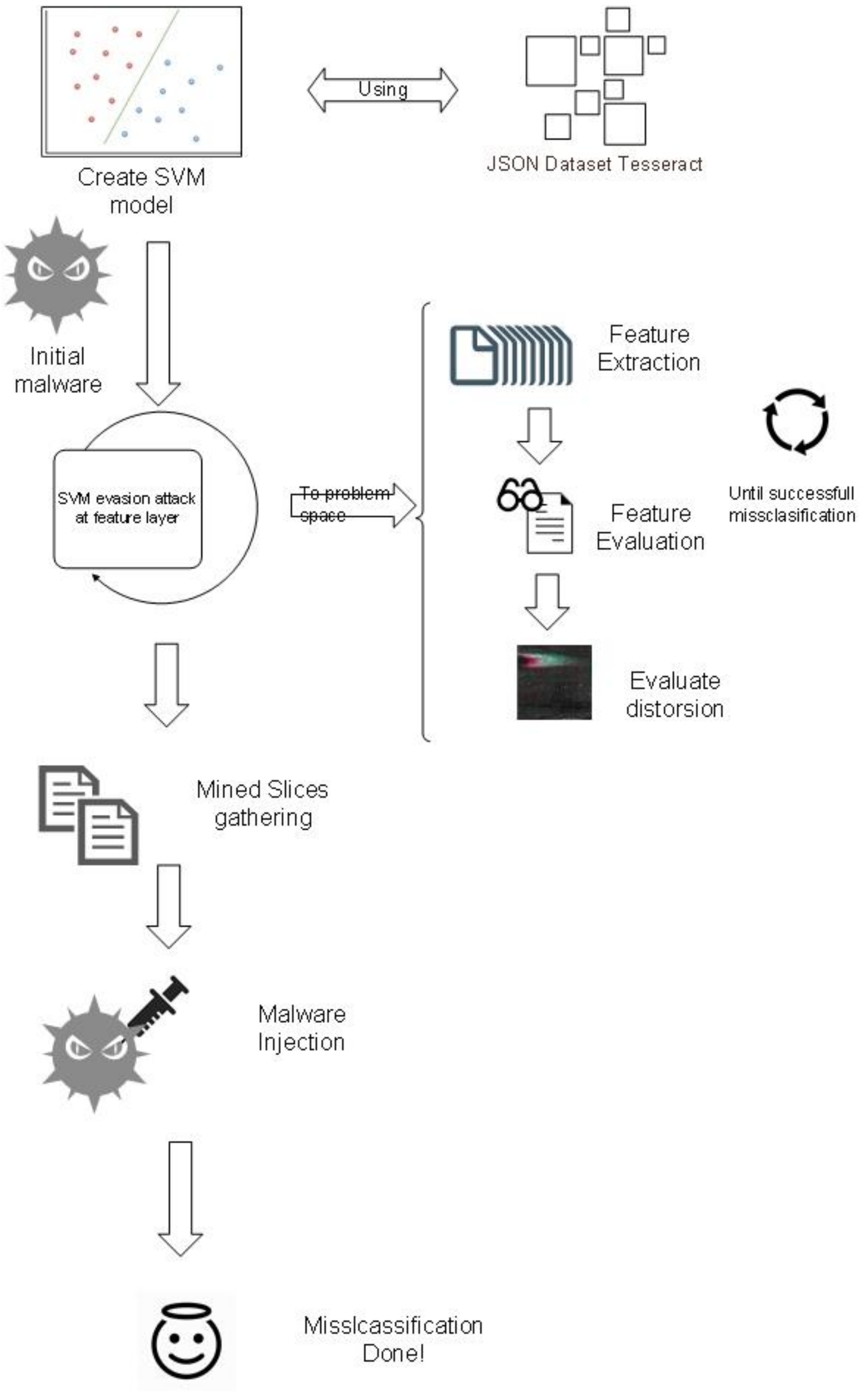
The two jar file modules are:

- The Extractor, which handles the extraction of the needed classes and dependencies from the benign applications, including also components, like Broadcast Receivers, Content Provider, etc.
- The Injector, which on the other hand handles the injection of all the set of dependencies inside the malware. A modified version of the injector is used also for the gadget evaluation part, in order to inject a specific gadget into the template apk, for verify if the whole set of features is benign or malicious

Another core module is the Drebin feature extractor, which is a faithful implementation of the Drebin extraction model written in Python. It basically is able to correctly extract all the features from a specific Android application following the Drebin model. This is possible by decompiling the target application and analysing the files contained in it. It makes also use of both smali/baksmali, which have been discussed before, and of AAPT (Android Asset Packaging Tool). AAPT takes an application resource files, such as the AndroidManifest.xml file and the XML files for the Activities, and compiles them. This is a great tool which helps to view, create, and update your APKs (as well as zip and jar files). It can also compile resources into binary assets. It is the base builder for Android applications. Using both tools, it can correctly match the presence of a certain features and the respective permission, by using an ad hoc file containing all the mappings between a specific call and the relative permission. It saves all the results into a JSON file, which contains all the features extracted. It has been extensively used into the PRISM framework in order to extract the features from a specific application during all the needed phases.

All these components are successfully coordinated by a Python program which contains the full logic of the framework: it includes the creation of the data model, the setup of the classifier and the invocation of the various modules described before. This orchestration script verifies that every step concludes correctly, making all the necessary checks on the single operations. It includes also the final application signature and the final evaluation to check whether the attack has ended successfully.

We tested this script on a huge dataset for the experiment part, so in order to achieve this we have coordinated the whole set of invocation by a multiprocessing strategy, also full implemented in python and run over the S2LAB cluster.



Create SVM model
Using
JSON Dataset Tesseract
Initial malware
SVM evasion attack at feature layer
~~To problem space~~
Feature Extraction
Feature Evaluation
Until successfull missclasification
Evaluate distorsion
Mined Slices gathering
Malware Injection
Misslcassification Done!

Figure 3.2

3.2.1 Classifier

As already mentioned before, the classifier we have adopted is a Linear SVM. Generally speaking, a Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labelled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In a two-dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

An example is represented by Figure 3.3. It fairly separates the two classes. Any point that is under the line falls into blue square class and everything above falls into red square class. The decision function is fully specified by a subset of training samples, the support vectors.

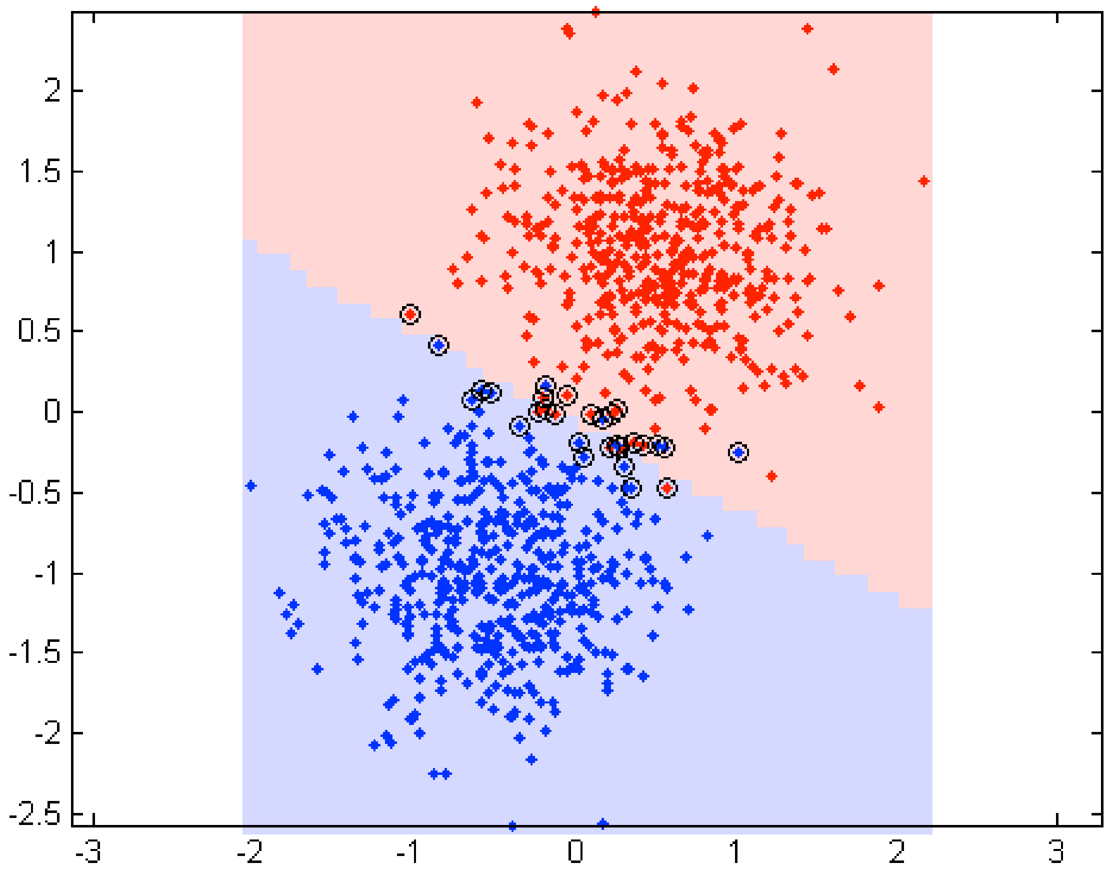


Figure 3.3

So mainly for a SVM:

- Input: set of (input, output) training pair samples; call the input sample features $x_1, x_2, \dots, x_n$ and the output result y. Typically, there can be lots of input features x_i .
- Output: set of weights w (or w_i), one for each feature, whose linear combination predicts the value of y.

Moreover, each SVM classifier has tuning parameters, like the regularization parameter, gamma or kernel, which defines whether we want a linear of linear separation.

The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra. This is where the kernel plays role. For linear kernel the equation for prediction for a new input using the dot product between the input (x) and each support vector (xi) is calculated as follows:

$$f(x) = B(0) + sum(a_i * (x, x_i))$$

This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients B0 and ai (for each input) must be estimated from the training data by the learning algorithm.

The Regularization parameter (often termed as C parameter in Python's sklearn library) tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points. The gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. In other words, with low gamma, points far away from plausible separation line are considered in calculation for the separation line. Whereas high gamma means the points close to plausible line are considered in calculation.

And finally, the last but very important characteristic of SVM classifier is the margin. SVM to core tries to achieve a good margin, which is a separation of line to the closest class points. A good margin is one where this separation is larger for both the classes. The image below, Fig. 3.4, gives a visual example of good and bad margin. A good margin allows the points to be in their respective classes without crossing to other class.

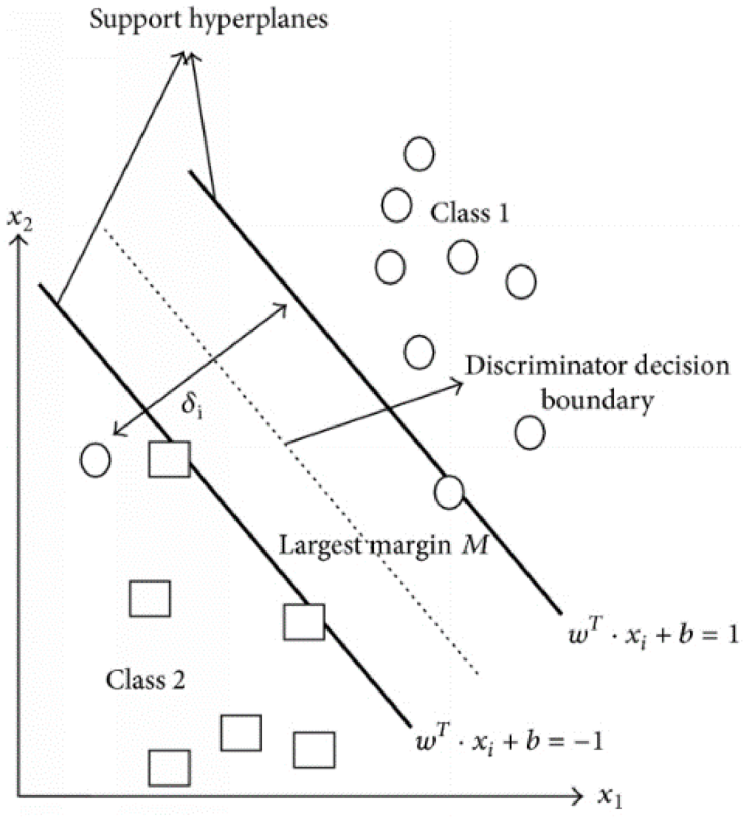


Figure 3.4

The advantages of the SVM technique can be summarised as follows:

- SVMs provide a good out-of-sample generalization, if the parameter C is appropriately chosen. This means that, by choosing an appropriate generalization grade, SVMs can be robust, even when the training sample has some bias.

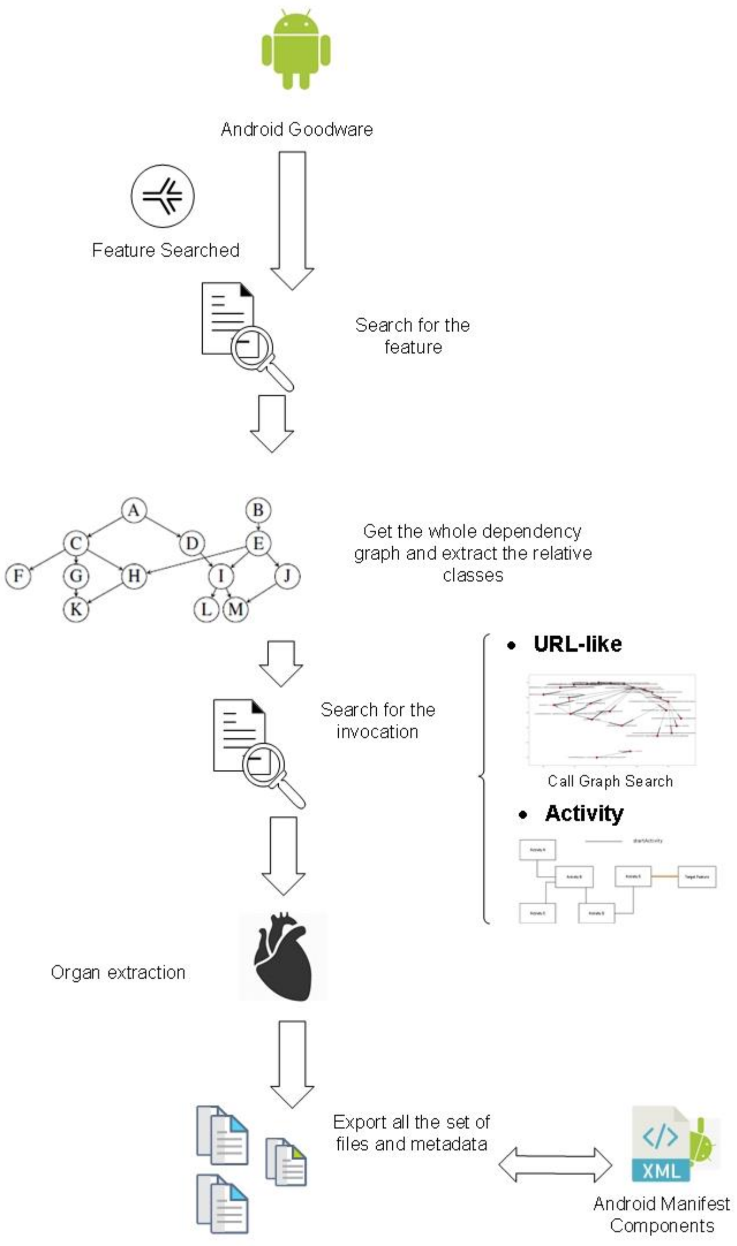
- SVMs deliver a unique solution, since the optimality problem is convex. This is an advantage compared to Neural Networks, which have multiple solutions associated with local minima and for this reason may not be robust over different samples.
- With the choice of an appropriate kernel one can put more stress on the similarity between elements, because the more similar the element structure of two data is, the higher is the value of the kernel.

