Software Verification Techniques for Malware Detection

MRes Year 2 Thesis

Robert Payne

B.A - Media, Macquarie University, 2016
B.IT, Macquarie University, 2016

MACQUARIE University SYDNEY·AUSTRALIA

Department of Computing
Faculty of Science
Macquarie University, NSW 2109, Australia

Submitted 24th October 2019

Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed: R. Payne ....

Date: 07/10/2019 ....

Acknowledgements

I would like to thank my supervisor Franck Cassez for his guidance and patience throughout the course of this thesis.

I would also like to thank both Franck Cassez and Anthony Sloane for access to the tool Skink and their creation of it.

Abstract

Information leakage is the undesired output of information from a program that reveals the value or nature of a piece of private information that should not be shared. Detecting information leakage is a problem that has been approached using methods including taint tracking and analysis which deals with the marking of private data in a program and tracking its flow to detect if that flow is connected to an undesired output.

In this thesis, a technique is presented to specify information leaks in programs with regard to assertions (program annotations) and demonstrate that this technique can be used to unveil common information leakage using current software verification tools.

This method is then compared with information leakage examples from current publications and current taint analysis benchmarks for detecting information leakage to demonstrate its effectiveness.

Motivation for this work is driven by a need for a formal method for defining and encoding leaks through program annotation, that leads to more effective future work into accurately detecting information leakage, both in malicious programs (malware) and unintentional leaking programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Information leakage can be defined loosely as any output from a computer program that provides information that should not be shared outside the program to an external observer. This type of information may include personal information such as passwords, GPS coordinates, pin codes, id numbers, names, addresses, and other personal details of users. If any of this data needs to be kept private within the program but is not, this is considered information leakage. When a program performs an information leak deliberately, it is called malware i.e. a malicious program.

Why information leakage is such a serious problem can be seen by considering the nature of data handled in many software systems. If a personal banking application on a mobile device leaks the pin code out to an observer, the security of the user's bank account is compromised and an attacker can potentially steal funds from them by logging in using that pin code. Similarly, if another program on a mobile device leaks the GPS coordinates of the device to outside observers, the privacy and security of the user is compromised by that leak and malicious actors can use that information to locate them.

Software is a complex and changing landscape and with many different approaches to handling information flow that can result in leakages both by malicious intent or by accident. Accuracy in detecting these leaks is essential, so programs with damaging leaks can be identified and stopped, while programs that do not leak are able to continue unhindered.

1.2 Motivation

As will be discussed in Chapter 2, a major problem with current methods of detecting information leakage is the inability to absolutely confirm that there are no leaks occurring in a given program. This is due in part to the lack of formal methods for defining leaks which results in many inaccuracies in the detection process. Current methods of identifying information leakage are often only concerned with the possible relation between private data and an output, and does not address whether the output actually provides any compromising information.

Current methods of leakage detection (such as taint analysis) can result in the incorrect identification of program outputs as leaks when they are not (false positives), or identifying them as not leaks when they are (false negatives). Examples of these cases will be discussed in Section 2.2 of Chapter 2.

Identification of false positives and false negatives in current leakage detection methods demonstrates the need for a more precise and formal way of defining information leakage before proceeding to check if leaks are occurring. To better assist the work of detecting information leakage in programs, this work will present a method for defining information leakage that can be used to specify what is considered a leak by developers at specific points where an observer can read the values of public variables.

Motivation for this work is driven by a need for a formal specification for defining and encoding leaks through program annotation will lead to more effective techniques to detect information leaks and malicious programs.

1.3 Goals

The goals of this thesis are to formally define what a leak is; show that this formal definition can be used to annotate programs with predicates that characterise information leakage; and demonstrate how to use software model checking techniques to automatically identify information leaks.

1.4 Overview of Thesis

Chapter 2 presents background on information leakage and malware detection. Chapter 3 provides basic definitions and demonstrates a method for formally defining leaks and annotating programs with the predicates that characterise them. Chapter 4 demonstrate the annotation technique in a range of examples addressing information leakage examples of different forms. Chapter 5 present the method for detecting leaks using software verification techniques and the Skink tool for static analysis of programs. Chapter 6 presents a literature review of current methods of malware detection through information leakage detection. Chapter 7 provides a conclusion and discussion of potential future work.

Chapter 2

Background

2.1 Information Leakage

Information leaks are the undesired output of information from a program that reveals the value of, or nature of, some private information handled by the program. This undesired output could be the direct output of private data in full, such as a password being sent out to a third party system or a private calendar entry being made public, to an output that provides derivative information, such as leaking that a bank account contains more than five hundred dollars.

It is important to note that some applications are not malicious but still unintentionally leak information as a by-product of carrying out their purpose.

Information leakage is described by Li et al. as the path flowing from the source of the sensitive information through an application and then back out of the application (a point usually referred to as the sink) [4]. However, what happens to that information along that flow can influence whether that information is still compromising by the end of the flow or if it gets missed by the tracking software before being sent out of the program.

Boreale investigated this issue when looking at quantifying information leakage in his example of the PIN-Checking function which explored whether or not the pin would be considered as leaked or not when multiple rejections of non-matching pins were given [5]. Boreale then defined two models for assessing information leakage, one where an attacker to the application is assumed to have full control and one where the attacker has a more limited control over the application.

The rest of this chapter outlines the basic features of taint analysis and discusses why there is a need for formal definitions of leaks with examples of information leakage highlighting this need.

2.2 Taint Analysis

A prominent method currently employed in the process of malware detection is taint tracking and analysis for identifying information leakage, both malicious and unintentional. Taint tracking involves marking (tainting) information which is considered private and should not be leaked outside of the application being analysed. Taint tracking can be defined as the process of observing the flow of the tainted pieces of information throughout the execution of an application, making note of what it influences and raising an alarm if the tainted data, or some important information about its nature, is leaked out of the program in some form.

The tainted information is tracked throughout the execution of the program, noting when it is transferred between variables or if it influences the formation of other information in a way that may still reveal nature of the original tainted information. According to this logic all variables affected are also tainted.

When public variables (which are variables that are shared outside the program) become tainted, information is leaked.

Taint analysis is dependent on the taint propagation rules it is based on which defines how taint propagates across variables during the program execution. These are dependent on factors such as, whether the tainted variable is passed along to a new variable, or whether a new variable is written due to that code's execution dependence on a decision using the tainted data. Different taint propagation rules used by different publications are discussed in Section 6.2.1 of the literature review in Chapter 6.

A simple example of a taint tracking scenario is a program that takes a variable containing tainted secret data. When that data is moved to a new variable, that new variable is also marked as tainted. The new variable is then leaked in some form, such as being printed on the screen, and a flag is raised by the taint analysis software to indicate that tainted data is being leaked.

There is no perfect implementation of taint tracking and propagation, all current implementations have cases which will result in over-tainting or under-tainting during tracking, which in turn lead to false positives and false negatives. Over-tainting is the result when an output is marked as a leakage of information when it is not. This is because of an issue in propagation rules that leads to the incorrect taint labels being assigned to variables that should not have them and ultimately results in a false positive for leaks. Under-tainting is the opposite case, where information leakage is not detected. This can be the result of the propagation logic not continuing to assigning taint to new variables as information moves through the system and ultimately results in a false negative for leaks.

Some methods even take conservative approaches of deliberately producing false positives to avoid false negatives. A prime example of where false positives commonly occur can be seen in the tools HybridDroid, FlowDroid and TaintDroid where all elements of arrays are considered to be sensitive when the data being tracked is put in just one of its elements [6, 7, 8]. This is done as a trade off of accuracy for minimising overhead during analysis.

Some analysis methods also allow false negatives when they consider those leaks to be inconsequential because they only leak a small part of the original secret information. Kang et al. state this directly as part of their approach when they present a program example that they believe demonstrates a case that does not require tracking [2].

These types of inaccuracies can be because some outputs of information may be considered leaks where in other cases they may not be. An example of such might be when information is obscured enough in the program before leaking that it might be considered by some to no longer be compromising to leak, but for others, any derivative output or the secret information is enough to count as a leak.

There are two main types of taint analysis, static and dynamic. Static analysis that is usually performed on the code without executing it, minimising risk but also taking up a lot more processing power and time as it generally concerns itself with all possible branches in the code and all possible inputs of data.

Dynamic analysis is performed alongside the program as it is executed and can be more efficient compared to static analysis as it only concerns itself with the path that execution actually takes and the data it actually uses, it does however incur an overhead for the program being analysed at run-time.

Sometimes combinations of both static and dynamic approaches are employed to benefit from features of both [9][2][10].

Perfect taint analysis will probably never be fully attainable. With static analysis, this is due to requiring to make approximations during analysis [11] or not being able to account for dynamic language features which can lead to false negatives [12]. With dynamic analysis, this is due to the inherent unsoundness of dynamic analysis, that is that the result of one run of dynamic analysis cannot be generalised.

2.3 Non-Interference and Other Approaches to Information Security

Non-interference is the case first presented by Goguen and Meseguer [13] of identifying information leakage focused on if changes in the private data in a program (often referred to as *high* security data) should not cause any obvious change in the state to the public data (*low* security) coming out. Often variables of these natures are referred to as h and l respectively in examples. This is similar to how private data and public data is represented in this thesis.

Language based security is the approach taken to enforcing security by implementing security rules at the language level. In a survey of the area, Sabelfeld and Myers discussed the various approaches to information flow security and the policies implemented to enforce them [14]. Foundational to this type of work is the work from Denning which focused on implementing static analysis to verify secure flow of information through systems [15].

2.4 The Contextual Impact of Leaks

Across the literature in taint analysis there is a trend of not precisely defining leaks beyond the assumption that a compromising leak exists if there is a possible flow of information between a source of private data and an output.

This work builds upon an informal definition of a leak as being any compromising output from program that provides information to an observer that should not be shared outside the program. Drawing from this, it is important to give leakage formal definitions that are specific to a program, its data, and its specific locations in the program where the information can be viewed by an external observer.

To illustrate this, take the example of a simple program on a smart phone that has access to the device's GPS coordinates. A program that handles GPS data can possibly afford to leak the first digits of each coordinate (displaying coordinates such as "4x.xxxx,4x.xxxx") as it might not even be possible to discern the country of origin from such general coordinates. But if the first two digits or more were leaked (such as "41.xxxx,41.xxxx"), then it might be possible to even discern not only the country of origin but the city too.

Kang et al. show that in constructing taint propagation rules, this determines what is and isn't labelled as a leak for taint analysis [2]. In consequence, some outputs can occur undetected that carry risk of revealing compromising features of the original information.

Schwartz et al. discussed how these rules can be configured for analysis of specific contexts by changing the taint propagation rules for different analysis cases [1]. They attempted to address this partially with their example of the "tainted jump policy" table defining what would and would not be defined as a case for propagating taint. This approach though is still not as granular as a check at a specific output point that defines what should and should not be allowed out at that point.

This is why it is important to define leaks formally in the context of the program being analysed. Taking into account the nature of the private data and how it is handled in a program can influence whether a leak actually has an impact in that program's context and help eliminate false positives and false negatives in analysis.

The next chapter outlines an approach to representing programs, specifying leaks, and identifying where leaks can occur in programs.

Chapter 3

Formal Definitions for Information Leakage

3.1 Introduction

To be able to detect leaks, there must first be a formal definition of what a leak is. There must also be formal definitions provided for how programs will be represented, how they handle sensitive data, and where leaks can occur. This chapter describes how control flow graphs (CFGs) are used to represent programs, how they handle private information with public/private variables in these programs, how these programs can leak information from these variables at locations called observation points, and how these leaks can be specified as predicates. The final section demonstrates how these observation points and leak predicates can be annotated onto the CFGs of programs.

3.2 Programs

In this thesis, programs are represented by their control flow graphs (CFGs), showing the flow between different locations in a program ($Prog$) made up of nodes and edges, $CFG(Prog) = (N_{Prog}, E_{Prog})$. The nodes ($N_{Prog}$) in graphs represent the different states or locations in programs, including what commands have been executed and what command is to be executed next. The edges (E_{Prog}) represent the paths between nodes in a program that are taken when the next command is executed. Such commands could include the assignment of an expression to a variable (such as $v_1 := 10 + 2$), or the choosing a path at a branch based upon a condition (such as $v_1 < 20$).

CFGs are a standard representation of programs and are used in this thesis with the goal of making the analysis of programs and their paths of execution easier. CFGs are used to map this method of leak detection to programs and demonstrate their effectiveness. This representation of programs and their clear description of a program's operation makes it easy to convert them to any programming language as required, as shown in Chapter 5.

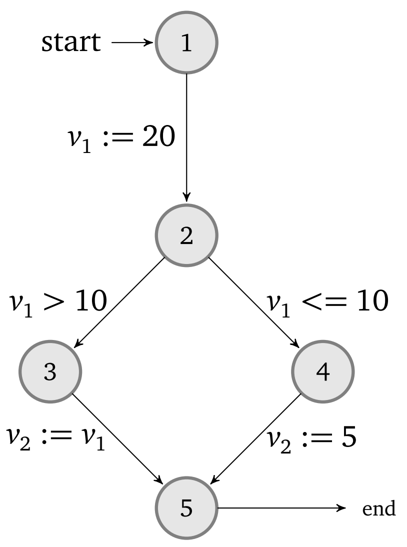

Figure 3.1: CFG for Program $Prog_1$

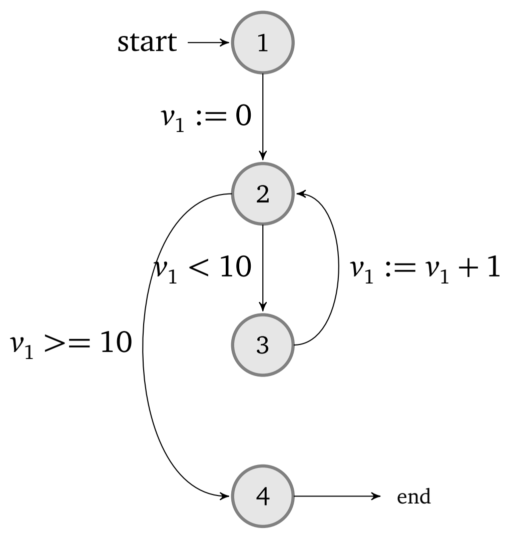

Figure 3.3: CFG for Program $Prog_2$

```
$v_1 := 20$
$if(v_1 > 10)$
        $v_2 := v_1$
$else$
        $v_2 := 5$
```

Figure 3.2: Program $Prog_1$

```
$v_1 := 0$
$while(v_1 < 10)$
        $v_1 := v_1 + 1$
```

Figure 3.4: Program $Prog_2$

Figures 3.1 and 3.3 provide basic examples of CFGs. Figure 3.1 shows a simple CFG that contains two paths based on a decision at location point 2. It starts by assigning the value of 20 to the variable v_1 and then checks to see if it is greater than 10. If it is greater than 10, the left path is taken and v_2 is then assigned the value of v_1. If it is not, the right path is taken and v_2 is assigned the value of 5. Figure 3.3 shows a simple loop in a CFG that assigns 0 to the variable v_1 and loops through the action of adding one to its value before finally exiting the loop after v_1 is found to be equal to or greater than 10.

Pseudo-code examples of these two CFGs are presented in Figures 3.2 and 3.4 to show how these programs would be written in a programming language. The indented code below the $if()$ and $while()$ commands represent the commands that are only executed while their conditions are true.

As part of these programs rules surrounding variables and how they can be used are added. The following sections outline what public and private variables are and how their nature influences the definition of leaks.

3.3 Public and Private Variables

Programming languages make use of variables, V, which are a set of memory locations that contain values of varying types including numerical, boolean and strings. To represent how programs handle private data, public and private variables are introduced here.

Public variables are a subset of all variables, $P \subseteq V$, that can potentially be observed by outside actors and be sent outside of the application. **Private variables** are a different subset of all variables, $S \subseteq V$, that should be kept secret from observers who seek to discern their value.

A private variable's contents can be transferred to a public variable but a variable itself cannot be both a public and private. Hence the intersection of the public (P) and private (S) variables is an empty set, ($S \cap P = \emptyset$).

Other variables within programs may be referred to as general variables like v_1 or as conditions such as c_1. In these cases, these variables are not public or private variables.

These definitions of public and private variables draw inspiration from the public and private variables seen in current popular programming languages such as Java and C# where the private variables within a class are not directly accessible from outside the class.

It should be noted that just like in the case of private variable data still being accessible via alternative means (such as the use of a $getPrivateData()$ function in the same class as the private data variable), the private variables in our definition can easily be worked around with the simple act of assigning of the private variable's data to a public variable ($p_1 := s_1$). This can be seen in the program discussed in Figure 3.5.

3.4 Information Leaks

Information leaks occur when the contents of a private variable either influence the contents of, or is assigned directly to, a public variable which is then observed by an outsider.

Leaks are formally defined in this thesis as a predicate which is true for all possible executions of a given program at a specific location. This predicate is specified as a logical formula that will be checked during static analysis of a program at a location where observation can occur. Doing so provides a method to prove precisely that there is a leak occurring at that point. The predicate is a logical formula on all variables within a program.

This method for defining leaks is different to the more standard and general definitions of leaks discussed in Section 2.1 of Chapter 2, this is because this thesis is focused on the detecting precisely leaks that occur consistently at a point in a program. More general leaks that can occur but don't occur consistently are not within the scope of the work in this thesis and can potentially be addressed in future work.

Take, for example, the program comprised of three commands in Figure 3.5 that has a private variable, s_1, that contains private data that should not be leaked, the contents of which is then assigned to a public variable, $p_1 := s_1$, followed by that public variable being readable to an observer in location 3. The predicate for defining the leak in that program is $s_1 = p_1$, where it holds true for all possible values of s_1 at the observation point (Section 3.5 will explain observation points in more detail). If the predicate is true at program point 3, then an external observer can use this knowledge to retrieve the value of s_1 by reading p_1.

This way of specifying leaks is focused on **consistent leaks**, that is, a leak which with each possible execution, the variable being read at the observation point has a reliable and repeatable way to be used to work out some compromising aspect of the private data. If there is one or more possible executions for which this is not the case then it is not possible to be considered a consistent leak because the external observer cannot consistently retrieve the secret information.

The main separating feature of the predicate annotation method compared to other methods like taint analysis for detecting leaks is that it is only concerned with the leaks that consistently occur at that given point in the program. Unlike the taint analysis method that tracks multiple flows of information, this method is specifically focused on checking for a consistent leak at the location where leaks are able to happen avoiding the extra work involved in tracking multiple flows of information to that observation point.

3.5 Observation Points

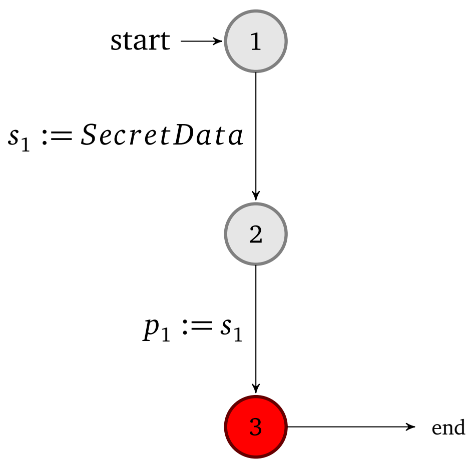

Figure 3.5: Simple Leaking Program

Observation points are locations in programs that allow for the observation of a public variable's contents from outside the program. These are the locations in programs where information from inside the program is made observable and hence where leakages of sensitive information can potentially occur.