On the other hand, the disadvantages are that the theory only really covers the determination of the parameters for a given value of the regularisation and kernel parameters and choice of kernel. In a way the SVM moves the problem of over-fitting from optimising the parameters to model selection. Sadly, kernel models can be quite sensitive to over-fitting the model selection criterion. Please note, however, that this problem is not unique to kernel methods, most machine learning methods have similar problems. The hinge loss used in the SVM results in sparsity. However, often the optimal choice of kernel and regularisation parameters means you end up with all data being support vectors.

The Linear SVM classifier is suitable for what we need because we only need to classify a target application as malware or *goodware*, which is represented as a binary problem. We do not have any multiclass classification need, so the Linear SVM fits perfectly. Due to the risk of overfitting we have trained the classifier on a large-scale dataset, already used into other previous experiment done by the S2LAB team during other researches on malware classification with machine learning.

3.2.2 Extractor

The Extractor module focuses on extracting all the set of necessaries dependencies from a specific *goodware*. This represents the first step after the identification of the set of the needed features to transplant and the correspondence hash of the application that contains it. The whole module has been built on top of Flowdroid [16], which is built on Soot framework. The module has been designed in order to being able to successfully extract Activity features and URL-like ones, like URLs and



Android Goodware
Feature Searched
Search for the feature
A
B
C
D
E
F
G
H
I
J
K
L
M
Get the whole dependency graph and extract the relative classes
Search for the invocation
URL-like
Call Graph Search
Activity
Organ extraction
Export all the set of files and metadata
XML
Android Manifest Components

Figure 3.5

API calls. Of course, the two extraction methods are slightly different. The process graph is represented in Figure 3.5. Getting into the details on how the extraction process works:

- For the Activity features, the Extractor module at first searches if the searched feature is inside the target application. Indeed, there are cases in which the activities are imported from some C/C++ libraries [17] and unfortunately our analysis tool is not able to catch them. After having identified the target Activity, it analyses the whole target body class and extract all the dependencies needed using a PDG. These include field types, interfaces and other external classes needed for the correct operation of the target class. The next step consists in trying to extract a slice containing the invocation of the current focused Activity. This is particularly complicated for complex applications. Our approach can be compared to a greedy algorithm: we analyse the body of all the methods of all the classes in the current `Scene`, see Section 2.2, searching for the presence of the `startActivity` method invocation and of the declaration of our target Activity. If the module finds a match, then the Extractor tries to extract the Activity invocation leveraging on the CFG, getting the set of basic blocks that contain the invocation. Of course, we need to be sure that the extracted set is independent and for this reason we double check if it contains all the needed dependencies to correctly work. In case it doesn't, we try to extract also the set of missing dependencies and add them to the final `Scene`. In case we do not find any match, the extraction part stops just after having found all the set of dependencies of the target activity.

- On the other hand, for the URL-like features, we decided to change the last part of the extraction process: if an Activity is a pure Android component and the extraction is a pretty straightforward operation, for a generic URL-like string the whole operation gets more articulated. In order to make it robust enough to be generalised we decided to focusing on extracting the callee method that invokes the method containing the feature we are searching for. For example, if we have a method M0 which contains the feature F0 in class C0, we are searching for a method M1 of another class C1 which calls M0, emulating the same *goodware* operation.

In order to make this approach successful, we adopted the following methodology: at first we search in the current application for the presence of the searched feature, identifying the class which contains it. That will represent our extraction seed, from which the whole extraction procedure starts, identifying also the method of that class that contains the searched feature. From that point, we will add all the set of dependencies recursively, guaranteeing the correct operation of the extracted class, with the same methodology adopted for Activities features type, using the modified PDG.
After this first phase, we scan the Call Graph of the whole application, searching for an invocation to the method containing the searched feature recursively. In order to not introduce too much overhead during the whole process we have decided to set a custom threshold which indicates the maximum level of recusivity of the greedy search algorithm, otherwise it could have brought to a timeless execution. If everything works fine, the module has been able to identify the method and the class that in the benign application are used to invoke the feature we are searching for and we are then able to replicate the same mechanism into the malware. After that we try to extract the method invocation in a pretty similar way to the Activity features: we use the CFG to identify the necessary basic blocks for build an independent slice of code containing the needed invocation. During the whole process all the dependencies are recursively found and added to the final output, in order to guarantee an independent set of entities. If the slice extraction ends successfully, the whole process is terminated, returning a positive result, and the module execution ends. Otherwise, it retries on other eligible classes if during the identification process more than a class has been matched, otherwise just return an error and exit.

During both processes the module creates some basic metadata files which are necessary for the correct execution of the Injector module.

3.2.3 Injector

The Injector module handles all the injection part, importing all the needed entities to inject and injecting those into the target

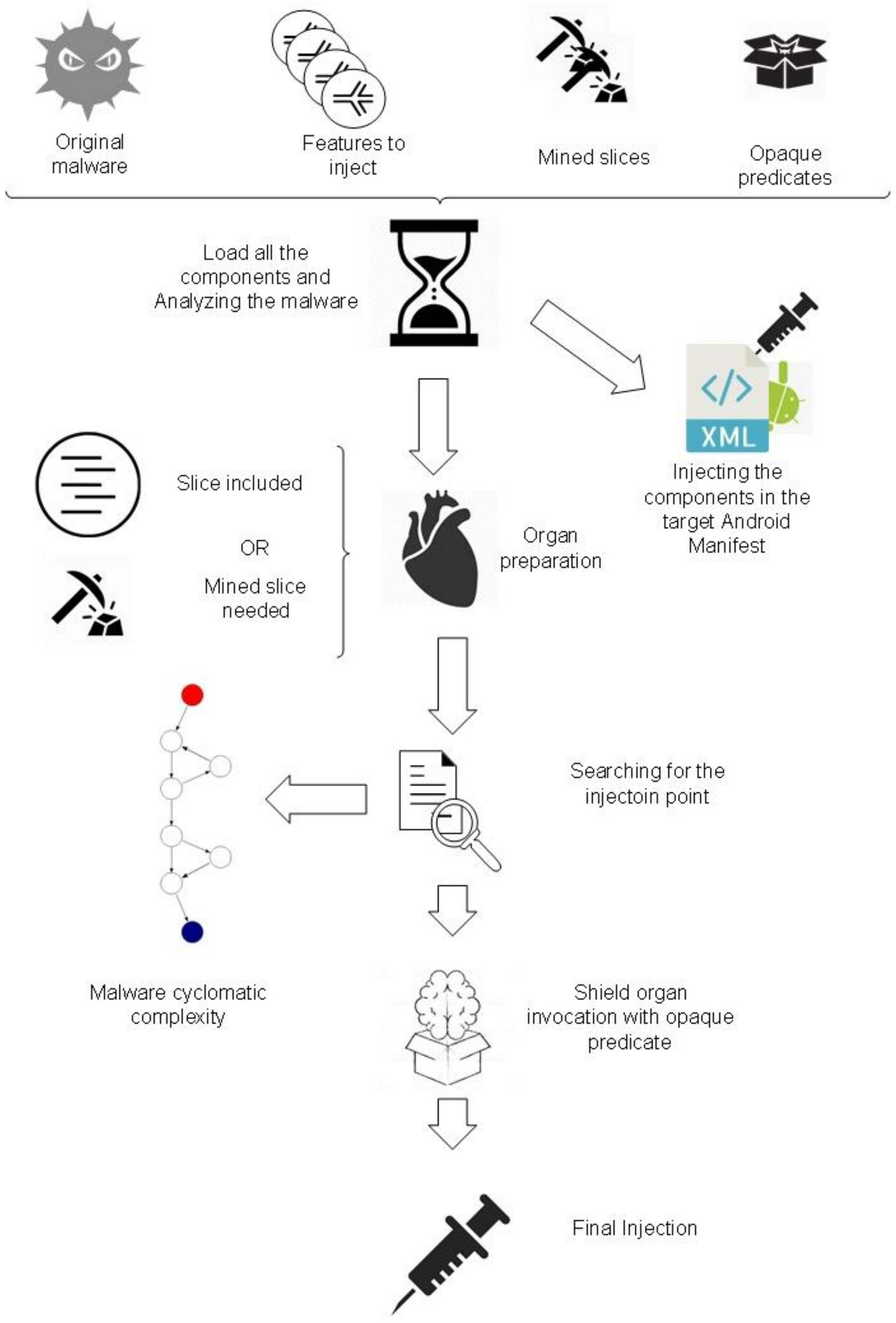

Original malware
Features to inject
Mined slices
Opaque predicates
Load all the components and Analyzing the malware
Injecting the components in the target Android Manifest
Slice included
OR
Mined slice needed
Organ preparation
Searching for the injectoin point
Malware cyclomatic complexity
Shield organ invocation with opaque predicate
Final Injection

Figure 3.6

malware. As the Extractor module this is also fully implemented on top of FlowDroid [16], granting a huge instrumentation flexibility. The whole operation process is represented in Figure 3.6, showing the various entities and operation that are part of the whole module.

Due to being robust to a larger case set, we decided to implement the injection using the opaque predicates, specifically implementing the 3SAT type of them. Generally speaking, an opaque predicate is a predicate—an expression that evaluates to either "true" or "false"—for which the outcome is known by the programmer *a priori*, but which, for a variety of reasons, still needs to be evaluated at run time. Opaque predicates have been used as watermarks, as it will be identifiable in a program's executable.

For this reason, we use the opaque predicates to protect our injection operation against static dead code elimination techniques. Indeed, our main goal is not to change the semantic equivalence of the malware and in order to obtain it we need to be sure that the code we are going to inject won't be executed in any case. And here is where the opaque predicates play their role: using this kind of construct is possible to obfuscate the CFG of that instruction statically speaking, forcing an hypothetic analyser to dynamically analyse the application to figure out if that specific branch is going to be executed or not.

Since we do not need any complex obfuscation, we adopted the 3SAT construction strategy deeply explained in this paper [18]. In a nutshell, the idea of the following opaque constant is that we encode the instance of an NP-hard problem into a code sequence that calculates our desired constant. That is, we create an opaque constant such that the generation of an algorithm to precisely determine the result of the code sequence would be equivalent to finding an algorithm to solve an NP-hard problem. For our primitive, we have chosen the 3-satisfiability problem, explained as here [19] as a problem that is known to be hard to solve. The 3SAT problem is a decision problem where a formula in Boolean logic is given in the following form:

$$\wedge_{i=1}^{n} (V_{i1} \vee V_{i2} \vee V_{i3})$$

where $V_{ij} \in \{v_1, ..., v_m\}$ and $v_1, ..., v_m$ are Boolean variables whose value can be either true or false. The task is now to determine if there exists an assignment for the variables v_k such that the given formula is satisfied (i.e., the formula evaluates to true). This problem in NP-Hard to solve, guaranteeing overhead for a generic static solver. Moreover, we though that having an invocation for the feature searched is fundamental for bypassing by construction dead code elimination features.

We have also implemented the functionality that adapts extracted code sliced to different features, in order to use features from which was not possible to find an eligible slice. For this reason, we created then a slice database folder in which are contained all the benign invocation that could be reused for future iterations. This situation could occur for example in the case in which the feature to inject is a Main Activity and for this reason there is no explicit invocation inside the Android application, because it is automatically launched by the intent filters in the manifest. We made the module also robust to this kind of situation, loading some external invocation slices and modifying the referenced class, by instrumenting it with Soot.

Getting back to the general module operation, the first thing this module does is to collect all the necessary entities to inject into the malware, analysing them and, if necessary, adding the Android components to the AndroidManifest of the target malware. This step is crucial to correctly misclassify the malware, because the Manifest file is the biggest source of metadata of any Android application and it is the first file analysed to understand the current application structure. The whole injection must be clean and the final Manifest must not include any duplicate or malformed tag, in order to not seem instrumented. It is able to inject all the Android components and their relative tags, which we want to import as well due to a matter of correctness.

Then the module proceeds identifying the needed invocation slice to inject, searching if it is possible to extract it directly from the original *goodware*. In case it is not possible, as already mentioned, it randomly choose one of the available *'mined slices'* in order to shape it and adapt it to the current invocation we need.

The last and crucial step is selecting in which malware class inject the invocations of the various features. In order not to introduce any instrumentation evidence, we decided to adopt the Cyclomatic Complexity (CC) score as meter in order to select a set of eligible classes. The CC of a section of source code is the number of linearly independent paths within it. For instance, if the source code contains no control flow statements (conditionals or decision points), the complexity would be 1, since there would be only a single path through the code. If the code has one single-condition IF statement, there would be two paths through the code: one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE, so the complexity would be 2. Two nested single-condition IFs, or one IF with two conditions, would produce a complexity of 3.

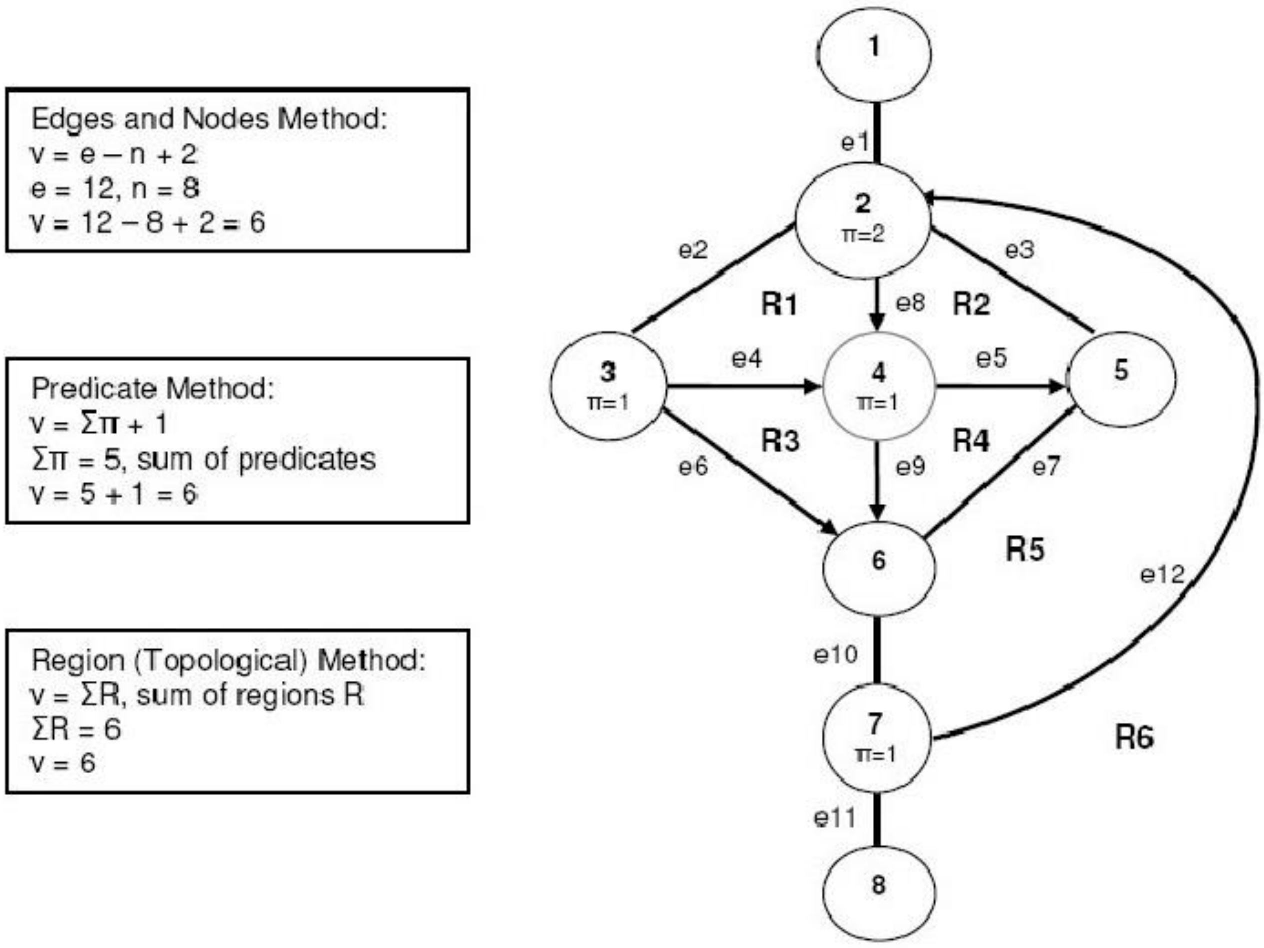
Mathematically, the CC of a structured program is defined with reference to the CFG of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. For a single program (or subroutine or method), P is always equal to 1. So, a simpler formula for a single subroutine is:

$$M = E - N + 2$$

where:

- *E = the number of edges of the graph.*
- N = the number of nodes of the graph.

An example of different ways to calculate CC is shown in Figure 3.7, and, as it is clear, all of them bring to the same result. We adopted the above formula because we have access to the CFG, so we are free to iterate between Nodes and Edges.


Edges and Nodes Method:
$v = e - n + 2$
$e = 12, n = 8$
$v = 12 - 8 + 2 = 6$

Predicate Method:
$v = \Sigma\pi + 1$
$\Sigma\pi = 5$, sum of predicates
$v = 5 + 1 = 6$

Region (Topological) Method:
$v = \Sigma R$, sum of regions R
$\Sigma R = 6$
$v = 6$

Figure 3.7

After having identified all the malware classes, we calculate the average CC of the whole malware program, and we find those classes that have a CC score that satisfies the following equation:

$$c + \delta = A \pm \Delta$$

where c is the CC of the injected class, ∂ is the whole slice CC, A is the CC average of the malware and Δ is the current adaptation level, allowing some kind of flexibility. This is done in order not to create any CC score spike in certain classes, which could lead to a strong signal of instrumentation from a hypothetic analyser.

The module then tries to inject the invocation in all the eligible classes, until it can successfully inject in one. We decided to randomly select the starting point inside the method body in which we are going to inject the slice, in order not to introduce any constant mechanism in our injection mechanism, with the goal to make it even harder for an analyser to understand some kind of pattern. As already mentioned, the final injected slice is composed by the extracted invocation shielded into one of the available opaque predicates, shielding it.

3.2.4 Slices evaluation

As introduced before, during the experiments we noticed that each feature extraction inevitably drags with itself some other side-effect features because of all the dependencies identified and extracted. For this reason, we need to evaluate them, in order to weight their contribution to the whole gadget score. The whole process is described in Figure 3.8.

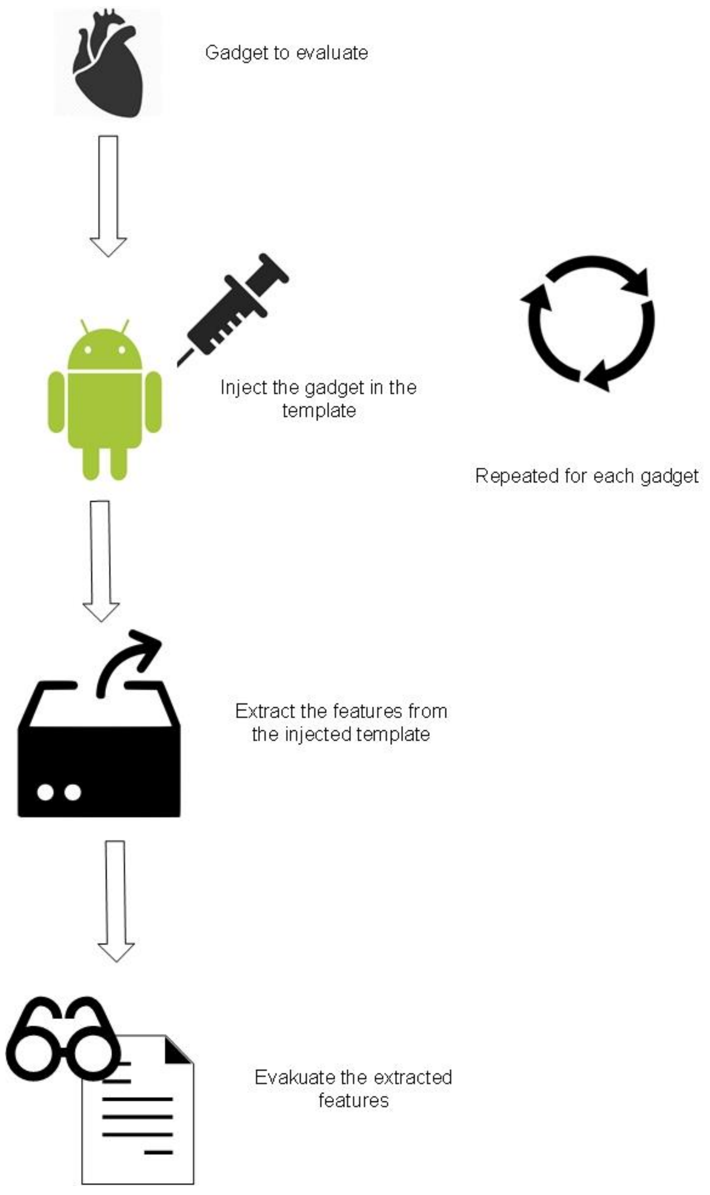

Gadget to evaluate
Inject the gadget in the template
Repeated for each gadget
Extract the features from the injected template
Evakuate the extracted features

Figure 3.8

This demonstrates that injecting a feature at problem space level is not trivial as it seems because it is necessary to consider also all the set of external entities from which a specific feature depends on. Indeed, a single entity could rely on a huge number of external resources, building a large dependency chain and adding a large set of side effect classes. From the final score point of view there are two possible scenarios: the external dependencies strengthen the whole benign score or it reduces it. For this reason we created a separated module which is a modified version of the Injector one that simply focuses on injecting a particular gadget set into an ad-hoc Android application, without using any mined slice. For this set of operations, we created a specific basic Android application that we used as template. This template application consist into an empty Android app that does not contains any particular component, just including the basic elements for make an application suitable for instrumentation.
Indeed, it has no role except that verifying the nature of the gadgets we are going to work with. The first important thing is verifying that no particular feature is removed during the injection, which could compromise the initial malware workflow and operation.

After this, we verify which are the added features and which is the final gadget score. During this operation we also check if the permissions required by the gadget are already included in the malware. Indeed, we have decided to put a custom threshold T in order to avoid the injection of an excessive number of permissions. Even if we need to extend the feature injection also to them due to attack secrecy, application permissions are one of the core points of Android malware analysis, so we need to be discrete. For this we decided to hold *goodware* gadgets only if we are not forced to add more than T further permissions, letting that malware part slightly modified.

This phase is crucial in order to understand the effective and real impact that a single feature has on the target malware, considering the whole gadget set.

4 PRISM realization

In this section we will present the technical details and the choices made for implementing **PRISM**. Indeed, after having provided the general idea of the behaviour of each component during the previous section, in this one we will discuss the implementation strategies adopted and the reasoning behind them, due to the problem faced.

In this section we will explain the transition between the theoretical attacks explained in Section 3 to their concrete implementation, using also a Minimum Working Example (MWE) in order to show clearly how each single module operates. We will use a complete example, composed by a single malware and a set of features needed to being injected in order to misclassify it.

We will cover in details all the modules of the framework, starting with the classifier used, in particular focusing on the dataset adopted and the scores obtained. Then we will present in details the Extractor java module, starting from the implementation overview, followed by an example both for Activity and URL-like feature. As following then the Injector module and how the whole injection is computed, with particular focus on how we achieve the safeguard of the semantic equivalence of the instrumented malware. The whole explanations will be integrated with parts of the final orchestrator script, in order to give to every module the right context in the whole execution flow.

Moreover, due to the will to test its robustness in large scale scenario we tested the current framework on a large dataset of Android applications, thanks to the resources of S2LAB. Due to this, the whole framework has been built with the purpose of correctly handle also multiprocessing scenarios, avoiding anomalies. An example of this prevention measure is that the framework creates for each malware run a temporary environment, avoiding in this way to have shared resources.

4.1 Classification and Dataset

The classifier used is a Linear SVM and in order to train and test it we have used the Python language. In particular we have adopted the Scikit-lean library set, which is a simple and efficient set of tools for machine learning and data analysis [20].

The whole classifier has been trained on the same dataset used for Tesseract [21], which is composed by a temporal slice extracted from the AndroZoo [22] dataset. Indeed, this dataset contains more than 5.8 million application between 2010 and 2018, including a timestamp and, until the apps of 2016, also a VirusTotal metadata results. The dataset is constantly updated by crawling from different markets (e.g., more than 4 million apps from Google Play Store, and the remaining from markets such as Anzhi and AppChina). In order to select malware and *goodware* from this dataset we rely on the VirusTotal metadata entry, considering a specific application a malware if there are at least 4 entries into the relative metadata section, indicating the number of anti-virus report that classify that app as malicious. The number 4 has been adopted due to past researches by Miller [23]. The opposite is valid for the *goodware*: we classify as benign all the apps with less than 4 positive anti-virus reports.

We choose to use a slice of the available dataset, precisely only the three years between the 2014 and the 2016. Getting more into the details, we subdivided this temporal slice dataset in a way that the applications between 2014 and 2015 compose the training dataset, while the ones belonging to 2016 year represent the test dataset. This way we could also observe also the time decay effect on the whole attack. Training the classifier with these settings we are able to obtain the following scores:

F1 score	0.91749
Precision	0.90756
Recall	0.92764

Fig 4.1

Precision is the fraction of relevant instances among the retrieved instances, while recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. On the other hand, the F1 score indicates the accuracy of a specific test, considering both precision and recall and can be expressed as following:

$$2 * \frac{precision * recall}{precision + recall}$$

The classifier is then able to recognize if a specific Android application is a malware or not depending on the features contained in it. Indeed, the whole dataset can be represented as a huge matrix containing as rows the whole set of considered features and as columns the application analysed, as in Figure 4,2

	App $_{i+1}$	App $_{i+2}$	App $_{i+3}$	App $_{i+5}$	App $_{+6i}$	App $_{n}$
Feat $_{i}$	0	1	1	0	1	1
Feat $_{i+1}$	1	1	0	0	1	0
Feat $_{i+2}$ …	0	1	1	1	0	0
Feat $_{n}$	1	1	1	0	1	0

Figure 4.2

In order to correctly classify a malware with the classifier, we need the feature extractor module in order to extract the features contained in a single application. The final output, which is a JSON file, can be then loaded in memory and formatted to create a similar vector to the one showed in Figure 4.2. This way we can represent an arbitrary application by its own feature vector, which is understandable for the classifier, which will output the supposed classification category.

If the classification is then successful and the application is identified as a malware we will start the feature layer attack introduced in Section 3.1. Specifically, the attack can be explained with the following algorithm:

```
$\{w_1', w_i', \dots, w_n'\} = sort(\{w_1, w_i, \dots, w_n\})$
For $w_j$ in $\{w_i', w_{i+1}', \dots, w_k'\}$:
    If $w_j < 0$:
        M.add($w_j$)
    If $|\sum_k^l w_k| > \beta$:
        Stop
```

Algorithm 1

where $\{w_1, w_i, \dots, w_n\}$ are the weights of the features of the classifier, *sort()* is a function that sorts by absolute value the given input vector, and then we add to a malware **M** all the benign features (weight less than 0) of the considered feature vector. $|\sum_k^l w_k|$ represents the sum in absolute value of the weights considered until now, while β is the minimum perturbation needed for the misclassification. This way we are adding to the initial malware the minimum feature set that generates enough perturbation for crossing the hyperplane boundary. An example is shown in Figure 4.3, in which se red square become a green one, crossing the boundary. We transfer the injected malware from the malicious hyperplane to the benign one, operating the misclassification. When the boundary has been crossed, the attack is considered successful and ends, returning a set containing the features added, which represents the minimum set to inject in order to misclassify **M**.

After this step, the whole pipeline will continue searching for the best candidate apps for each feature contained in the extracted vector, trying to extract the necessaries entities from the selected *goodware*. In case that there are multiple donors, the framework tries to extract the best option between them. This whole phase can be looped in case during the evaluation phase the benign score of the organs is reduced, do the other side-effect features.