An observation map $O : N_{Prog} \to 2^P$ is a map from all nodes to subsets of all public variables. An observation point is a node n where $O(n)$ is not an empty set ($O(n) \neq \emptyset$).

For easier readability in this thesis, observation points in a program CFG are represented by red nodes in the graph as demonstrated in Figure 3.5.

Figure 3.5 shows a program that assigns the value of some secret data to the secret variable s_1, the value from s_1 is then assigned to the public variable p_1, finally the value of p_1 is able to be observed at location point 3, $O(3) = \{p_1\}$.

Observations of a program are presented in an accompanying table. Table 3.1 shows how annotations are mapped to the control flow graph with the annotation on the left and the contents of the observation on the right. Observations that are empty are not included in the table.

Node	Observation
3	$\{p_1\}$

Table 3.1: Observation Map for Figure 3.5

3.6 Specifying Leaks Using Observation Points and Predicates

A leak predicate can be annotated to a location n in a program in the same way as an observation point with a leak predicate annotation. It is used to state the leak predicate in an annotation that should be always be true at that specific location within the program for a leak to be consistently occurring.

The leak predicate map $L : N_{Prog} \rightarrow 2^F$ is a map from nodes to subsets of leak predicate formulas F.

An example of this can be done for Figure 3.5 by annotating the leak predicate at location point 3 of $L(3)$ with $(s_1 = p_1)$ along with the observation point annotation. Table 3.2 shows the updated annotation table with the leak predicate added.

Node	Observation	Leak Predicate
3	$\{p_1\}$	$p_1 = s_1$

Table 3.2: Observation and Leak Predicate Maps for Figure 3.5

If the annotated predicate at program point 3 of Figure 3.5 is found to be consistently true for all possible paths through the program (as is the case for this program), then there is a leak confirmed to be occurring because an attacker can discover the value of the secret data based upon the contents of this observation point and the knowledge of the predicate.

3.7 Leaks Involving Calculation

There are cases of leaks which can be defined as the result of a calculation such that there is still a consistent way for every execution of the program for an observer to deduce relevant data from the output.

In Figure 3.6, the contents of the private variable, s_1, is passed to a public variable, p_1, while being modified through the addition of a constant, 5, to its value and then shared at the observation point at program point 3.

In this case the annotated leak predicate would be defined as $p_1 = s_1 + 5$ at program point 3. A more general definition of this leak would be the $\exists k$ s.t. $p_1 = s_1 + k$. Where k is the constant, 5, in this program.

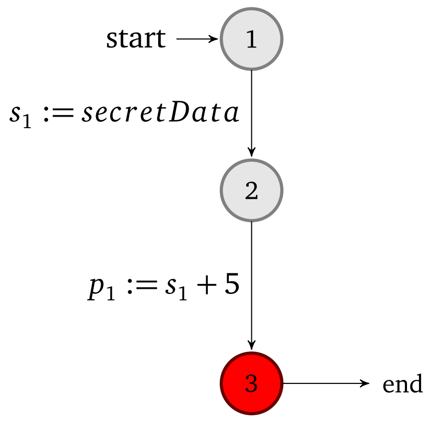

Figure 3.6: CFG of a Program Containing a Leak Involving a Calculation

Node	Observation	Leak Predicate
3	$\{p_1\}$	$p_1 = s_1 + 5$

Table 3.3: Observation and Leak Predicate Maps for Figure 3.6

The clear present limitation for this method of defining leak predicates is the limitation of defining then when they are rely on functions outside the analysis scope. At present these are not addressed in this thesis but is identified as an area required to be addressed in future work.

Another factor that is outside the current scope of this thesis is the complexity of these maps when more private variables and observation points need to be accounted for. This is discussed in the future work section of the final chapter of this thesis.

Figure 3.6 and Table 3.3 represent the simplest example of a leak involving a calculation of some form. Chapter 4 provides further examples where leak predicates need to account for more complex cases. Before progressing to these examples, there is a need to explain some assumptions made when presenting these program examples and their annotations.

3.8 Considering Observer Capabilities

In specifying observation and leak annotations, assumptions are made about the observer's capabilities in these scenarios, that is, if they can perform calculations upon the observed data and how much memory they have to store the observed data.

With Figure 3.5 there is the assumption that there is the minimum ability to at least observe and use the information attained at the observation point.

Other assumptions might be made in defining a predicate such as if they have the computing power to consistently perform an operation on the received data such that they can reconstruct the secret data from the observed value. This is the case shown in the Figure 3.6 and Table 3.3 where it is assumed that the observer can perform the simple addition/subtraction calculation to reconstruct s_1.

For the examples used in this thesis, the assumption is made that the observer can at least observe one item at an observation point and perform some simple computation upon it as presented in the corresponding leak predicate.

Chapter 7 will briefly discuss more complex considerations relating to observer capabilities in the future work section.

3.9 Summary

This chapter has presented a way for leaks to be formally defined as specific predicate annotations that are used to verify accurately if a leak is occurring. When the annotations of these predicates consistently hold true for all possible executions of a program being analysed, these leaks are proven to be occurring.

Chapter 4 demonstrates how this method can be applied to a variety of example programs, some of which are adapted from current published literature on information leakage. Chapter 5 discusses how to automate this analysis.

Chapter 4

Specifying Leakages with Program Annotations

4.1 Introduction

This chapter will demonstrate how the annotation technique is used to encode information leakage with program annotation in a range of examples and show how predicates are designed for each annotation case.

These examples are both original, and draw from current published literature showing different program cases of importance when detecting information leakage. These examples are intended to show how this method can be applied to accurately define leaks in different scenarios.

The benefit of annotating a CFG with a leak predicate is that the program and its annotations (predicates) are in a form that is compatible with current software model checking techniques and tools (which will be covered in Chapter 5).

4.2 Specifying Leaks for Explicit Flows

The example presented in Figure 3.6 for simple calculations in leaks demonstrated a case with a minor step involving a calculation on the value of s_1 before allowing the its leak through the observation of p_1. This is an example of a simple obfuscation, an act of trying to hide secret information's flow in the program before it is leaked. In this case it is done by changing its value in a way that is easily reversed after it is observed. This is a similar case to some of the examples presented by Schwartz et al. when discussing different cases where they need to track the flow of information to detect resulting information leaks [1].

These examples are included here to demonstrate how this predicate annotation method can be used to clearly identify leaks already being detected by other methods. This is done before going on to prove the method's accuracy in examples where other methods do not detect leaks accurately.

The examples have been adapted into the CFGs displayed in Figures 4.1 and 4.2 and have annotated to show how this method can be applied in these cases. In both cases the sensitive data being tracked was originally a user input, but in these examples it is a private variable being assigned some general value of $secretData$.

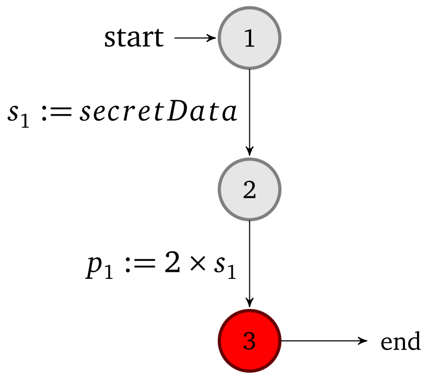

Figure 4.1: CFG of Example 1 from Schwartz et al. [1]

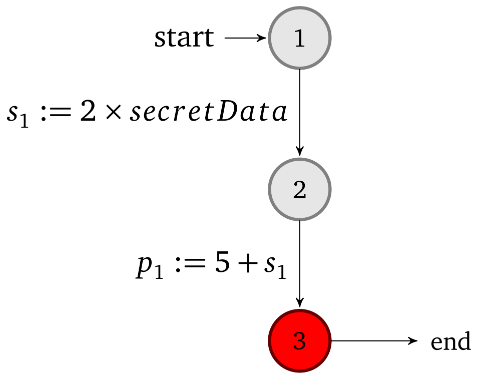

Figure 4.2: CFG of Example 3 from Schwartz et al. [1]

Node	Observation	Leak Predicate
3	$\{p_1\}$	$s_1 = \frac{p_1}{2}$

Table 4.1: Observation and Leak Predicate Maps for Figure 4.1

Node	Observation	Leak Predicate
3	$\{p_1\}$	$s_1 = \frac{p_1 - 5}{2}$

Table 4.2: Observation and Leak Predicate Maps for Figure 4.2

Figure 4.1 is an adaptation of Example 1 from Schwartz et al. which was originally solely focused in on the task of tracking the transfer of sensitive data to a new variable [1]. For analysis this example is extend by adding an observation point containing p_1 at the end of the program. This was added since that is when the flow of information ultimately becomes compromising. In this case the predicate for defining a leak here would be $s_1 = \frac{p_1}{2}$, or more generally as $s_1 = \frac{p_1}{k}$ where $k = 2$. The annotation table for this observation point and leak predicated can be seen in Table 4.1.

Figure 4.2 is an adaptation of Example 3 from Schwartz et al. [1]. Once again the input is replaced with $secretData$. Additionally, instead of leaking the value with a goto command, the observation point annotation at location point 3 is used. The annotation table for this observation point and leak predicated can be seen in Table 4.2.

These examples are some of the simpler cases for detecting leaks, in which the observer can see just one variable and make calculations from there. In Section 4.3 a program will be introduced which is used to show how leaks are defined in a slightly more complex scenario.

4.3 The Influence on Specifying Leaks From Observation Points

The contents of an observation point can drastically influence how a leak predicate is defined. The program in Figure 4.3 demonstrates how the variables the observer has the ability to read at an observation point influences how a leak predicate is defined.

In Figure 4.3, based on an arbitrary value or condition c_1 at location point 2, the program either adds or subtracts 5 from the value of the private data, s_1, before reaching an observation point at location point 5.

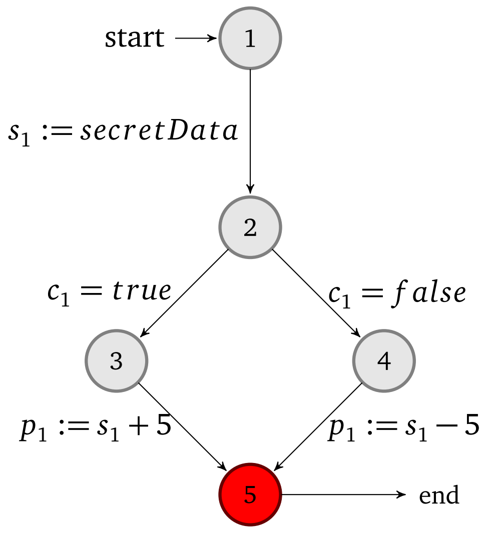

Figure 4.3: CFG of a Program Demonstrating How an Observation Point's Contents Influences Leak Predicate Definition

Node	Observation	Leak Predicate
5	$\{p_1, c_1\}$	$(c_1 \implies s_1 = p_1 - 5) \wedge$ $(\neg c_1 \implies s_1 = p_1 + 5)$

Table 4.3: Observation and Leak Predicate Maps for Figure 4.3 with c_1

Node	Observation	Leak Predicate
5	$\{p_1\}$	$(s_1 = p_1 + 5) \vee$ $(s_1 = p_1 - 5)$

Table 4.4: Observation and Leak Predicate Maps for Figure 4.3 without c_1

The leak predicate to be defined here is heavily influenced by what is being observed at the observation point.

Here if the observer is able to observe both p_1 and c_1 at location point 5 (such that

$O(5) = \{p_1, c_1\}$) they can fully reconstruct the value of s_1 using this information. The resulting predicate would be $s_1 = p_1 - 5$ if c_1 was true and $s_1 = p_1 + 5$ if c_1 was false. Or more simply put, $(c_1 \implies s_1 = p_1 - 5) \wedge (\neg c_1 \implies s_1 = p_1 + 5)$. The annotations for this case are presented in Table 4.3.

If the observer can only observe p_1 at the observation point (such that $O(5) = \{p_1\}$), it is not possible for them to fully reconstruct the value of s_1. Without knowing the condition c_1 at location point 2 the leak predicate can only be defined as either $p_1 = s_1 + 5$ or $p_1 = s_1 - 5$. The observer cannot know which path was taken and hence does not know which calculation was carried out when assigning the value to p_1. The observation and leakage annotations for the case where only p_1 is observed without c_1 is presented in Table 4.4.

In this example, the more variables an observer can view at an observation point, the more power they have to discern the value of s_1, as seen by the difference in leak predicates between Tables 4.3 and 4.4.

The example of the second leak predicate case shown in Table 4.4 shows how leak predicates can be constructed precisely for cases where a leak does not lead to the full reconstruction of s_1 but can still be considered compromising. Depending on the context, cases with outputs like these may be considered acceptable.

In both of these cases though, predicate annotation is used to precisely define what the leak predicate is and can be used to verify that the corresponding leak is occurring.

The next section will discuss an example of a leak that is considered acceptable by many, and hence considered inconsequential for tracking purposes.

4.4 Specifying Leaks for Implicit Flows

There are some cases where publications present programs involving leaks that they do not attempt to detect because they consider them to be inconsequential. This can be seen in the program from Kang et al. with an implicit flow that does not preserve enough of the secret information for them to consider it worthy to track [2]. This program has been adapted into a CFG in Figure 4.4.

Implicit flows are cases where the secret information is passed on by more indirect means as a result of control flow rather than a direct passing of secret information from one variable to another [16]. In this case, the information being passed through this implicit flow is preserving just a small detail about the secret information but other implicit flows can certainly preserve it fully (this can be seen later in the discussion around Figure 4.11).

The program they present only outputs a message revealing if the secret information was a value smaller or larger than 100. They did not consider the output of either "large" or "small" as a compromising leak for their taint tracking system to track due to it leaking such a small part of the program's secret information.

It is suggested that this kind of program could be seen as a compromising leak too, depending on the nature of the secret information being protected. In one scenario, the secret information could be a bank account balance and the leak could be enough to make someone a target for scamming or other cyber-attacks.

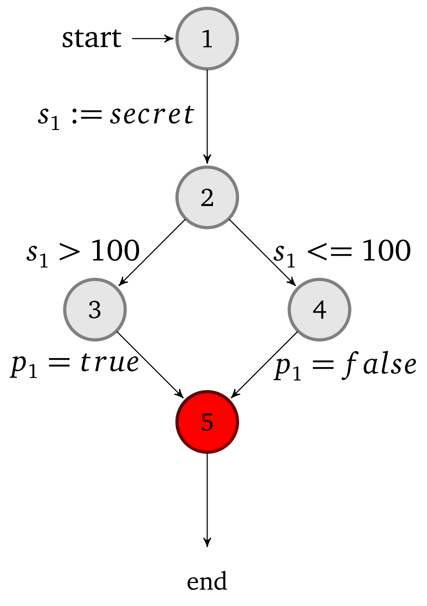

Figure 4.4: CFG of the small/large program from Kang et al. [2]

This example demonstrates how the predicate annotation method can be implemented to detect these forms of leaks due to the ability to exactly specify the leak predicate for that given observation point context.

The CFG in Figure 4.4 is simplified in some ways from the original example to be more compatible with the analysis techniques and coding language used in Chapter 5. The main change that has been made here is that instead of the observation containing a string saying "large" or "small" it is simplified it to a boolean variable being either true when it is larger than 100 or false if it is equal to or less than 100.

If there is concern about leaking information of if s_1's value being above or below 100, the leak predicate can be defined as $p_1 = (s_1 > 100)$ for the annotation of this program, which is used to verify that there is a consistent leak of this nature occurring. This annotation is presented in the annotations in Table 4.5.

Node	Observation	Leak Predicate
5	$\{p_1\}$	$p_1 = (s_1 > 100)$

Table 4.5: Observation and Leak Predicate Maps for Figure 4.4

This annotation example shows how we can be precise with the leak predicate annotation techniques used to detect leaks involving implicit flows that are usually ignored due to their minimal data preservation in the implicit flow. Section 4.5 demonstrates an example where different predicates can defined for different observation points, each crafted based on a different idea about what is considered a leak at that point.

4.5 Adjusting Leak Specifications for Specific Leakage Concerns

Section 4.4 discussed an example that some would consider to be an inconsequential leak and in Section 4.3 presented an example that could have different predicates depending on what was being observed. In this section a program is presented that can have different predicates specified for it depending on what we consider to be a compromising leak.

Figure 4.5 presents a CFG of a program that leaks the first two digits of a four digit pin code which is stored in array S_1 and leaked through the observation of array P_1 at location point 6. Figure 4.6 presents a pseudo code version of the program to help provide a better understanding of what is happening with the program with $observe(P_1)$ representing the observation point.

For this program, depending on the requirements surrounding the program's security this might be considered a leak or it might not, the PIN code is not useful unless the attacker has all four digits but they are closer to knowing the full PIN than they were before, and that could be enough to count this as a leak. In either case, the leak predicate can be defined precisely to verify if these leaks are occurring.

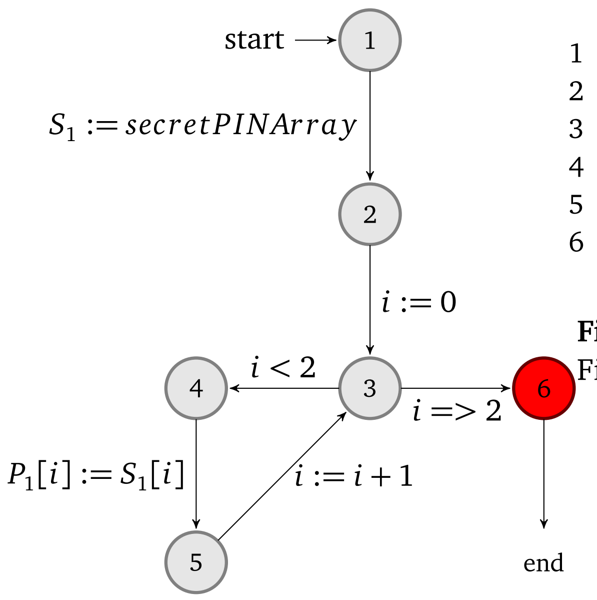

Figure 4.5: PIN Leak CFG

```
$S_1 := secretPINArray$
$i := 0$
$while(i < 2)$
    $P_1[i] := S_1[i]$
    $i := i + 1$
$observe(P_1)$
```

Figure 4.6: Pseudo Code for the CFG of Figure 4.5

Node	Observation	Leak Predicate
6	$\{p_1\}$	$(S_1[1] = P_1[1]) \wedge (S_1[2] = P_1[2])$

Table 4.6: Observation and Leak Predicate Maps for Figure 4.5

If there is a concern for leaking only those two first digits then a predicate could be specified as $(S_1[1] = P_1[1]) \wedge (S_1[2] = P_1[2])$ and in the case of this program this predicate would be found consistently true for all executions. This is the predicate included in the annotations in Table 4.6.

Alternatively, if the concern was only for the program in Figure 4.5 leaking all digits of the pin then the following predicate might be appropriate, where the intersection of the secret pin and the observed array is such that $(S_1[0] = P_1[0]) \wedge (S_1[1] = P_1[1]) \wedge (S_1[2] = P_1[2]) \wedge (S_1[3] = P_1[3])$. This predicate would be not true for all possible executions in Figure 4.5 and a consistent leak of the full pin code array is confirmed as not occurring.

This is a similar case to the GPS program described in Section 2.3 of Chapter 2 when discussing the contextual impact of leaks which could leak the first two digits of each coordinate. As demonstrated with the above program leaking the first two digits of a pin code, we could similarly construct a predicate to confirm a leak of the first two digits of each GPS coordinate. We could also similarly construct a predicate to check for just one digit of each coordinate or all digits as demonstrated with predicate for checking for a consistent full PIN leak.