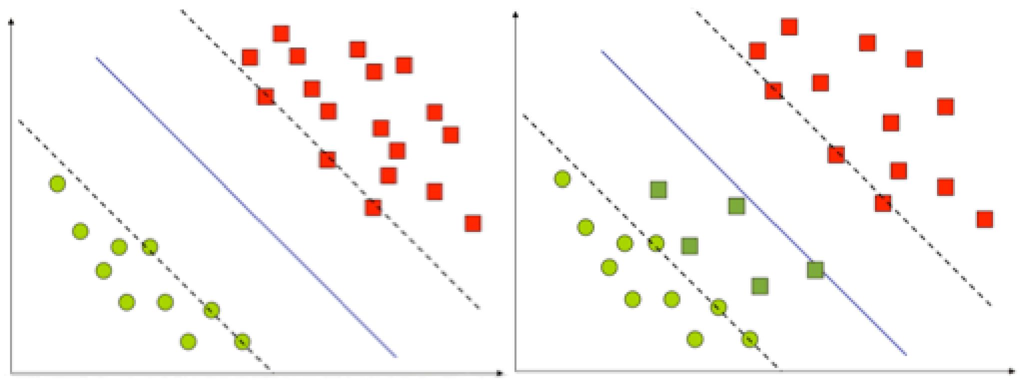


Figure 4.3

4.1.1 PoC

We will now introduce a malware application as example during the whole current Section. We will indicate this malicious application as M^T. So, taking the example of M^T it has been successfully identified as malware from the classifier C, which gave as output the malware class as prediction with a correspondent total weight of 2.69, which correspond toβ. This means that the minimum perturbation needed for misclassify $M^T$is β.

After this step, we launch the described attack on M^T, which identifies two features to add in order to evade C, more being precise two activities, *cxim.qngg.Tehr.sFiQa* and *.guessidiom*. At feature space level, injecting those two features is enough in order to evade the target classifier, so the framework use both of them as seed in order to extract the whole dependency set from the *goodware* containing them. The whole gadget has then to be evaluated in order to weight also the contribution of the side-effect features.

4.2 Extraction

After having identified the set of *goodware* that we need for the attack, the next phase is extracting the needed features from them. As already underlined previously, the core operation is the extraction of the full dependency chain of the interested code part, including static fields, superclasses, methods, variables, etc.

In order to avoid any kind of conflict directly at design level, every single process, starting from this phase, creates a temporary environment in which it operates, preventing any type of file-sharing issue. Indeed, we create a temporary folder and copy all the needed files for the current run, making a dedicated file for each single malware we operate on, even if it means a higher consumption of resources.

The whole extraction phase consists in calling the Extractor module, already presented at high level at Section 3.2.2. After the necessary arguments are computed by the whole orchestration script, this module handles the entire extraction of the needed entities, including the invocation code slice. Moreover, it is also robust to the lack of an invocation section for the Activity features, thanks to the *'mined slice'* functionality of PRISM. This allows to increase the eligibility of the tool in different kind of scenario.

The main extraction strategy is the following, despite the difference between the URL-like and Activity: using a PDG, introduced in Section 2.2, we are able to gather all the needed dependencies, which is a functionality provided by Soot. Because its PDG implementation had some limits, we integrated the functionalities provided. Our implementation of PDG is indeed able to inspect interface dependencies, superclasses, static fields and other classes called, retrieved by inspecting the value of the variables passed. This way in the final malware is possible to guarantee that there are no dead references to other ghost components. If the feature is an Activity, the whole dependency extraction can be summarized in this way. Due to the inclusion inside an Android device of the Android library classes, this analysis automatically excludes from the scope all the native libraries, such as Android ones or Java.

On the other hand, if the feature is a URL-like features, like an API call or a general URL, the whole process is different, as explained in Section 3.2.2. Indeed, as first we need to identify which is the class containing the feature searched and which kind of entity it is. In order to achieve this, we use a parser that inspect the whole body of all the classes in the current `Scene` searching for that specific String. Once it identifies the class containing the it, the same process using the PDG explain previously is invoked, extracting the whole chain of needed java entities.

While for an Activity feature the next step would have been the exclusion of all the entities not needed from the `Scene` to export, for a URL-like feature there are still few more steps. Indeed, after having identified the class containing the URL feature, we need to figure out in which method M it is currently used, in order to extract that invocation. In order to do this, we use again a parser, which scans all the class static fields and methods searching for the target feature. For the matter of robustness, we decide to extract all the possible methods and iterate between them until we can find the right seed.

Indeed a method represent the initial node from which start the backward search into the Call Graph of the current *goodware*. As a matter of fact, the module tries to find the method M' calling our target method M, in order to identify a subset of eligible entities for extract the code slice containing the invocation of M. The whole research needs to be recursive because it could happen that some operations are called from the class itself or during the creation phase (constructor), so we want to understand which entity triggers the whole set of operations.

Moreover some features could be declared as static fields, so the whole module needs to be robust even for that kind of declaration. In that case the smali bytecote representation provides the code with a <clinit> method, which is a static constructor in case the class has some static fields and it is called every time that there is a class initialization. So basically, it is invoked during every method invocation that uses any of the declared static variables.

After these phases, we are able to define our set of target entities to extract, so we proceed by excluding all the classes that do not belong to the output dependency set from the current `Scene`. From this set the only missing entity that we miss in order to have a complete extraction is the invocation slice, which is a `startActivity()` method invocation for the Activity feature and a general method invocation for the URL-like features.

The whole slicing process can be seen as an execution of a backward slice of the target method. Indeed after we have discovered the class containing the searched feature, the slicer is able to identify the statement that contains the target invocation. It tries then to gather all the needed previous statements in order to create an autonomous code section that can be freely exported: a slice. In order to achieve this, we have mainly used the CFG representation of the method, considering also the set of data dependencies that they need. The CFG representation that we used is the most reliable block one that Soot provides in term of control flow analysis: the ExceptionalBlockGrap [24]. This class includes edges from throw clauses to their handler (catch block, referred to in Soot as Trap) and also takes into account exceptions that might be implicitly thrown by the VM. For every Unit that might throw an implicit exception, there will be an edge from each of that units predecessors to the respective trap handler's first unit. Furthermore, should the excepting unit contain side effects an edge will also be added from it to the trap handler.

Then we deeply inspect the Jimple IR of the seed block, which is the one containing the invocation and tries to reconstruct the chain of needed blocks in terms of dependencies. We focus on literally select all the needed statements in order to make the target autonomous and exportable, extracting all the variables needed, even the static ones. In the end we create a subset of the CFG composed by the minimum set of basic blocks that contains the entire set of dependencies of the method invocation. During this process we need to make sure not to include any kind of ghost reference, so after the initial export we double check the output for removing those.

In case there are multiple slices available, we choose the less complex in term of dependencies and procedures. Indeed, we estimate the current complexity of a slice with the following formula :

$$\gamma + \delta + CC$$

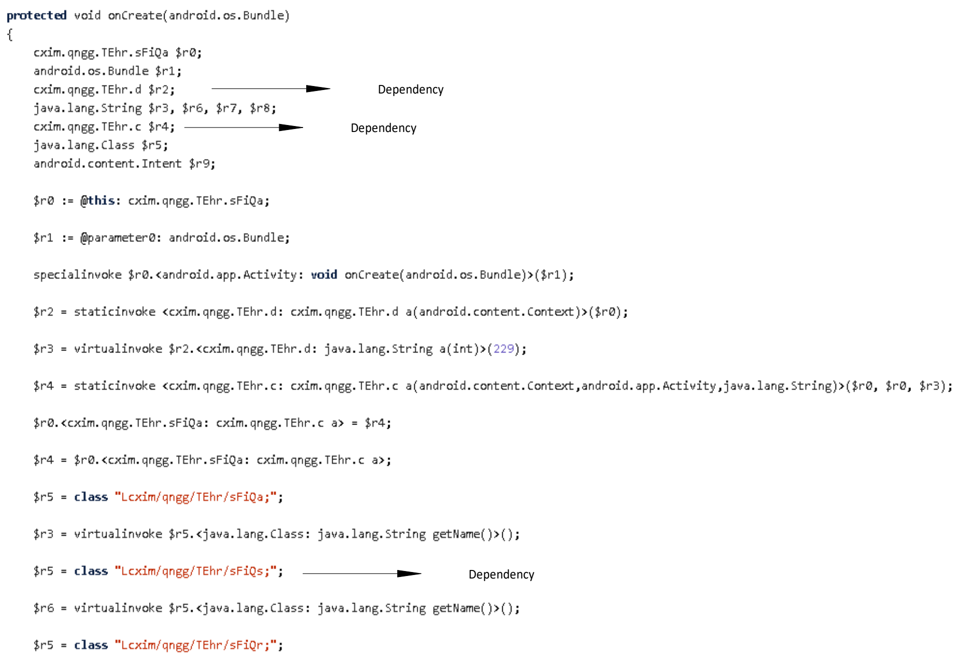
in which γ is the number of complex objects contained into the slice, δ is the number of variables and CC is the Cyclomatic Complexity of the slice considered.

The extracted slice is then stored into an ad-hoc Java object, which defines some basic utilities for operate on the slice. Indeed, in order to export the slice as a Jimple file, which is the format in which we store the gadgets, we need to create a temporary class due to Soot conformity. In this class we store all the statements composing the slice, which is stored as body of the only method of the class. It will be then loaded back from the Injector module in the next phase, which is going to be discussed in Section 4.4.

In case of success, the extracted slice is moved into the folder containing the mined slices, which are used in case it is not possible to extract any slice from certain *goodware*. Indeed, in the case these pre-made slices are used, the Injector tries to modify the needed statements to adapt the slice for the current feature. This is going to be deeply explained in Section 4.4.

4.2.1 PoC

Getting back to our example M^T, both of the features selected are Activity and the Extractor module will be called twice. Due to the fact that both of them are the same component, We will just take as example the iteration

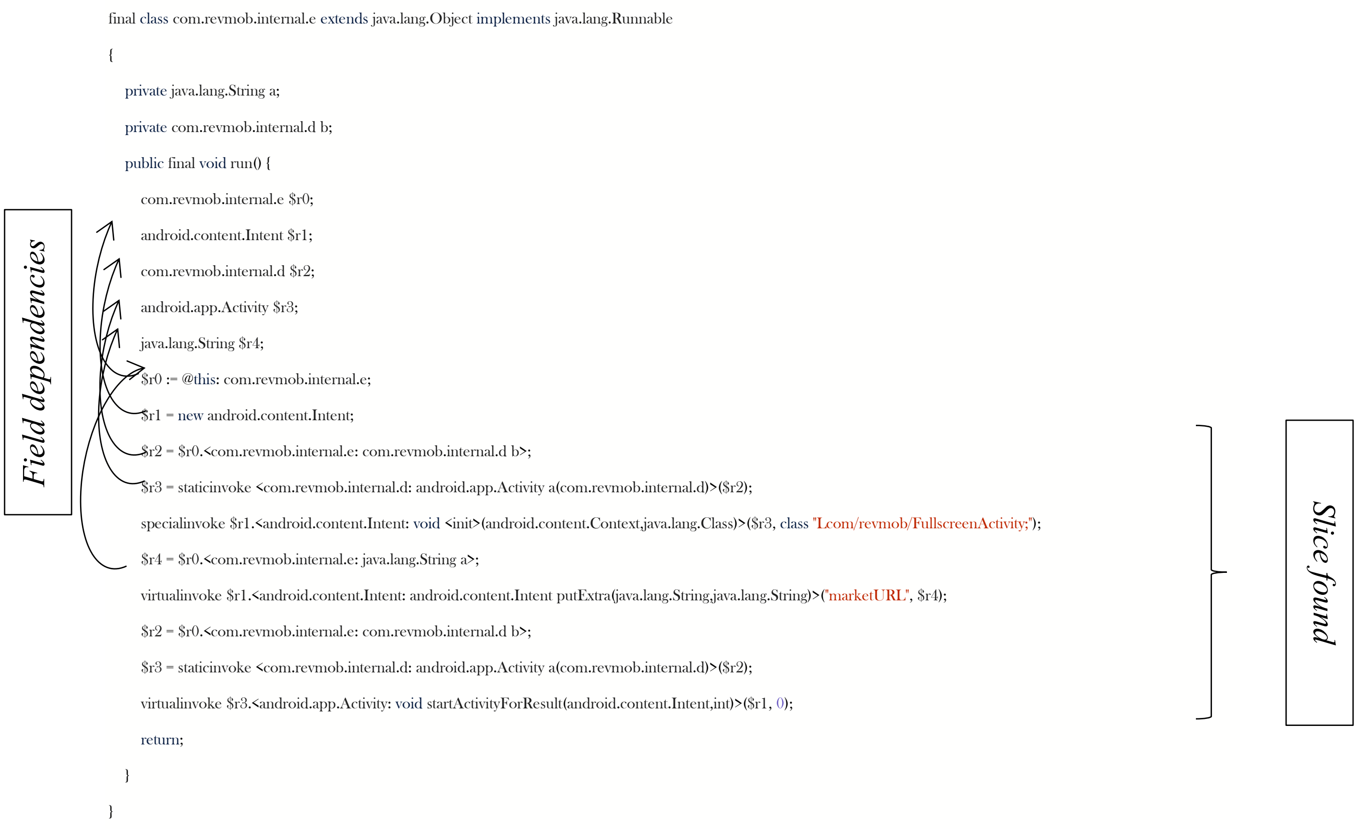


```
protected void onCreate(android.os.Bundle)
{
    cxim.qngg.TEhr.sFiQa $r0;
    android.os.Bundle $r1;
    cxim.qngg.TEhr.d $r2;            ——→          Dependency
    java.lang.String $r3, $r6, $r7, $r8;
    cxim.qngg.TEhr.c $r4;   ——→          Dependency
    java.lang.Class $r5;
    android.content.Intent $r9;

    $r0 := @this: cxim.qngg.TEhr.sFiQa;

    $r1 := @parameter0: android.os.Bundle;

    specialinvoke $r0.<android.app.Activity: void onCreate(android.os.Bundle)>($r1);

    $r2 = staticinvoke <cxim.qngg.TEhr.d: cxim.qngg.TEhr.d a(android.content.Context)>($r0);

    $r3 = virtualinvoke $r2.<cxim.qngg.TEhr.d: java.lang.String a(int)>(229);

    $r4 = staticinvoke <cxim.qngg.TEhr.c: cxim.qngg.TEhr.c a(android.content.Context,android.app.Activity,java.lang.String)>($r0, $r0, $r3);

    $r0.<cxim.qngg.TEhr.sFiQa: cxim.qngg.TEhr.c a> = $r4;

    $r4 = $r0.<cxim.qngg.TEhr.sFiQa: cxim.qngg.TEhr.c a>;

    $r5 = class "Lcxim/qngg/TEhr/sFiQa;";

    $r3 = virtualinvoke $r5.<java.lang.Class: java.lang.String getName()>();

    $r5 = class "Lcxim/qngg/TEhr/sFiQs;"    ——→          Dependency

    $r6 = virtualinvoke $r5.<java.lang.Class: java.lang.String getName()>();

    $r5 = class "Lcxim/qngg/TEhr/sFiQr;";
```

Figure 4.4

of the feature *cxim.qngg.Tehr.sFiQa.* For example, considering the method *onCreate()* of the target class we can extract the dependencies shown in Fig 4.3. As we can notice in this little piece of code we can identify four dependencies, two from the declared variables $r2, $r4 and two more from the string declaration of $r5. The whole identification process proceeds along all the others methods of the class. In the end we will obtain a set D = { d_0, d_i, d_{i+1}, ... , d_n } , which includes all the dependencies extracted recursively. Then $\forall\, d_i \notin D$ is going to be removed from the current Soot `Scene`, which represents the whole gadget.