These examples show how we can design leak predicate annotations which account for not only each observation point, but also account for the tolerance level of what we consider a leak for that location in its program context. Taint analysis methods of leakage detection do not provide this level of granularity.

4.6 Specifying Leaks via Loop Counters

Another case where leaks can be obfuscated from detection are within loops. The following program illustrated in Figure 4.7 contains a loop that is used to obfuscate the passing of a private variable's data to a public one before it is being observed. The purpose of this example is to show how it is possible to annotate leaks that are obfuscated this way by loops.

In Figure 4.7 there is an implicit flow with a loop that attempts to obfuscate the assignment of p_1 to the value of s_1. This is done incrementing p_1 as a loop counter until it is equal to the value of s_1, at which point the loop ends and p_1 is observed at location point 5.

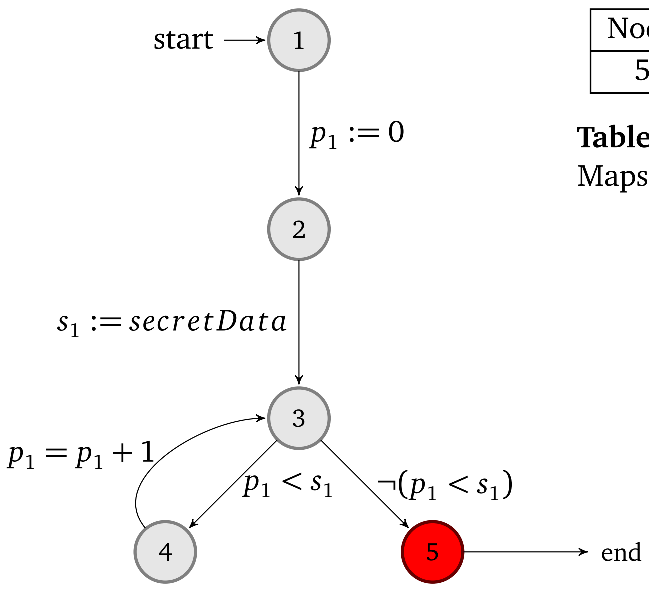

Figure 4.7: Leak Through a Loop Counter CFG

Node	Observation	Leak Predicate
5	$\{p_1\}$	$p_1 = s_1$

Table 4.7: Observation and Leak Predicate Maps for Figure 4.7

The leak predicate in this example would be the same leak predicate as in the first leak example from Figure 3.5 of $p_1 = s_1$. The leak predicate presented in Table 4.7 will hold true at location point 5.

Although the leak predicate here is simple to define, this leak can be hard to detect with information flow tracking because the variable being observed has never been directly assigned the value of the secret data but has instead been incremented to the value of the secret data through its use as a loop counter.

4.7 Annotating Examples From Benchmarks

Benchmarks for leakage detection exist which contain sets of programs addressing sets of different scenarios where programs can leak information. They are created so that leakage detection methods can be tested on them to discern their accuracy. There are a few benchmarks in current literature that have relevance to the predicate annotation method, such as JInfoFlow-bench [17], SecuriBench Micro [18], Droidbench 2.0 [3] and ICC-Bench [4], provide test cases for measuring leakage detection accuracy for taint tracking implementations.

Section 4.7.1 presents some example cases selected from the Droidbench 2.0 [7] and SecuriBench Micro [18] benchmarks adapted to CFGs to demonstrate how the annotation method can be used to detect leaks in these test cases and meet benchmark requirements.

4.7.1 Leaks Through Arrays Commonly Resulting in False Positives

The predicate annotation method has been demonstrated with an example involving arrays in Figure 4.5, but that example was not a type of case commonly resulting in false positives for some leakage detection methods. As discussed in Section 2.2 of Chapter 2, some taint tracking methods do not have the precise accuracy to confirm a leak is not occurring when the secret information is passed into an array and another piece of information is passed out.

The purpose of this thesis is to present predicate annotation as a way to improve the accuracy of information leakage detection, so it is important to demonstrate the method against cases well known for their common false positives.

Figures 4.8 and 4.9 present the CFG adaptations of ArrayAccess1 and ArrayAccess2 benchmark test cases from the DroidBench 2.0 benchmark [3]. Both programs are designed to test for the common occurrence of false positives caused by information flows involving arrays.

These two test cases are important because many taint tracking tools consider the whole arrays tainted when tainted data is added to just one item in the array and, as a result, any output of any value from the array will often be labelled as a leak when there is actually no leak occurring.

ArrayAccess1 is a case where an array is assigned three separate values, one being from the source of secret data, and then an observation is performed on one of the non-compromising pieces of data in the array. ArrayAccess2 is a case where an array is assigned two values, one from the source of secret data and one that isn't, and then the element from the array that is not compromising is accessed after an attempt to slightly obscure the index of the element being accessed using the function $calculateIndex()$.

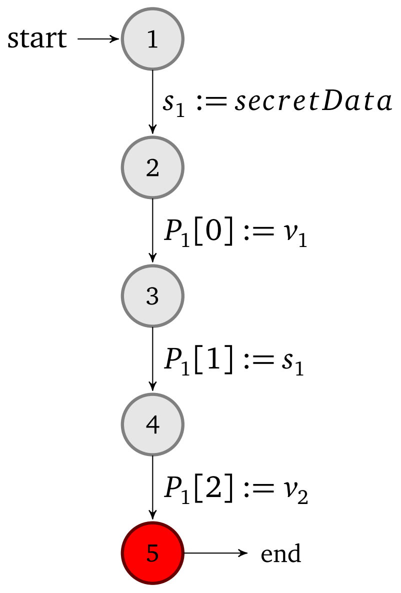

Figure 4.8: ArrayAccess 1 CFG

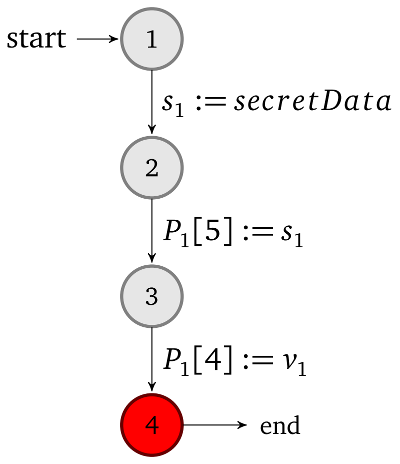

Figure 4.9: ArrayAccess 2 CFG

Node	Observation	Leak Predicate
5	$\{P_1[2]\}$	$P_1[2] = s_1$

Table 4.8: Observation and Leak Predicate Maps for ArrayAccess1

Node	Observation	Leak Predicate
4	$\{P_1[4]\}$	$P_1[4] = s_1$

Table 4.9: Observation and Leak Predicate Maps for ArrayAccess2

To verify that ArrayAccess1 is not leaking the value of s_1 the predicate $P_1[2] = s_1$ can be used to confirm that the output of $P_1[2]$ does not consistently share the value of the secret data. To verify that that ArrayAccess2 is not consistently leaking, the predicate $P_1[4] = s_1$ can be used to prove this. 4 being the consistent result of the $calculateIndex()$ function in the case of this program. In other scenarios it may still be necessary to define as $P_1[calculateIndex()] = s_1$ with the function still involved (this is still possible to verify using the verification tool used in the coming chapter on software model checking), or with another formula representing the result of the function if required.

With their corresponding predicates being annotated, both ArrayAccess1 and ArrayAccess2 would be found to be false and hence not consistently leaking. Tables 4.8 and 4.9 demonstrate how they can be annotated for analysis for leakage detection, and confirm that there are not consistent leaks occurring in these programs.

Using these test cases it can be demonstrated how the method of predicate annotation can avoid the false positives often occurring in current taint analysis methods.

These test cases hold similarities to the array test cases from the older SecuriBench Micro benchmark which is more focused on Java web applications and their potential security vulnerabilities [18]. There are 10 cases in this set of array test cases and demonstrating against each of them individually would take up too much space in this thesis. Therefore only a CFG adaptation and annotation table for SecuriBench Micro benchmark case Arrays1 in Figure 4.10 and Table 4.10 is presented so as to show a true positive case for detecting a consistent leak occurring through an array.

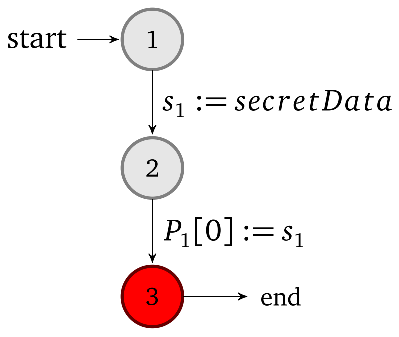

Figure 4.10: SecuriBench Micro Arrays1 CFG

Node	Observation	Leak Predicate
3	$\{P_1[0]\}$	$P_1[0] = s_1$

Table 4.10: Observation and Leak Predicate Maps for Arrays1

Figure 4.10 shows a program that assigns the secret data to the first element in an array and the observation point annotation shows that that element is then observed. The leak predicate to check for this is then defined as $P_1[0] = s_1$ to prove that there is a consistent leak occurring.

4.7.2 A More Complex Implicit Flow Benchmark

As discussed in Sections 4.4 and 4.6, implicit flows in programs can result in information leaks that are more likely to result in false negatives from many taint tracking methods. We have already discussed examples involving some form of implicit flow in the previous examples, but in this section seeks to prove our technique against a much more complex case.

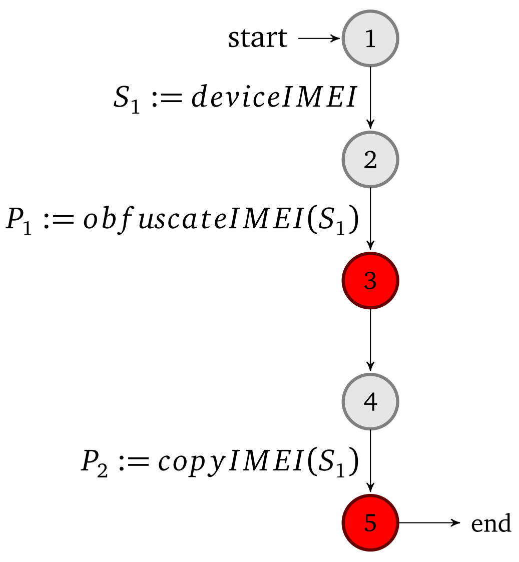

Figure 4.11: ImplicitFlow1 Test Case

The DroidBench "ImplicitFlow1" test case is used as a more complex example of a leak involving implicit flows [3]. This program tries to deliberately obfuscate the leak of the 15 digit identification code for a mobile device, called an IMEI, by obscuring its flow through the conversion from an integer to an array of characters and sending that array out. It then also converts the original IMEI to a character array and then makes an array using the ASCII values of those characters (e.g. "1" becomes 49) before compiling them back into a string and leaking the newly reconstructed IMEI out. This test case has been simplified and adapted to the CFG method in Figure 4.11.

In this example adapted into a CFG in Figure 4.11, the command of $obfuscateIMEI()$ contains a switch case that takes in the original IMEI and returns it converted into an array of characters where 0 becomes "a", 1 becomes "b", and so on. $copyIMEI()$ represents a function that converts the IMEI to a char array of the ASCII values of the numbers in the id code and returns a string of the IMEI reconstructed after the process.

The expectation of the benchmark in this test case is that the leak at the second of the two observation points, location point 5, will be the more difficult to detect due to the implicit flows involved. With the annotation of predicates introduced in this thesis, this becomes much easier to detect as the check of the output is now being done right before the observation point.

Defining predicates in this case becomes more complex when involving string and

Node	Observation	Leak Predicate
3	$\{P_1\}$	$(P_1[0] = S_1[0] + 97) \wedge (P_1[1] = S_1[1] + 97) \wedge$ $(P_1[2] = S_1[2] + 97) \wedge \ldots \ldots \wedge (P_1[14] = S_1[14] + 97)$
5	$\{P_2\}$	$(P_1[0] = S_1[0] + 48) \wedge (P_1[1] = S_1[1] + 48) \wedge$ $(P_1[2] = S_1[2] + 48) \wedge \ldots \ldots \wedge (P_1[14] = S_1[14] + 48)$

Table 4.11: Observation and Leak Predicate Maps for Figure 4.11

character variables but after treating the ASCII values of each character in a string as the integer of its ASCII value the predicates for each observation point can be easily defined. The leak predicate to check for the leak at the first observation point is $(P_1[0] = S_1[0] + 97) \wedge (P_1[1] = S_1[1] + 97) \wedge (P_1[2] = S_1[2] + 97) \wedge \ldots \ldots \wedge (P_1[14] = S_1[14] + 97)$ where 97 is the ASCII value of the character "a". The predicate for the second observation point would be $(P_1[0] = S_1[0]+48) \wedge (P_1[1] = S_1[1]+48) \wedge (P_1[2] = S_1[2] + 48) \wedge \ldots \ldots \wedge (P_1[14] = S_1[14] + 48)$ where 48 is the ASCII value of the character "0". Table 4.11 outlines the annotations for both observation points and their corresponding leak predicates.

Specifying these leaks with the leak predicates allows confirmation that there are leaks in this program for which other tools such as FlowDroid [7] return false negatives.

4.8 More Complex Predicates

This method of annotation can be successfully applied to program examples with increasing complexity, addressing common information leakage issues. Therefore, it is considered this will continue to apply in more complex program examples.

A more complex example could involve observations of encrypted data or cases where the leak is the result of a calculation using data observed at multiple observation points. While defining more complex predicates for outputs dealing with leaks like these is possible, these are outside the scope of this current thesis. This thesis focuses on methods for automating this detection. Discussion of more complex extensions of this method is included in Section 7.3 on potential future work in Chapter 7.

Chapter 5 focuses on how to automate the process of detecting these leak predicates and verifying if they are true for all executions using, Skink, the static analysis tool for software verification [19].

Chapter 5

Software Model Checking for Detecting Leaks

5.1 Using Skink for Static Analysis

This thesis demonstrates its method in action by using the static analysis tool for software verification, called Skink [19], which can confirm if a C program is running correctly based on conditions specified as assertions within the program being analysed.

Since Skink was developed to analyse C programs, the CFG examples in this thesis will be adapted to C code so as to be compatible for analysis. The following sections in this chapter will outline the process of adapting examples to C programs, analysing them, the accuracy of these cases, and comparing them to cases from other publications.

5.2 Adapting Examples to C Code for Verification

Take the simplest of the program CFGs presented in the previous chapter with Figure 3.5. A C code adaptation of the CFG is presented in Figure 5.1. Here s_1 and p_1 are defined as integers and the leak predicate annotation is now encoded with __VERIFIER_assert() at the location in the program where it was annotated in its CFG. Variables that can have different values every execution like s_1 are assigned non-deterministic values using the function __VERIFIER_nondet_int().

The program is only found to be correct by Skink when the predicate within __VERIFIER_assert() is always found to be true at that program point.

Analysing the program presented in Figure 5.1 with Skink returns that the predicate here, asserted at line 5, holds true for all possible executions, confirming that a consistent leak is indeed occurring in this basic example.

```
int main() {
    int s1 = __VERIFIER_nondet_int(); // secret
    int p1 = s1; // public

    __VERIFIER_assert(s1==p1) ;
}
```

Figure 5.1: C code for the program of Figure 3.5

5.3 Verifying Leaks Influenced By Observation Points

Figure 5.2 presents the code adaptation of the CFG from Figure 4.3 which had a leak that could only be deduced the original value of s_1 to a set of possible values. For the non-deterministic condition c_1 from the original CFG, it is also set to a non-deterministic boolean using the function __VERIFIER_nondet_bool().

The first leak predicate in this example was $(c_1 \implies s_1 = p_1 - 5) \wedge (\neg c_1 \implies s_1 = p_1 + 5)$ but predicates may need to be written differently to be compatible with the C environment for analysis, hence a predicate is rewritten to (c==(p == s + k))&&(!c==(p == s − k)) while still preserving the predicate's original definition.

Similarly the second predicate of $(s_1 = p_1 + 5) \vee (s_1 = p_1 - 5)$ was rewritten to (p == s + k)||(p == s − k) while preserving the original definition's function.

```
int main() {
    int s = __VERIFIER_nondet_int();; // secret
    int p; // public
    int k = 5;
    _Bool c = __VERIFIER_nondet_bool();

    if(c){
        p = s + k;
    }else {
        p = s - k;
    }
    __VERIFIER_assert((p == s + k)||(p == s - k));
    __VERIFIER_assert((c==(p == s + k))&&(!c==(p == s - k)));
}
```

Figure 5.2: C code for the program of Figure 4.3

Running the code with both assertions included from Figure 5.2 with Skink also returns that the asserted leak predicates holds consistently true for all possible executions.

5.4 Verifying Implicit Flow Leaks

Figure 5.3 is the C code adaption of Figure 4.4. The predicate is relatively unchanged in the adaption to its assertion in C code at line 13. Skink again verifies that the asserted leak predicate holds true for all possible executions, confirming that the resulting leak after this implicit flow is occurring.

```c
1  int main() {
2      int p; //public
3      int v1 = 1;
4      int v2 = 0;
5      int s = __VERIFIER_nondet_int(); //secret
6  
7      if (s > 100) {
8          p = v1;
9      } else {
10         p = v2;
11     }
12 
13     __VERIFIER_assert( p == (s>100) );
14 }
```

Figure 5.3: C code for the program of Figure 4.4

5.5 Verifying Leaks via Loop Counters

```c
int main() {
    int s = __VERIFIER_nondet_int(); // secret
    int p = 0; // public

    while(p<s){
        p = p +1;
    }

    __VERIFIER_assert(p == s);
}
```

Figure 5.4: C code for the program of Figure 4.7

Figure 5.4 is a C adaptation of the CFG from Figure 4.7 demonstrating the example which, as part of its process, incremented the value of p_1 and was leaked when its value was equal to s_1 by slightly obfuscated means. Again, when analysed by Skink, the assertion of the leak predicate $p_1 = s_1$ was found to be true in all executions where the observation point was reached, confirming a consistent leak in the program after the loop has run its course.

5.6 Verifying Leaks Adjusted for Specific Leakage Concerns

Figure 5.5 is the code adaptation of the PIN leak example form Figure 4.5 with some extra alterations, primarily that the program now leaks all four digits of the pin code and there are now three leak predicates being asserted. In the discussion of potential leak predicates for this program in Section 4.5, two predicates were originally discussed, based on the context of the program and the focus on what was considered to be a compromising leak.

```
 1  int main() {
 2      int S[4]; // secret pin
 3      int P[4]; //public
 4
 5      S[0] = __VERIFIER_nondet_int()%10;
 6      S[1] = __VERIFIER_nondet_int()%10;
 7      S[2] = __VERIFIER_nondet_int()%10;
 8      S[3] = __VERIFIER_nondet_int()%10;
 9
10      for(int i=0;i<4;i=i+1){
11          P[i] = 0;
12          int temp = S[i];
13          P[i]=temp;
14      }
15
16      //check for first 2 digits leak:
17      __VERIFIER_assert((S[0] == P[0])&(S[1] == P[1]));
18
19      //check for any 2 or more digit leak:
20      __VERIFIER_assert(2<=((S[0]==P[0])+(S[1]==P[1])+(S[2]==P[2])+(S[3]==P[3])));
21
22      //check for all 4 digits leak:
23      __VERIFIER_assert((S[0]==P[0])&(S[1]==P[1])&(S[2]==P[2])&(S[3]==P[3]));
24  }
```

Figure 5.5: C code for the program of Figure 4.5

In this C code three different predicate assertions are presented to show how they can be used to test for slightly different cases depending on what was could be considered to be a leak in that context. In the first predicate being asserted at line 17 of the code in Figure 5.5, (S[0] == P[0]) & (S[1] == P[1]), is used to check for the consistent leaking of **only the first two digits** of the PIN code. The second predicate line 20, 2 <= ((S[0]==P[0]) + (S[1]==P[1]) + (S[2]==P[2]) + (S[3]==P[3])), is used to verify if the program is **leaking two or more** of the PIN code digits. The final predicate at line 23, (S[0]==P[0]) & (S[1]==P[1]) & (S[2]==P[2]) & (S[3]==P[3]), verifies if the program leaks all four digits.