The last step is the slice identification. Unfortunately both these *goodware* have no slice available, so we will report the example of another feature from another example *goodware*, *com.revmob.FullscreenActivity.* After having inspected all the classes for an invocation and having identified the class from which extract the slice, which in this case is *com.revmob.internal.e*, the Extractor tries to define the statements belonging to the slice and the relative dependencies:



```
final class com.revmob.internal.e extends java.lang.Object implements java.lang.Runnable
{
    private java.lang.String a;
    private com.revmob.internal.d b;
    public final void run() {
        com.revmob.internal.e $r0;
        android.content.Intent $r1;
        com.revmob.internal.d $r2;
        android.app.Activity $r3;
        java.lang.String $r4;
        $r0 := @this: com.revmob.internal.e;
        $r1 = new android.content.Intent;
        $r2 = $r0.<com.revmob.internal.e: com.revmob.internal.d b>;
        $r3 = staticinvoke <com.revmob.internal.d: android.app.Activity a(com.revmob.internal.d)>($r2);
        specialinvoke $r1.<android.content.Intent: void <init>(android.content.Context,java.lang.Class)>($r3, class "Lcom/revmob/FullscreenActivity;");
        $r4 = $r0.<com.revmob.internal.e: java.lang.String a>;
        virtualinvoke $r1.<android.content.Intent: android.content.Intent putExtra(java.lang.String,java.lang.String)>("marketURL", $r4);
        $r2 = $r0.<com.revmob.internal.e: com.revmob.internal.d b>;
        $r3 = staticinvoke <com.revmob.internal.d: android.app.Activity a(com.revmob.internal.d)>($r2);
        virtualinvoke $r3.<android.app.Activity: void startActivityForResult(android.content.Intent,int)>($r1, 0);
        return;
    }
}
```

Figure 4.5

By looking the CFG of the method run(), which contains exactly what we are searching, it is possible to observe that it is composed by one single basic block containing the whole body. This is the luckiest scenario, in which in the same basic block we find both the feature declaration and also the `startActivity()` invocation. After having identified the needed statements of the slice, this module analyses each statement searching for unresolved dependencies and adding the needed variables to the final slice. Automatically the module adds all this set of dependencies to the final output, classifying those as *'slice dependencies'*. This information is going to be used during the slice gathering, in which we collect single slices for the *'mining'* feature.

4.3 Slice evaluation

After the extraction of the gadgets we need to evaluate them, in order to weight the whole set of dependencies. The problem is already deep introduced in Section 3.2.5 and here we are looking at it in details.

Because this problem could lead to very different scenarios, due to the fact that it is possible to discover features with a real small dependency chain or with an enormous one, we need to make our solution is general enough to adapt to every situation. In a nutshell, our strategy consists in creating a mock basic app T in which inject the target gadget g_i and extract the feature by the feature extractor. More formally:

$$\begin{aligned} &for\ g_i\ in\ \{\ G\ \}: \\ &\quad E(I(g_i)) \end{aligned}$$

Where {G} is the whole set of gadgets to evaluate, I is the injection function that inject the input gadget in T and E is the evaluation function, which extract the new gadgets from the injected T. During the evaluation we double check that the injection adds only features, without removing any of them. This is crucial due to the Sematic Equivalence that we want to preserve, as explained in Section 3.1. The implementation of I is a simplified version of the Injector module, which does not include the mining slice feature, because in this phase we want to evaluate the pure gadget, without adding any extra component to it.

The template application T is the most basic application, containing only a MainActivity to allow the correct launch of the app, without any extra features. This way we use this set of features as base to subtract from the final feature set detected from the instrumented T and the difference represents the whole set of the features of the gadget.

During the Evaluation phase, we check which kind of features are added for two main reasons:

- Firstly, we do not want to insert too many permissions into the manifest. Indeed, for each attack, we set a maximum threshold of the total amount of permissions that can be added to the instrumented malware. Moreover, we do not want to add any extra dangerous permission in order to not increase the risk of identification [25].

- Secondly, we do not want to count any feature that is already included in the initial malware. For this reason, we extract all the initial features from the starting malware and store them in memory, ready for the match.

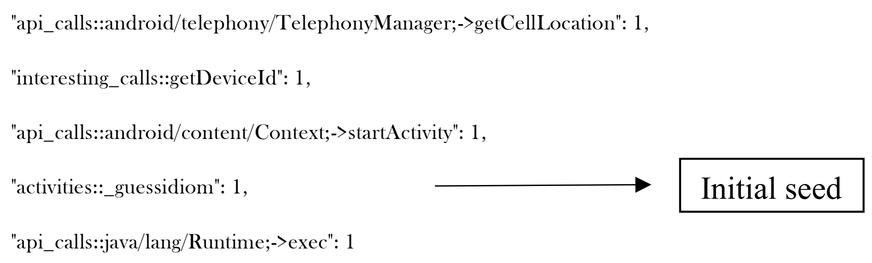
Last, our algorithm automatically excludes gadgets with a superficial contribute, in order to limit the total number of injected features.

4.3.1 PoC

Once we extract the two needed features from the previous phase, we need to evaluate them. Because *cxim.qngg.Tehr.sFiQa* gadget is really little, I will explain in detail *.guessidiom*.

This single feature implies the following set of features:

```
"api_permissions::android_permission_INTERNET": 1,

"interesting_calls::getCellLocation": 1,

"activities::org_cocos2dx_lib_Cocos2dxActivity": 1,

"api_permissions::android_permission_READ_LOGS": 1,

"interesting_calls::getSystemService": 1,

"interesting_calls::Read/Write External Storage": 1,

"api_calls::java/net/HttpURLConnection": 1,

"api_permissions::android_permission_ACCESS_WIFI_STATE": 1,

"activities::template_template_TemplateMainActivity": 1,

"interesting_calls::printStackTrace": 1,

"api_calls::android/net/wifi/WifiManager;->getConnectionInfo": 1,

"interesting_calls::Cipher(r0)": 1,

"activities::_TemplateMainActivity": 1,

"interesting_calls::Cipher(DES)": 1,

"api_permissions::android_permission_READ_PHONE_STATE": 1,

"api_permissions::android_permission_ACCESS_FINE_LOCATION": 1,

"api_calls::android/telephony/TelephonyManager;->getDeviceId": 1,
```



```
"api_calls::android/telephony/TelephonyManager;->getCellLocation": 1,

"interesting_calls::getDeviceId": 1,

"api_calls::android/content/Context;->startActivity": 1,

"activities::_guessidiom": 1,

"api_calls::java/lang/Runtime;->exec": 1
```

Figure 4.6

As we can notice, from a single feature we have extracted a complex set of other extra n features. So we search for the weight w_i of the feature f_i into the classifier weight set and we calculate S = $\sum_i^n w_i$. Then, if S > T, where T is the minimum contribute, we add the whole gadget to the final malware.

4.4 Injection

After having collected the whole set of entities needed for the misclassification and evaluated them, the next step is to inject all of them into the target malware. As already introduced, this whole phase is handled by the Injector module, previously discussed in Section 3.2.3.

The first operation done by this module is loading into the current `Scene` all the classes needed for the injection. This includes also the opaque predicates, already deeply explained in Section 3.2.3, and all the mined slices, which could be used during the application. In order to make it work, all the dependencies are loaded as ad-hoc Jimple classes, that the module knows how to handle for extract only the needed code parts (more in the Example part of this section). Secondly the module inspects all the current gadgets in order to understand where to use the *'mined slices'*. Indeed, it probes all the classes contained into a gadget searching for the Slice class and if it does not find it, it just marks the current gadget as *'sliceless'*. This is going to be used to decide which injection strategy use.

Then, for each gadget it also checks which are the Android components and Permission to add to the final Manifest. In order to achieve it, the module is provided by a recursive function that checks if the ancestor of the current class hierarchy matches with one of the Android components, as Activity, Broadcast Receivers, Service or ContentProvider. If one of these components is detected the module provides to add the relative tag to the manifest. In order not to dramatically warp the behaviour of the application, we have decided not to add any extra intent-filter tag to the malware manifest because it would be sensible to implicit intents.

For example, in case the module discovers a Broadcast Receiver if we inject the relative intent-filters, for example reading a file, our instrumented application will answer to any 'read file' intent that passes through the Android system. This could bring to an unwanted behaviour, in particular in case the instrumented malware has absolutely no relation with the intent caught. E.g. the instrumented malware is a mobile game and the intent-filter injected is a READ_FILE one. Moreover, we have chosen not to import multiple MAIN_ACTIVITY intent filters, in order to not deeply compromise the application flow, compromising the first launched activity

In the case of some extra permissions are needed, the module proceeds by adding them to the final malware. As already mentioned, during each injection, the number of permissions added is limited and it cannot exceed a custom threshold, due to the importance of this kind of feature.

Then, the module needs to identify the malware classes and which of them are eligible for the injection process. The identification of the malware classes is done by exclusion: indeed, it is known a-priori the whole set of classes composing all the gadgets and the default android libraries. From this knowledge base it is possible to automatically identify the classes that belong to the original malware class-set. Subsequently it calculates the average Cyclomatic Complexity of the whole malware set, which is going to be used as our classification meter as already explained in Section 3.2.3. In this way we obtain a set of eligible classes $\{ C \} = \{ c_0, c_i, c_{i+1}, \dots , c_n \}$.

After this phase the module focuses on the injection of the feature invocation. Here two different scenarios are possible: in the first one the selected gadget contains the slice, while in the second one it does not and the module need to use a mined slice. This feature is applicable only to Activity features, due to the slice structure. The slices are represented using the same Java Object adopted in the Extractor.

In the first case scenario, the Injector extracts the actual slice from the Jimple class, crafted by the Extractor. Then it will try to inject the slice into the selected class $c_i \in C$. For each method contained into c_i, the module tries to implant the slice starting from a random statement. This randomness has been introduced in order not to introduce any logical artefact in term of instrumentation, so for a hypothetical analyser it would be harder to identify an instrumentation because it operates randomly. The slice is then shielded into one of the available OP_i , which is also chosen randomly for the same reason. After that the whole slice has been successfully build, it is then injected into the chosen method of c_i.

On the other hand, in the second case scenario, there is no slice available for the feature invocation and the only possibility is to use a mined slice. A mined slice is a slice extracted from a different *goodware* which can be adapted for the invocation of a different feature. This is done by identifying which is the variable containing the invocation target and by changing the reference class, as showed in the Example part of this Section. These slices contain only the necessary dependencies needed from the current invocation and need to be added to the final `Scene` of the instrumented malware as well, otherwise the invocation will fail and it would create dead references into the final code. Once a random mined slice is chosen between the whole set, the injection process is the same as before.

In the end, we double check that all the necessary classes are contained into the final Soot `Scene` and extract them as Android application. The instrumented malware is going to include new components into the manifest, new classes and some modified ones, in which it has injected the feature invocation.

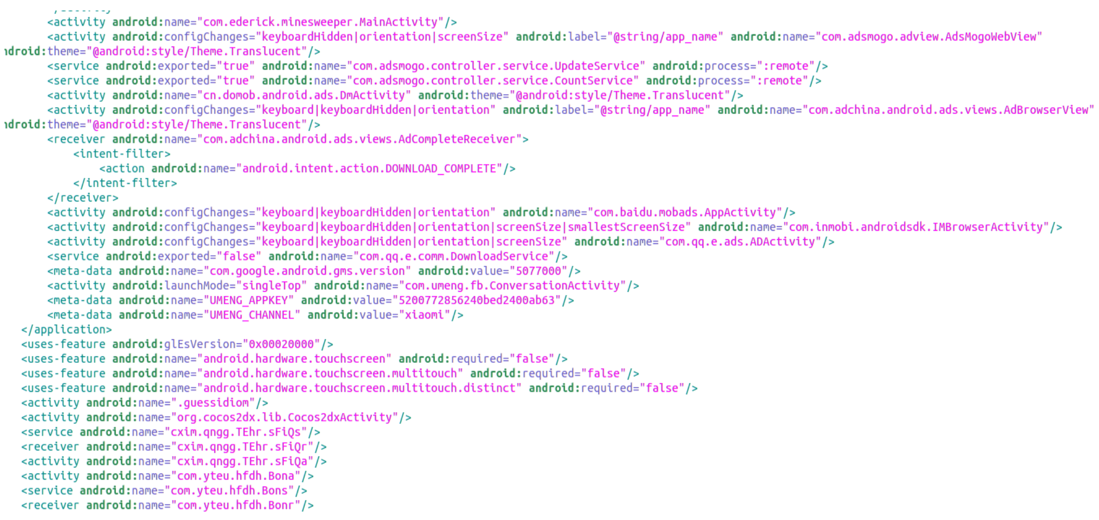
4.4.1 PoC

Getting back our example M^T , once we have extracted the two features , *cxim.qngg.Tehr.sFiQa* and *.guessidiom,* and evaluated them, they will feed the Injector.

After it loads into the current `Scene` all the necessary entities, it inspects the current entities to inject, searching for Android Components to inject into the final AndroidManifest.xml. Taking the example of *cxim.qngg.Tehr.sFiQa* , the whole dependency set is composed by other 5 classes :

- cxim.qngg.TEhr.c, which is a simple Java Class
- cxim.qngg.TEhr.d, which is a simple Java Class
- cxim.qngg.sFiQs, which is a Service
- cxim.qngg.sFiQr, which is a Boradcast Receiver
- cxim.qngg.sFiQa, which is an Activity

When the Injector understands that they are Android Components, it create the ad-hoc xml tag to add to the final Manifest.



```
        <activity android:name="com.ederick.minesweeper.MainActivity"/>
        <activity android:configChanges="keyboardHidden|orientation|screenSize" android:label="@string/app_name" android:name="com.adsmogo.adview.AdsMogoWebView"
droid:theme="@android:style/Theme.Translucent"/>
        <service android:exported="true" android:name="com.adsmogo.controller.service.UpdateService" android:process=":remote"/>
        <service android:exported="true" android:name="com.adsmogo.controller.service.CountService" android:process=":remote"/>
        <activity android:name="cn.domob.android.ads.DmActivity" android:theme="@android:style/Theme.Translucent"/>
        <activity android:configChanges="keyboard|keyboardHidden|orientation" android:label="@string/app_name" android:name="com.adchina.android.ads.views.AdBrowserView"
droid:theme="@android:style/Theme.Translucent"/>
        <receiver android:name="com.adchina.android.ads.views.AdCompleteReceiver">
            <intent-filter>
                <action android:name="android.intent.action.DOWNLOAD_COMPLETE"/>
            </intent-filter>
        </receiver>
        <activity android:configChanges="keyboard|keyboardHidden|orientation" android:name="com.baidu.mobads.AppActivity"/>
        <activity android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize" android:name="com.inmobi.androidsdk.IMBrowserActivity"/>
        <activity android:configChanges="keyboard|keyboardHidden|orientation|screenSize" android:name="com.qq.e.ads.ADActivity"/>
        <service android:exported="false" android:name="com.qq.e.comm.DownloadService"/>
        <meta-data android:name="com.google.android.gms.version" android:value="5077000"/>
        <activity android:launchMode="singleTop" android:name="com.umeng.fb.ConversationActivity"/>
        <meta-data android:name="UMENG_APPKEY" android:value="5200772856240bed2400ab63"/>
        <meta-data android:name="UMENG_CHANNEL" android:value="xiaomi"/>
    </application>
    <uses-feature android:glEsVersion="0x00020000"/>
    <uses-feature android:name="android.hardware.touchscreen" android:required="false"/>
    <uses-feature android:name="android.hardware.touchscreen.multitouch" android:required="false"/>
    <uses-feature android:name="android.hardware.touchscreen.multitouch.distinct" android:required="false"/>
    <activity android:name=".guessidiom"/>
    <activity android:name="org.cocos2dx.lib.Cocos2dxActivity"/>
    <service android:name="cxim.qngg.TEhr.sFiQs"/>
    <receiver android:name="cxim.qngg.TEhr.sFiQr"/>
    <activity android:name="cxim.qngg.TEhr.sFiQa"/>
    <activity android:name="com.yteu.hfdh.Bona"/>
    <service android:name="com.yteu.hfdh.Bons"/>
    <receiver android:name="com.yteu.hfdh.Bonr"/>
```

Figure 4.7

Figure 4.7 shows the final manifest file of M'^{T}, which is the instrumented version of M^T . As we can notice all the needed features has been added correctly. This way they will be recognized from the Drebin Extractor and considered in the final evaluation.

In order to simplify the entire operation we have stored all these information into a Set iterable by CC value, like : CC$_i$ → { c$_0$, c$_i$, ... , c$_n$ }. We obtain a set {C} containing all the eligible malware classes, depending on the current average value that we want to obtain. The module starts then iterating between all the classes belonging to {C} trying to inject the selected slice.

In our case the feature does not have any slice. So, we need to use a mined slice. The module selects randomly one between the available slice, we are supposing that during this iteration the slice S. This is composed by:

```
0. this := @this: Slice00ADBDEE7ED68BB4C243F30EA0BABD56C034574303783924DC9654F2916A43E8;

1. $r0 = virtualinvoke this.<android.content.Context: android.content.Context getApplicationContext()>();

2. $r2 = new android.content.Intent;

3. $r3 = staticinvoke <com.yteu.hfdh.c.h: java.lang.Class a(android.content.Context,java.lang.Class)>($r0, class
"Lcom/yteu/hfdh/Bona;");

 4. specialinvoke $r2.<android.content.Intent: void <init>(android.content.Context,java.lang.Class)>($r0, $r3);

5. $i3 = staticinvoke <com.yteu.hfdh.c.h: int p(android.content.Context)>($r0);

6. virtualinvoke $r2.<android.content.Intent: android.content.Intent putExtra(java.lang.String,int)>("l", $i3);

7. virtualinvoke $r2.<android.content.Intent: android.content.Intent addFlags(int)>(268435456);

8. virtualinvoke $r0.<android.content.Context: void startActivity(android.content.Intent)>($r2);
```

```
9. staticinvoke <com.yteu.hfdh.c.h: void x(android.content.Context)>($r0);

10. return;
```

Figure 4.8

The Figure 4.8 shows the Jimple representation of S, which is a mined slice gathered from the extraction of the feature *com.yteu.hdh.Bona*. All the dependencies are circled, in black the ones that are already included in the standard libraries and will not be extracted, while in red the only dependency which is going to be considered. The next step is to adapt this mined slice, which in this case consists in a single modification. Indeed we need to modify only the line 3, replacing the string *"Lcom/yteu/hfdh/Bona;"* in *"Lcxim/qngg/sFiQa;"*, which is possible through Soot.

The last step is the injection of the adapted slice into a selected class ci. As already mentioned the entire slice will be injected inside an opaque predicate, which will shield it from static analysis tool. In Fig 4.9 is indeed shown the final injected code, divided in three main regions: the blue section identifies the code already present into the method, the black section represents the Opaque predicate part, while the red one is the Slice.

```
.method public handleMessage(Landroid/os/Message;)V
    .locals 24
    .param p1, "msg"    # Landroid/os/Message;

    .line 295
    move-object/from16 v0, p0

    .line 295
    move-object/from16 v1, p1

    .line 295
    invoke-super {v0, v1}, Landroid/os/Handler;->handleMessage(Landroid/os/Message;)V

    .line 296
    move-object/from16 v0, p1

    .line 296
    .local v3, "$i0":I, ""
    iget v3, v0, Landroid/os/Message;->what:I

    .line 296
    sparse-switch v3, :sswitch_data_0

    .line 296
    goto :goto_0

    .line 296
    :goto_0
    return-void

    .line 298
    :sswitch_0
    move-object/from16 v0, p0

    .line 298
    .local v4, "$r2":Ljava/lang/String;, ""
    iget-object v4, v0, Lcn/domob/android/a/a/d$c;->b:Ljava/lang/String;

    .line 298
    const/4 v5, 0x0

    .line 298
    move-object/from16 v0, p0

    .line 298
    invoke-direct {v0, v4, v5}, Lcn/domob/android/a/a/d$c;->a(Ljava/lang/String;Z)V

    .line 299
    invoke-static {}, Lcn/domob/android/a/a/d;->b()Lcn/domob/android/m/i;

    move-result-object v6

    .line 299
    .local v6, "$r3":Lcn/domob/android/m/i;, ""
    const-string v7, "upload picture failed"

    .line 299
    invoke-virtual {v6, v7}, Lcn/domob/android/m/i;->b(Ljava/lang/String;)V

    new-instance v8, Ljava/util/Random;

    .local v8, "$r2":Ljava/util/Random;, ""
    invoke-direct {v8}, Ljava/util/Random;-><init>()V

    const/16 v5, 0x32

    invoke-virtual {v8, v5}, Ljava/util/Random;->nextInt(I)I

    move-result v9

    .local v9, "$i0":I, ""
    if-gez v9, :cond_0

    const/4 v10, 0x0

    .local v10, "$z7":Z, ""
```

Regular Class code

Opaque

```
goto :goto_1

:cond_0
const/4 v10, 0x1

:goto_1
move v11, v10

.local v11, "z0":Z, ""
const/16 v5, 0x14

invoke-virtual {v8, v5}, Ljava/util/Random;->nextInt(I)I

move-result v12

.local v12, "$i1":I, ""
if-gez v12, :cond_1

const/4 v13, 0x0

.local v13, "$z8":Z, ""
goto :goto_2

:cond_1
const/4 v13, 0x1

:goto_2
move v14, v13

.local v14, "z1":Z, ""
invoke-virtual {v8}, Ljava/util/Random;->nextBoolean()Z

move-result v15

.local v15, "z3":Z, ""
invoke-virtual {v8}, Ljava/util/Random;->nextBoolean()Z

move-result v16

.local v16, "z4":Z, ""
invoke-virtual {v8}, Ljava/util/Random;->nextBoolean()Z

move-result v17

.local v17, "z5":Z, ""
invoke-virtual {v8}, Ljava/util/Random;->nextBoolean()Z

move-result v18

.local v18, "z6":Z, ""
if-nez v15, :cond_2

if-nez v16, :cond_2

if-nez v17, :cond_a

:cond_2
if-eqz v16, :cond_3

if-nez v15, :cond_3

if-eqz v17, :cond_a

:cond_3
if-nez v17, :cond_4

if-nez v14, :cond_4

if-nez v15, :cond_a

:cond_4
if-eqz v16, :cond_5

if-nez v18, :cond_a

:cond_5
```

```
if-nez v17, :cond_6

if-eqz v16, :cond_6

if-nez v18, :cond_a

:cond_6
if-eqz v16, :cond_7

if-nez v17, :cond_a

:cond_7
if-nez v11, :cond_8

if-nez v14, :cond_8

goto :goto_3

:cond_8
if-nez v11, :cond_9

if-nez v14, :cond_9

goto :goto_3

:cond_9
```

Opaque predicate

```
new-instance v19, Landroid/content/Intent;

.local v19, "$r2":Landroid/content/Intent;, ""
const-class v22, Lcxim/qngg/TEhr/sFiQa;

move-object/from16 v0, v21

move-object/from16 v1, v22

invoke-static {v0, v1}, Lcom/yteu/hfdh/c/h;->a(Landroid/content/Context;Ljava/lang/Class;)Ljava/lang/Class;

move-result-object v20

.local v20, "$r3":Ljava/lang/Class;, ""
move-object/from16 v0, v19

move-object/from16 v1, v21

move-object/from16 v2, v20

invoke-direct {v0, v1, v2}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V

move-object/from16 v0, v21

invoke-static {v0}, Lcom/yteu/hfdh/c/h;->p(Landroid/content/Context;)I

move-result v23

.local v23, "$i3":I, ""
const-string v7, "l"

move-object/from16 v0, v19

move/from16 v1, v23

invoke-virtual {v0, v7, v1}, Landroid/content/Intent;->putExtra(Ljava/lang/String;I)Landroid/content/Intent;

const v5, 0x10000000

move-object/from16 v0, v19

invoke-virtual {v0, v5}, Landroid/content/Intent;->addFlags(I)Landroid/content/Intent;

move-object/from16 v0, v21

move-object/from16 v1, v19

invoke-virtual {v0, v1}, Landroid/content/Context;->startActivity(Landroid/content/Intent;)V

move-object/from16 v0, v21
```

slice

```
invoke-static {v0}, Lcom/yteu/hfdh/c/h;->x(Landroid/content/Context;)V

return-void

:cond_a
:goto_3
return-void

.line 299
return-void

.line 302
:sswitch_1
move-object/from16 v0, p0

.line 302
iget-object v4, v0, Lcn/domob/android/a/a/d$c;->b:Ljava/lang/String;

.line 302
const/4 v5, 0x1

.line 302
move-object/from16 v0, p0

.line 302
invoke-direct {v0, v4, v5}, Lcn/domob/android/a/a/d$c;->a(Ljava/lang/String;Z)V

.line 303
invoke-static {}, Lcn/domob/android/a/a/d;->b()Lcn/domob/android/m/i;

move-result-object v6

.line 303
const-string v7, "upload picture successful"
```

Regular Class code

Figure 4.9

Each slice is injected in a different class in order to avoid any kind of conflict during the execution of the instrumented application.

After the injection it is verified that the target feature has been correctly injected and that the final malware is able to correctly evade the target classifier.

5 Conclusions

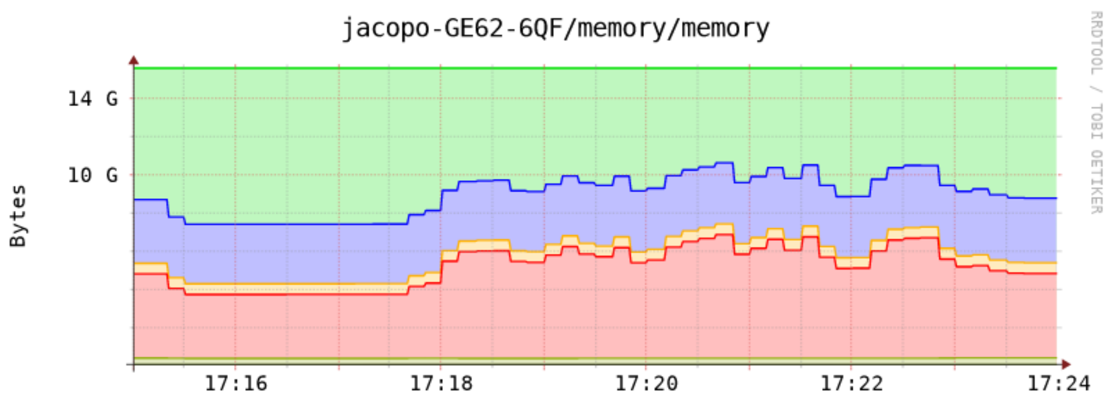
In this section we will present the results obtained with the developed framework. We will consider different aspects of injection success: indeed we can consider an implant successful if the current application keep works correctly, preserve its malicious behaviour after the injection, if it is able to correctly evade the classifier and if it is resistant to dead code elimination. All these aspects are fundamental for the final evaluation because the combination of these characteristic guarantees a stealth and efficient injection strategy. For the evaluation phase we have used different tools that we are going to present in the following sections.

This phase has been performed in part on my local computer and in part on the cluster computer provided by S2LAB. Indeed, while the experiments were focused on a reduced size sample set they could be handled in local, while if they focus on large scale experiment it could not have been performed on a single computer, due to the high resource consumption of the whole pipeline. Indeed, due to the huge number of operation needed and the number of objects to save in memory, the whole pipeline needs high resource availability, which is going to be discussed in the following sections.

5.1 Resource consumption

The whole PRISM pipeline, which includes all the steps described, is a heavy and expensive process. Figures 5.1, 5.2, 5.3 and 5.4 show the actual load on the RAM, one of the CPU cores, the total system load and the fork activity during a full pipeline operation. As it is possible to notice, it consumes a high amount of resources, particularly in terms of memory consumption: for a single run on a target malware it requires an average of 5 GB of memory. The whole process starts around the 17:17 and terminates at 17:23, for an average time of 6 minutes. The graphs have been extracted by Collectd [26].