Analysing this program with Skink finds that all three predicates hold true since the program leaks all four digits of the PIN code array every time. This shows how differently designed predicates can be used to test for slightly different leakage concerns.

Other example cases from current literature exist, such as the coming section on taint analysis benchmarks which are written in Java, where the code being tested is adapted from another programming language not presently compatible with Skink. In these cases, the programs have been manually translated to C code to aid in analysis while still working to maintain the purpose of these benchmark programs.

5.7 Verifying Java Based Benchmark Cases in C Code

DroidBench 2.0 is a benchmark for testing taint analysis methods for Android Java applications [3]. To prove the effectiveness of the leakage detection method in Skink against these benchmarks, these examples were adapted to C code for annotation and analysis. It must be noted that many of these test cases are built around features of the object oriented Java language and elements specific to the Android environment, as a result not all benchmarks are ideal for conversion to C and are left for future work when potentially applying this analysis method directly to Java code.

```java
public static String[] arrayData;

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_array_access1);

  arrayData = new String[3];

  arrayData[0] = "element␣1␣is␣tainted:";
  arrayData[1] = ((TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE))
    .getDeviceId(); //source
  //arrayData[2] is not tainted
  arrayData[2] = "neutral␣text";

  SmsManager sms = SmsManager.getDefault();

  //no data leak: 3rd argument of sendTextmessage() is not tainted
  sms.sendTextMessage("+49␣1234", null, arrayData[2], null, null);  //sink, no leak
}
```

Figure 5.6: Original Java code for ArrayAcces1 from DroidBench 2.0 [3]

Figure 5.6 presents the original Java code from the Droidbench 2.0 ArrayAccess1 test case and Figure 5.7 is the C code adaptation of the test case made for compatibility purposes. Here since the focus is on tracking the tainted elements in arrays, C does not have string type and only character arrays which would result in a multi-dimensional array if it was attempted to replicate the act of using strings as array elements. Since there is a whole separate test case in the same benchmark set for leaks involving multidimensional arrays, this example works with integer values instead to avoid adding extra complexity when adapting the code.

```
int main() {
  int s = __VERIFIER_nondet_int();
  int P[3];
  int v1 = __VERIFIER_nondet_int();
  int v2 = __VERIFIER_nondet_int();
  P[0] = v1;
  P[1] = s;
  P[2] = v2;

  __VERIFIER_assert(P[2]==s);
}
```

Figure 5.7: C code for the program ArrayAcces1

```
int main() {
  int s = __VERIFIER_nondet_int();
  int P[5];
  P[0] = __VERIFIER_nondet_int();
  P[1] = s;
  P[2] = __VERIFIER_nondet_int();
  P[3] = __VERIFIER_nondet_int();
  P[4] = __VERIFIER_nondet_int();

  __VERIFIER_assert(P[calculateIndex()]==s);
}

int calculateIndex(){
  int index = 1;
  index++;
  index *= 5;
  index = index%10;
  index += 4;

  return index;
}
```

Figure 5.8: C code for the program ArrayAcces2

Figures 5.7 and 5.8 present the C code of the two benchmark cases for which annotations were discussed alongside the CFGs of their programs in Figures 4.8 and 4.9. Here the other elements in the arrays are also assigned non-deteministic integers to confirm that their values do not affect the outcome in the assertion.

For the assertion for Figure 5.7 is simply written as P[2]==s, whereas the assertion for Figure 5.8 can be written as P[calculateIndex()]==s using the index obfuscation function of calculateIndex() in the predicate definition.

Analysing both of these C code adaptations in Skink return false confirming that there is not a consistent leak occurring in this example program.

```
int main() {
    int s = __VERIFIER_nondet_int(); // secret
    int P[10]; // public
    P[0] = s;

    __VERIFIER_assert(P[0]==s);
}
```

Figure 5.9: C code for the program Arrays1

Figure 5.9 presents the C code adaptation of the simple Arrays1 benchmark case. Here it is asserted with the predicate annotated in Table 4.10 to verify that the observation of the array element P[0] at that point will result in a consistent leak.

5.8 Complex Implicit Flows

An example of a more complex case from the DroidBench 2.0 benchmark is that of the ImplicitFlow1 case originally discussed alongside Figure 4.11. Cases involving implicit flows are not detectable by taint analysis tools like TaintDroid due to their difficulties if they were attempting to track them usually leading to an overabundance of false positives [8].

The adapted and annotated C code of this test case is displayed in Figure 5.10. Here some changes were made to adapt the original test from Java to C code. The biggest change is the incorporation of separate functions to be inline within the main function to facilitate analysis, treating strings as character arrays due to the lack of string variables in C. As a result, this test case has some similarities to the PIN leak example from Figure 5.5, in particular the definition of predicates using arrays.

It is important to note that the secret id number in this program was changed from the 15 digit id array to a 4 digit id array. This is done purely for presentability and minimising the space in this thesis that the predicates and code will take up for addressing the array of such length. The method is demonstrated with the 4 digit array knowing that this can easily be extended to a 15 digit one.

Here defining predicates became more complex but could be worked out by testing with extra non-deterministic values assigned to k. In the obfuscation in this test case, each digit is changed to a corresponding letter. The original test predicate designed to find the relationship between each digit and its corresponding output letter was to

```
int main() {
    int S[4];
    int P1[4];

    S1[0] = __VERIFIER_nondet_uint()%10;
    S1[1] = __VERIFIER_nondet_uint()%10;
    S1[2] = __VERIFIER_nondet_uint()%10;
    S1[3] = __VERIFIER_nondet_uint()%10;

    __VERIFIER_assume(S1[0]<10&&S1[0]>0);
    __VERIFIER_assume(S1[1]<10&&S1[1]>0);
    __VERIFIER_assume(S1[2]<10&&S1[2]>0);
    __VERIFIER_assume(S1[3]<10&&S1[3]>0);

    for(int i = 0; i<4; i++){
        switch(S1[i]) {
           case 0:
              P1[i] = 'a';
              break;
           case 1:
              P1[i] = 'b';
              break;
           case 2:
              P1[i] = 'c';
              break;
           case 3:
              P1[i] = 'd';
              break;
           case 4:
              P1[i] = 'e';
              break;
           case 5:
              P1[i] = 'f';
              break;
           case 6:
              P1[i] = 'g';
              break;
           case 7:
              P1[i] = 'h';
              break;
           case 8:
              P1[i] = 'i';
              break;
           case 9:
              P1[i] = 'j';
              break;
        }
    }

    //Check for first obfuscated
    //leak and observation point here:
    __VERIFIER_assert(S1[0]+'a'==P1[0]);
    __VERIFIER_assert(S1[1]+'a'==P1[1]);
    __VERIFIER_assert(S1[2]+'a'==P1[2]);
    __VERIFIER_assert(S1[3]+'a'==P1[3]);

    //convert integers to ASCII values since
    //C does not have casting like in JAVA
    int P3[4];
    P3[0] = S1[0]+'0';
    P3[1] = S1[1]+'0';
    P3[2] = S1[2]+'0';
    P3[3] = S1[3]+'0';

    int numbers[58] = {0,1,2,3,4,5,6,7,8,9,
            10,11,12,13,14,15,16,17,18,19,
            20,21,22,23,24,25,26,27,28,29,
            30,31,32,33,34,35,36,37,38,39,
            40,41,42,43,44,45,46,47,48,49,
            50,51,52,53,54,55,56,57};
    int P2[4];

    for(int i = 0; i < 4; i++){
        int tmp = numbers[P3[i]];
        P2[i] = tmp;
    }


    __VERIFIER_assert(S1[0]+48==P2[0]);
    __VERIFIER_assert(S1[1]+48==P2[1]);
    __VERIFIER_assert(S1[2]+48==P2[2]);
    __VERIFIER_assert(S1[3]+48==P2[3]);
}
```

Figure 5.10: C code Adaptation ImplicitFlow1

assert that $S_1[1] + k \neq P_1[1]$. This test found that when k was equal to 97 this assertion would fail, 97 being to be the ASCII value of the character "a". Taking this information, the predicate S1[1]+97 = P1[1] (or it could even be written as S1[1]+"a"= P1[1] in the C code since char variables will be treated as their integer ASCII values in the calculation) was asserted to identify that in all executions this predicate held true and a consistent leak was occurring for that digit of the id number. Extending this predicate to check all four digits proved that there was a consistent leak occurring for the whole id number. A similar method was used to construct the predicate for the second observation point resulting in the predicates listed below.

Leak Predicate 1:

(S1[0]+97 = P1[0]) & (S1[1]+97 = P1[1]) & (S1[2]+97 = P1[2]) & (S1[3]+97 = P1[3])

Leak Predicate 2:

(P1[0] = S1[0]+48) & (P1[1] = S1[1]+48) & (P1[2] = S1[2]+48) & (P1[3] = S1[3]+48)

These predicates are broken down into separate assertions, as seen on lines52-55 and 79-82, for the purpose of readability. This provides the same result during analysis as if they were still one whole predicate since Skink will only return true for this program if all assertions within it are found to be true.