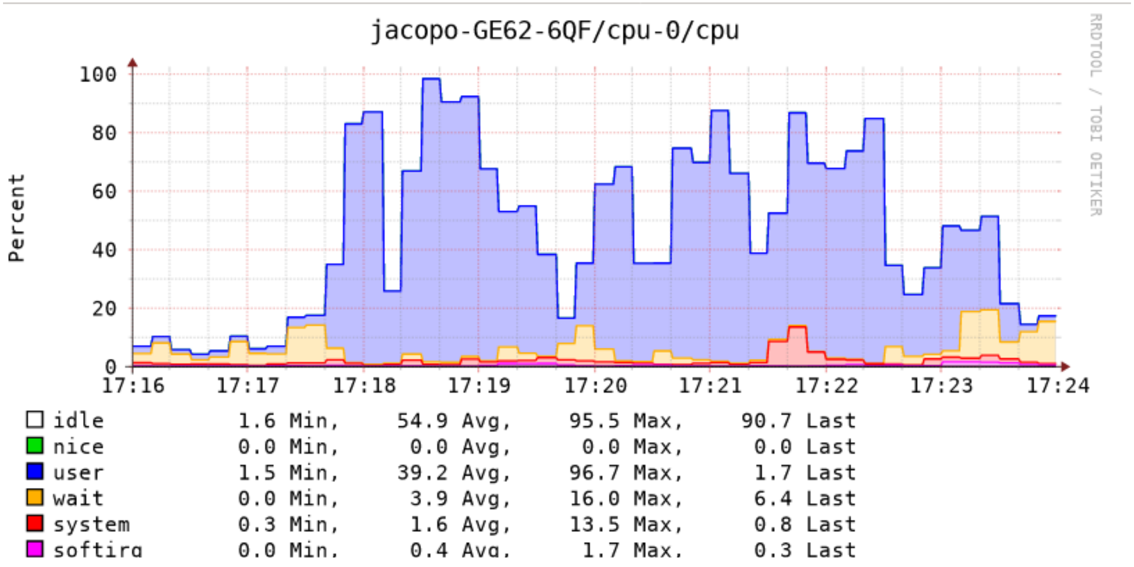
For this reason, a lot of data that are reused iteration after iteration are stored into persistent folders in order to reduce the amount of computational resources, in particular on large scale experiments. An example are the current features contained into a gadget: once they are extracted once, they are saved into a JSON file and all the times they are needed they are directly loaded from there, without re-extracting and re-evaluating the whole gadget. This strategy works pretty fine; indeed, for example considering the same execution shown in the



```
🟩 free          5.0G Min,    6.6G Avg,    8.2G Max,    7.7G Last
🟦 cached        3.1G Min,    3.2G Avg,    3.4G Max,    3.2G Last
🟧 buffered    573.0M Min,  574.5M Avg,  577.1M Max,  577.1M Last
🟥 used          3.3G Min,    4.8G Avg,    6.5G Max,    3.7G Last
🟩 slab_recl   322.9M Min,  329.0M Avg,  348.0M Max,  331.5M Last
🟩 slab_unrecl  80.8M Min,   81.0M Avg,   83.3M Max,   80.9M Last
```

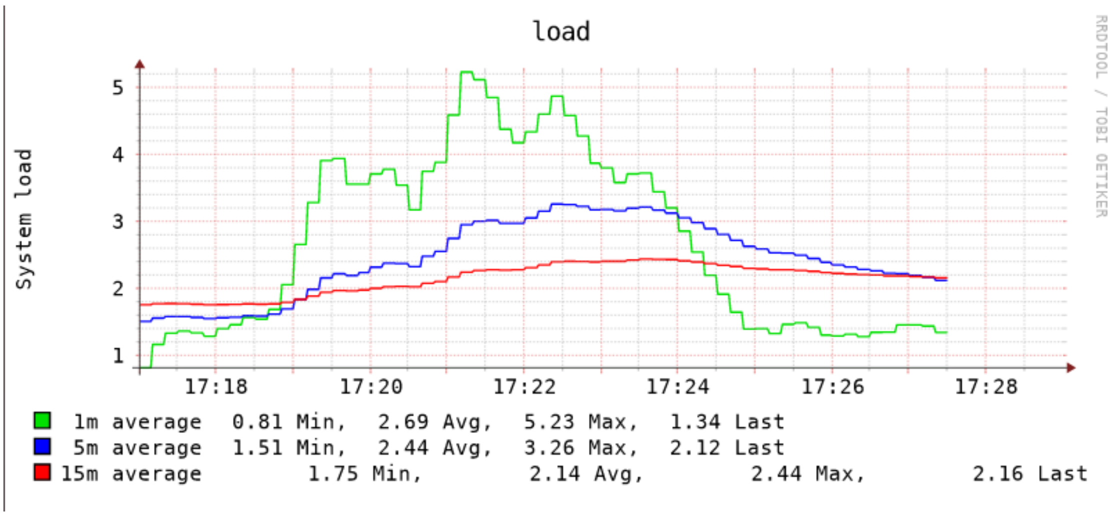
Figure 5.1

Figures, with this approach takes an average of 3min, instead of the initial 6 minutes.



```
□ idle          1.6 Min,   54.9 Avg,   95.5 Max,   90.7 Last
🟩 nice          0.0 Min,    0.0 Avg,    0.0 Max,    0.0 Last
🟦 user          1.5 Min,   39.2 Avg,   96.7 Max,    1.7 Last
🟧 wait          0.0 Min,    3.9 Avg,   16.0 Max,    6.4 Last
🟥 system        0.3 Min,    1.6 Avg,   13.5 Max,    0.8 Last
🟪 softirq       0.0 Min,    0.4 Avg,    1.7 Max,    0.3 Last
```

Figure 5.2



```
🟩  1m average  0.81 Min,  2.69 Avg,  5.23 Max,  1.34 Last
🟦  5m average  1.51 Min,  2.44 Avg,  3.26 Max,  2.12 Last
🟥 15m average        1.75 Min,        2.14 Avg,        2.44 Max,        2.16 Last
```

Figure 5.3

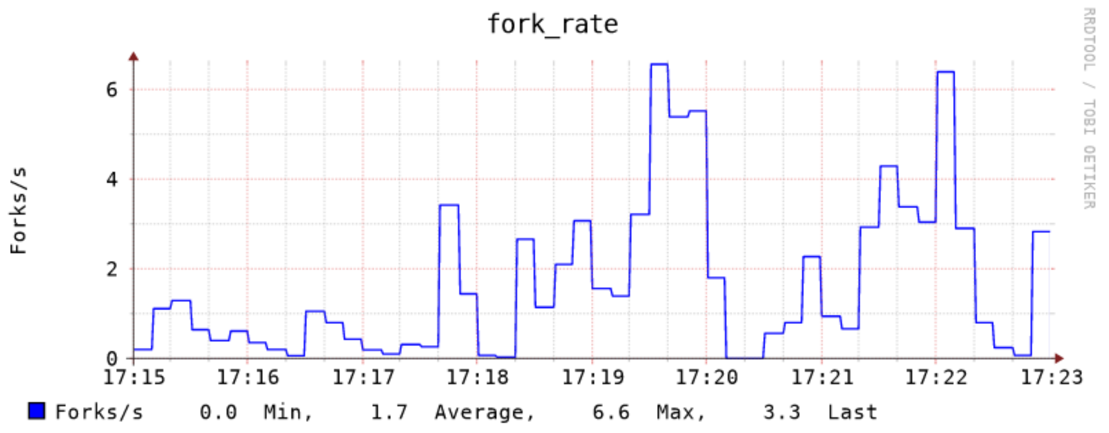


Figure 5.4

5.2 Working application

In order to verify the correct behaviour of the final instrumented malware, we have and analysed it with Android Studio. Indeed, it is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development. It is a replacement for the Eclipse Android Development Tools (ADT) as the primary IDE for native Android application development.

Included in Android studio we can find a huge set of tools, like for example ProGuard, which is a Java class file shrinker, optimizer, obfuscator, and preverifier. The shrinking step detects and removes unused classes, fields, methods and attributes. The optimization step analyses and optimizes the bytecode of the methods. The obfuscation step renames the remaining classes, fields, and methods using short meaningless names. These first steps make the code base smaller, more efficient, and harder to reverse-engineer.

Another interesting built-in-tool is Lint, which provides a similar ProGuard feature set as:

- Missing translations (and unused translations)
- Layout performance problems (all the issues the old layout tool used to find, and more)
- Unused resources
- Unused code
- Accessibility and internationalization problems (hardcoded strings, missing contentDescription, etc.)

Moreover, it includes the whole SDK platform tools, which is needed to Soot in order to correctly instrument Android applications. In order to being robust to every Android application, we decided to include the whole set of Android platforms starting from Android 10 until the last one, which is Android 28.

The last set of components that was essential for our tests were the emulators, that simulates Android devices on your computer so that you can test your application on a variety of devices and Android API levels without needing to have each physical device. The emulator provides almost all of the capabilities of a real Android device allowing to simulate incoming phone calls and text messages, specify the location of the device, simulate different network speeds, simulate rotation and other hardware sensors, access the Google Play Store, and much more. For testing the instrumented malware we used emulators with the lasts Android versions, as API 26,27 and 28.

Unfortunately, this kind of evaluation requires a large set of resources, because each instrumented application should be installed on an emulator and verify the current run. Moreover, it is almost impossible to automate the verification of the correct behaviour of the app: we would need a smart Android app tester that remembers how the non-instrumented malware work in order to compare with the instrumented one, which is not an easy problem and even more absolutely not cheap from the resources point of view.

For these reasons we decided not to include this feature in the whole pipeline, testing it only on a limited set of 15 malware. We have successfully verified that all these malware successfully work after the injection, preserving their natural behaviour, confirming that the gadgets have been correctly instrumented and the code is never executed. Each application has been deeply tested and none of them crashes in any tested actions.

5.3 Evasion ratio

We have currently tested the framework against 512 malwares and we have successfully verified that it works in all the situation tested. Unfortunately, due to Soot limits, for some application some exceptions are thrown, making impossible to correctly ending the attack. As far we have tested we have encountered two main errors that we have already issued on the Soot Github: the first error derives from Android Manifest parsing, while the second one derives from an error thrown from the Jimple parser, which apparently is only thrown during the multiprocessing, never during single-core execution.

Fortunately, these application are a limited number, on the current experiment set. As shown in Figure 5.5 we have recorded 467 (91.2%) malware successfully instrumented, 21 (4.1%) which are not recognised as malware from our classifier and 24 (4.7%) which are affected by the Soot errors I have mentioned before. Indeed even if our model has a high score set, as shown in Figure 4.1 – Section 4.1, some malware are misclassified as `goodware` even before the attack takes place, which are automatically skipped from the framework. The whole attack on 512 malware on 40 cores has taken 2h 4m and has been run on the S2LAB computer cluster. We run a different malware attack on each core, using multiprocessing for speed the whole process.

These results demonstrate that **PRISM** is able to correctly instrument and misclassify a large set of real-world application and that Linear SVM classifier are currently vulnerable to this kind of attack, even at problem space level.

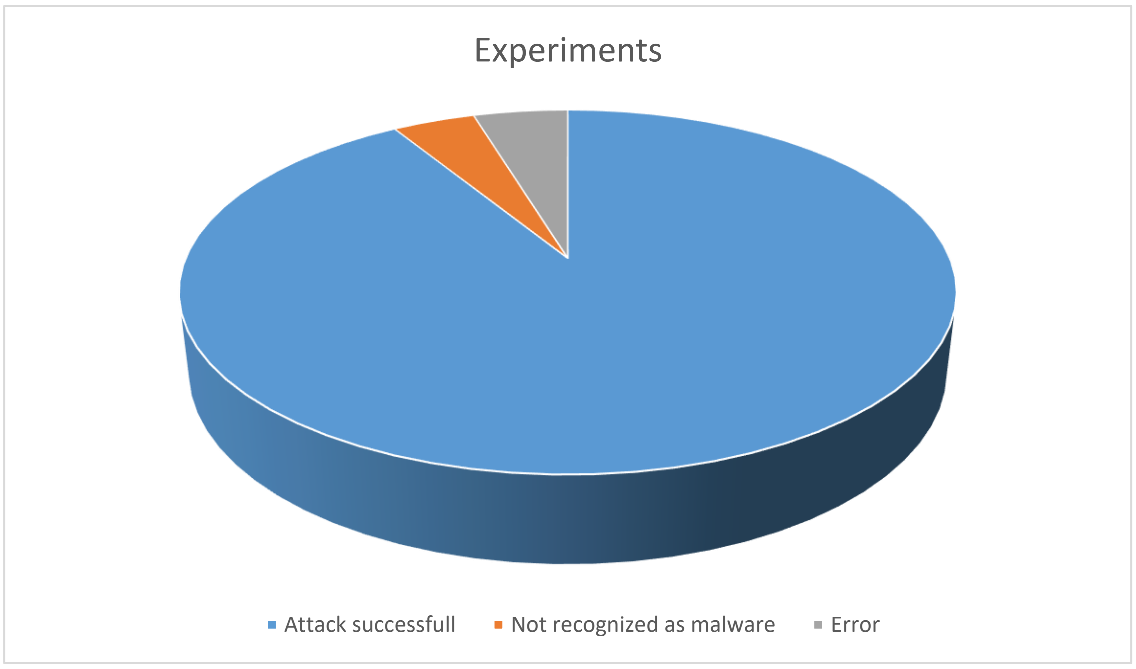


Figure 5.5

5.4 Dead code elimination

In order to verify the robustness of our injection to static analysis techniques, we tried to use static analysis tools and strategies on out final results. For the same reason of the high resource consumption and the difficulty of automate the whole process, we decided to test this robustness on the same sample set of Section 5.2.

As first we have used Android Studio tools for dead code elimination, which uses ProGuard and Lint, using two different approaches. In the first approach we directly load the final Android app into Android Studio, make it decompile the .dex files and analyse them. The result has been always positive and no unused code has been detected by it. Secondly, we tried to extract the java code from the instrumented apk and directly run the dead code elimination feature on it. It has been possible by using dex2jar [27], which is a tool able to convert a .dex file into a .jar . This way it was possible to successfully extract the whole injected invocation in java code and copy it into Android Studio and evaluate it directly as source code. Also, in this case the static analysers failed to identify our invocation as dead code, thanks to the effectiveness of our opaque predicates.

We have also tried to run Soot for dead code elimination. Indeed, Soot is also a Java Optimizer and during the transformation to its IR it automatically eliminates any dead code discovered. We have then extracted the slice from the converted jar and created a new Java class containing only the injected code slice, as showed in Figure 5.5. Then we replaced the inner code with a simple *System.out.println("DEAD")* because the goal is just to verify that the inner part of the loop survives the conversion. In the Figure 5.6 it is shown the Jimple representation of this example class and as we can notice the body of the inner loop is still present, confirming the robustness of our strategy.

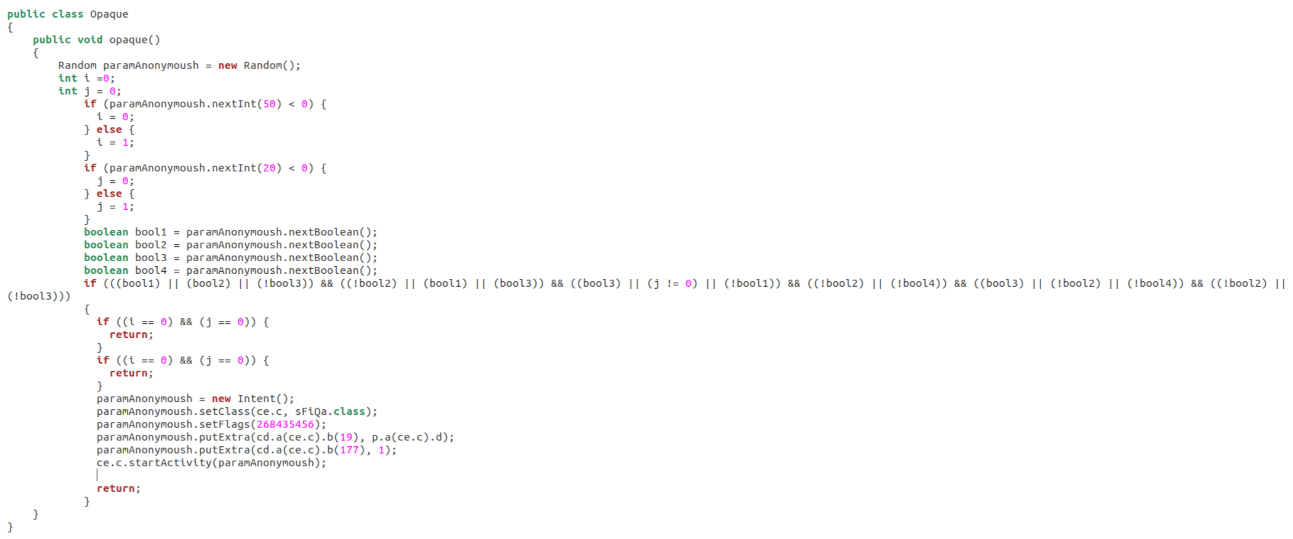
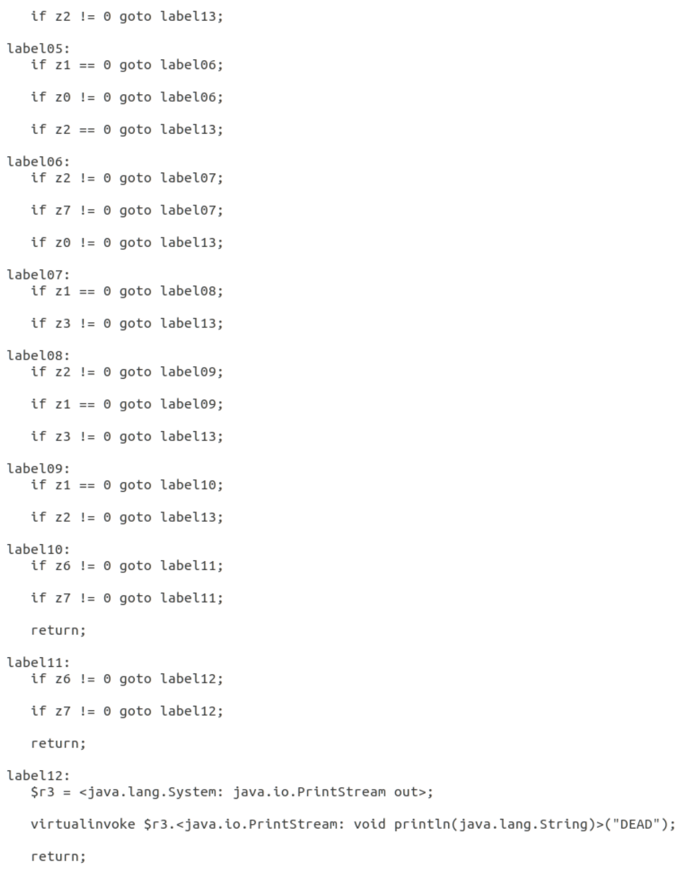

```
public class Opaque
{
    public void opaque()
    {
        Random paramAnonymoush = new Random();
        int i =0;
        int j = 0;
            if (paramAnonymoush.nextInt(50) < 0) {
              i = 0;
            } else {
              i = 1;
            }
            if (paramAnonymoush.nextInt(20) < 0) {
              j = 0;
            } else {
              j = 1;
            }
            boolean bool1 = paramAnonymoush.nextBoolean();
            boolean bool2 = paramAnonymoush.nextBoolean();
            boolean bool3 = paramAnonymoush.nextBoolean();
            boolean bool4 = paramAnonymoush.nextBoolean();
            if (((bool1) || (bool2) || (!bool3)) && ((!bool2) || (bool1) || (bool3)) && ((bool3) || (j != 0) || (!bool1)) && ((!bool2) || (!bool4)) && ((bool3) || (!bool2) || (!bool4)) && ((!bool2) ||
(!bool3)))
            {
              if ((i == 0) && (j == 0)) {
                return;
              }
              if ((i == 0) && (j == 0)) {
                return;
              }
              paramAnonymoush = new Intent();
              paramAnonymoush.setClass(ce.c, sFiQa.class);
              paramAnonymoush.setFlags(268435456);
              paramAnonymoush.putExtra(cd.a(ce.c).b(19), p.a(ce.c).d);
              paramAnonymoush.putExtra(cd.a(ce.c).b(177), 1);
              ce.c.startActivity(paramAnonymoush);
              
              return;
            }
    }
}
```

Figure 5.6



```
    if z2 != 0 goto label13;

label05:
    if z1 == 0 goto label06;

    if z0 != 0 goto label06;

    if z2 == 0 goto label13;

label06:
    if z2 != 0 goto label07;

    if z7 != 0 goto label07;

    if z0 != 0 goto label13;

label07:
    if z1 == 0 goto label08;

    if z3 != 0 goto label13;

label08:
    if z2 != 0 goto label09;

    if z1 == 0 goto label09;

    if z3 != 0 goto label13;

label09:
    if z1 == 0 goto label10;

    if z2 != 0 goto label13;

label10:
    if z6 != 0 goto label11;

    if z7 != 0 goto label11;

    return;

label11:
    if z6 != 0 goto label12;

    if z7 != 0 goto label12;

    return;

label12:
    $r3 = <java.lang.System: java.io.PrintStream out>;

    virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String)>("DEAD");

    return;
```

Figure 5.7

5.5 Future works

We are now working for attacking the SecSVM classifier proposed in [1], which bases its robustness assumption on the fact that distribute the weights of a classifier between a larger feature set. Indeed, in this way the attacker should modify a higher number of features in order to bypass the classifier, and they suppose *that if a large number of features has to be manipulated to evade detection, it may not even be possible to construct the corresponding malware sample without compromising its malicious functionality.* We want to demonstrate that the SecSVM assumption is wrong and that the real perturbation threshold regards the application size, not the number of feature modified.

Indeed, PRISM focuses only on the addition of extra features, exactly for preserve the malicious behaviour of the initial malware. The whole framework focuses on being able to extract and transplant autonomous gadgets with all the needed dependencies without interfering with the malicious functionality of the initial malware or between them. This way the number of features to modify does not represent an constraint, except they imply a huge size increase.

While this modification is going to happen soon, there are also long-term future works. Indeed, while currently it is resistant only against static analysers, we would like to extend the whole robustness also against dynamic malware analysis providing stealth also during the app execution, which represents a huge challenge.

Moreover, we would like to create a module for the automatic generation of opaque predicates. For the current state of work, we are using a limited set of opaque predicates, which potentially bring to the creation of artefacts into the final malware. Indeed, if we are going to inject a huge number of features, as we normally do, it is just a matter of probabilities that the same opaque predicate appear multiple times. But we decided to have this limit because at least it is robust to static dead code pruning.

At last, we would like to extend the attack also for removing features from a target Android app, maintaining the correct malicious behaviour. This issue is really complex because the semantic equivalence of the program is not preserved by design. It is necessary to find a successful strategy for achieve it.

5.6 Tools tried but not adopted

During the development of **PRISM** we tried different solutions in order to find the most suitable tool for instrumenting the target application and extract the code slices.

The two other main projects that we have tried but that we could not adopt for our purposes have been:

- SAAF
- WALA

SAAF (Static Android Analysis Framework) is a static analyser Proof of Concept of the following paper [28] [29]. It supports Program Slicing on smali code. It offers several quick-checks to check if some given app makes uses of certain features (e.g., uses classloaders, calls a method of interest, contains likely patched code, etc.). It has a GUI where the APK contents can be viewed and bytecode can be searched. CFGs can be created for (selected) methods. Analysis results can be persisted to a MySQL DB or to XML files. The main feature is the ability to calculate program slices for arbitrary method invocations and their corresponding parameters. SAAF will then calculate a slice for this so called slicing criterion and search for all constants which are part of that slice. In other words, SAAF will create def-use chains with the def information being the result and the use information being the slicing criterion.

For example, the slicing criterion could describe the method android/telephony/SmsManager->sendTextMessage(...) and the first parameter of that method (the telephone number). SAAF will then search for all invocations of that method in the smali code and will search for all constants which could be used as input for that parameter.

But unfortunately, it has a huge limit, because currently does not backtrack into methods found this way while backtracking a register. This would be fundamental to PRISM to correctly work, because if for example the searched feature is a URL-like feature we need to being able to backtracking until we find a suitable method invocation. For this reason, it has been discarded.

Then we tried WALA [30], which is an awesome framework for static analysis in general and it also support mobile application, as Android ones. It is indeed a set of Java libraries for static and dynamic program analysis, initially developed at IBM T.J. Watson Research Center.

It provides:

- Pointer analysis / call graph construction
- Interprocedural dataflow analysis framework
- Context-sensitive slicing framework, with customizable dependency tracking
- Multiple language compatibility

- Generic analysis utilities and data structures
- Limited code transformation

Indeed, it was the last point that prevented to use this framework for PRISM. Even if the static analysis part was excellent and it was possible to retrieve the needed slice, we have not found any way to export it for inject it in the target application. Moreover, it was declared into the WALA Intermediate Representation, that was difficult to translate into smali or java bytecode in order to export it and injecting it with Soot.

After these tries we have decided to adopt Soot for the whole Instrumentation part of the malware, which allows a higher grade of flexibility.

Bibliography:

[1] S. M. I. M. M. S. M. I. B. B. S. Ambra Demontis, “Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection,” in *IEEE Transactions on Dependable and Secure Computing*, 2017.

[2] K. P. N. M. P. B. M. A. M. P. GROSSE, “Adversarial perturbations against deep neural networks,” in *arXiv preprint arXiv:1606.04435* , 2016.

[3] S2LAB, “MainPage,” [Online]. Available: https://s2lab.kcl.ac.uk/.

[4] “WannaCry,” [Online]. Available: https://it.wikipedia.org/wiki/WannaCry.

[5] “2018 Malware Forecast,” 2017. [Online]. Available: https://nakedsecurity.sophos.com/2017/11/07/2018-malware-forecast-the-onward-march-of-android-malware/.

[6] S. D. university. [Online]. Available: https://cseweb.ucsd.edu/classes/fa03/cse231/lec6seq.pdf.

[7] M. Weiser, “Program Slicing,” in *IEEE Transactions on Software Engineering*, 1984.

[8] B. K. a. J. Laski, “Dynamic program slicing,” in *Information Processing Letters*, 1998.

[9] T. R. a. D. B. S. Horwitz, “Interprocedural slicing using dependence graphs,” 1990.

[10] M. S. M. H. G. K. R. Daniel Arp, “DREBIN: Effective and Explainable Detection,” in *Proc. of the 21st NDSS*, 2014.

[11] Soot, “Soot framework,” [Online]. Available: http://www.sable.mcgill. ca/soot/.

[12] I. E. E. F. B. L. A. R. C. X. A. P. T. K. D. S. Kevin Eykholt, “Robust Physical-World Attacks on Deep Learning Models,” in *CVPR 2018*, 2017.

[13] A. growth, “newzoo.com,” [Online]. Available: https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/.

[14] Sophos, “malware Forcasts 2018,” [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/malware-forecast-2018.pdf.

[15] Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Cyclomatic_complexity.

[16] FlowDroid, “FlowDroid GitHub,” [Online]. Available: https://github.com/secure-software-engineering/FlowDroid.

[17] “Android C/C++ Activity,” [Online]. Available: https://expertise.jetruby.com/android-ndk-using-c-c-native-libraries-to-write-android-apps-21550cdd86a.

[18] C. K. a. E. K. Andreas Moser, “Limits of Static Analysis for Malware Detection,” in *Twenty-Third Annual Computer Security Applications Conference* , 2007.

[19] Wikipedia, “3SAT Boolean problem,” [Online]. Available: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[20] Scikit-learn. [Online]. Available: https://scikit-learn.org/stable/index.html.

[21] F. P. R. J. J. K. L. C. Feargus Pendlebury, “TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time,” 2018.

[22] T. F. B. J. K. a. Y. L. T. K. Allix, “Androzoo: Collecting Millions of Android Apps for the Research Community,” in *ACM Mining Software Repositories*, 2016.

[23] A. K. M. C. T. S. A. R. B. R. F. B. Miller, “Measurement for Malware Detection,” in *DIMVA*, 2016.

[24] Soot, “ExceptionalBlockGraph API,” [Online]. Available: https://www.sable.mcgill.ca/soot/doc/soot/toolkits/graph/ExceptionalBlockGraph.html.

[25] Android, “Permissions,” [Online]. Available: https://developer.android.com/guide/topics/permissions/overview#permission-groups.

[26] Collectd, “MainPage,” [Online]. Available: https://collectd.org/.

[27] Dex2Jar, “Dex2Jar GitHub,” [Online]. Available: https://github.com/pxb1988/dex2jar.

[28] M. U. H. ,. M. S. Johannes Hoffmann, “Slicing Droids: Program Slicing for Smali Code,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[29] SAAF. [Online]. Available: https://github.com/SAAF-Developers/saaf.

[30] WALA, “WALA GitHub,” [Online]. Available: https://github.com/wala/WALA.

[31] Smali/Baksmali. [Online]. Available: https://github.com/JesusFreke/smali/wiki.

[32] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/.