5.9 Comparison Alongside Taint Analysis Accuracy

DroidBench [3]	Leak	FlowDroid	Predicate Annotation
ArrayAccess1	No	FP	TN
ArrayAccess2	No	FP	TN
ArrayCopy1	Yes	N/A	TP
MultidimensionalArray1	Yes	N/A	TP
Loop1	Yes	TP	TP
Loop2	Yes	TP	TP
UnreachableCode	No	TN	TN
ImplicitFlow1	Yes	FN	TP
ImplicitFlow2	Yes	FN	TP
SecuriBench Micro [18]	Leak	FlowDroid	Predicate Annotation
Arrays1	Yes	TP	TP
Arrays2a	Yes	TP	TP
Arrays2b	No	FP	TN
Arrays3a	Yes	TP	TP
Arrays3b	No	FP	TN
Arrays4	Yes	TP	TP
Arrays5	No	FP	TN
Arrays6	Yes	TP	TP
Arrays7	Yes	TP	TP
Arrays8a	Yes	TP	TP
Arrays8b	No	FP	TN
Arrays9	Yes	TP	TP
Arrays10a	Yes	TP	TP
Arrays10b	No	FP	TN

Table 5.1: Comparison on Taint-Analysis Benchmarks
TP: True Positive FP: False Positive
TN: True Negative FN: False Negative

The annotation method was tested against other benchmark cases to show leaks can be accurately detected where other methods often get false positives and false negatives.

Table 5.1 lists some taint analysis benchmark cases used from DroidBench 2.0 [3] and SecuriBench Micro [18] and shows where the method presented in this paper clearly determines whether or not a leak is occurring alongside FlowDroid [7]. Predicate annotation is compared to FlowDroid, since it was presented with the DroidBench benchmark and has been tested against both DroidBench and SecuriBench Micro.

Their results are recorded as true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). There are two cases for FlowDroid where their results are listed as N/A, this is because results weren't available in publications for how it performed in those tests.

5.9.1 Benefits Compared Taint-tracking Methods

A major benefit of using predicates and assertions in annotations for leakage detection as opposed to taint tracking methods is the removal of the need to track the each individual step along the way making the decision at each stage whether the tracking should continue for the pieces of derivative information generated at each point. Using our method, the checking can be done at the point of observation, affording a greater accuracy at those points as reflected by some of the results listed in Table 5.1.

Compared to taint analysis methods, use of leak predicate assertions can reduce the instances of unnecessary false positives. Take the example of the PIN code program discussed earlier in Figure 5.5 and the multiple predicates presented for different leakage concerns. Under normal taint analysis methods, all outputs discussed would result in being being labeled as leaks where as with a more specific leak predicate definition provided in this thesis's method would result in only the compromising one being considered a leak.

The other major benefit is the elimination of a variety of false positive cases connected to information flowing through arrays and implicit flows. Some taint analysis tools deliberately do not try tracking implicit flows (TaintDroid [8]) or do attempt to do so resulting in extra false positives (DTA++ [2]). The benefits of using predicate annotation in these cases have been demonstrated through a variety of examples in this and the previous chapter.

5.9.2 Drawbacks Compared to Taint-tracking Methods

The biggest drawback of the predicate annotation method in its present form is the required pre-knowledge and understanding of the program's leak so as to be able to construct a leak predicate.

This predicate annotation method requires the clear definition of leaks as predicates based on the program's context. If a predicate is not specified correctly, it may fail to identify a leak correctly and therefore produce a result stating there is no consistent leak for that predicate, when a correct predicate is specified it would return that there was a consistent leak. For example, the leak predicate defined in Table 3.3 is $p_1 = s_1 + 5$ and correctly identifies the leak in program in Figure 3.6, but if the incorrect leak predicate was defined for this program as $p_1 = s_1$ then the predicate would never hold true with no leak being detected.

Taint tracking has similar drawbacks when it comes to defining taint propagation rules, if the rules aren't defined correctly, leaks will be missed. The difference here is that these rules can be defined once for whole program analysis, predicate annotation requires predicates to be designed for every observation point.

Predicate annotation method cannot confirm both true positives and true negatives at the same time for programs that have multiple observation points where one leaks consistently and the other does not. If one assertion in Skink returns false, the whole program's analysis returns with a false.

Taint analysis methods can identify both types cases in one analysis (depending on the quality of their taint prorogation rules) as they only flag leaks when a flow of sensitive information reaches an output, any other output is not considered a leak.

5.10 Conclusion

This chapter demonstrated how, using the software model checking tool Skink [19], analysis of programs with leak predicate assertions can be automated to confirm if a program is consistently leaking. Chapter 6 provides a literature review of related work in malware detection and taint tracking before final conclusions in Chapter 7.

Chapter 6

Literature Review

6.1 Introduction

This chapter presents a literature review of current related works. It explores the current state of taint analysis and other malware detection methods (including signature detection and machine learning), examining their accuracy and some of the benefits and drawbacks from these approaches.

6.2 Taint Analysis

6.2.1 Taint Propagation Logic

One of the first implementations of data tainting was in the Perl programming language that allowed for the basic tracking and would throw fatal errors if tainted data was leaked [20]. Early work in taint analysis can be traced back to the 1980s with Ferrante et al.'s work into data and control dependencies, graphing out what pieces of data and sections of code are dependent on preceding lines of code [21].

In more recent research, work has continued to improve the process of tracking tainted data throughout the execution of a program, by implementing a range of new rules and methods for tracking more complex flows of information.

Schwartz et al. defined a series of rules for taint propagation in a dynamic analysis but observed that control dependencies could not be accounted for in dynamic analysis alone, as dynamic only concerned itself with the code path that was executed and not the others that were not taken by the application in that execution [1].

Bao et al. introduced the use of strict control dependencies, the idea that there are some control dependencies that should not result in the tainting of another variable and helped to remove cases of over-tainting arising from using more basic control dependencies [22].

Kang et al. observed cases within the strict control dependencies base rules from Bao et al. that they worked to correct by presenting their system that worked to identify unlabelled under-tainting cases when analysing an application and then generated it's own taint propagation rules to account for these cases in analysing that application [2].

For Android systems specifically, Enck et al. presented TaintDriod which looked at four levels for taint propagation (variable, message, method and file) to more effectively track taint in the smart phone environment [8]. Zhang et al. presented VetDroid that had the added benefit of identifying information leakage relating to the devices network state (which TaintDroid did not detect) [23]. Tam el at. observed that TaintDroid and VetDroid both had the drawback of requiring heavy modification of the Android OS to implement their own methods. Tam el at.'s implementation, CopperDroid, focused on using behavioural profiles instead to identify malware [24].

Graa et al. also sought to reduce under-tainting in the android environment and did so by identifying cases such as leakages that occur when a variable is not changed, leaking that another branch of code was taken upon a decision using the private data [25]. Extending upon this work in a later study in 2014, they explored how to detect further obfuscation attacks used to hide information leakages and introduced new rules to propagate taint thorough these cases [26]. In both cases, the focus was purely on the more pressing need to reduce under-tainting rather than reduce over-tainting.

A method of using static taint analysis as part of a malware signature detection was employed by Feng et al., however, they noted they were not fully prepared for obfuscation methods [27]. Not being prepared for obfuscation could lead to under-tainting.

Arzt et al. in 2014, presented FlowDroid which claimed to take into account the context of input data during its taint analysis. They demonstrated how it could be used to detect leaking apps and malicious apps from multiple sources, along with their own benchmark set that they developed called DroidBench which they used to test FlowDroid against various key leakage scenarios [7].

You el at. looked into identifying implicit flows in the Android environment and demonstrated how they could be exploited, leaving a window for future work in detecting and defending against these cases [28]. Li et al. presented their own inter-component communication information leakage detection application, IccTA, which used static analysis methods and was tested against their own set of benchmarks they created [4], extending from DroidBench [7].

Table 6.1 compares different approaches to tracking taint and how they perform specifically in regard to over-tainting and under-tainting based on their own observations (stated). Another column for unstated under-taint is added based on evaluations from other papers or based upon simple observation of the propagation rules, to list cases of under-taint that are not mentioned in the original papers. If no identified unstated under-taint exist, the entry is listed as "unclear" as there is the potential for still undiscovered cases to exist.

Paper	Year	Stated Over-taint	Stated Under-taint	Unstated Under-taint
Bao et al.[22]	2010	No	No	Yes
Schwartz et al. [1]	2010	No	No	Yes
Kang et al [2]	2011	Yes	Yes	Yes
Graa et al. [25]	2013	Yes	No	Unclear
Enck et at. [8]	2014	Yes	No	Yes
Graa et al. [26]	2014	Yes	No	Unclear
Feng et al. [27]	2014	No	Yes	Unclear
Arzt et al. [7]	2014	Yes	No	Unclear
Wei et al. [29]	2014	Yes	No	Unclear
Lee et al. [6]	2016	Yes	No	Unclear
Fu et al [30]	2018	Yes	Yes	Unclear

Table 6.1: Taint Propagation Logic

6.2.2 Dynamic Taint Overhead

A complication that comes with dynamic taint analysis is the overhead it brings to the execution of the application being analysed. Overhead being the extra processing power required alongside to carry out the analysis.

Moore & Chong presented a way to minimise overhead by using preliminary static analysis to identify areas where it is no longer necessary to track information [10].

Taint analysis implementations, including HybriDroid [6], FlowDroid [7] and TaintDroid [8], have taken a different approach to minimise overhead in their dynamic analysis by allowing for over-tainting, by tainting whole arrays when a value within them are tainted. This is done to avoid dealing with the overhead incurred with tracking different taint tags for each entry within the array in the Java environment [8].

Ming et al. took a new approach again to tracking taint by making use of multi-core processing but depended on straight-line code which had a small amount of variables that it needed to track at runtime [31]. The same team presented StraightTaint in 2016 which minimised overhead by making use of offline symbolic taint analysis [32].

Table 6.2 outlines the overhead percentages of dynamic taint analysis methods that supply an overhead percentage in their articles.

Paper	Environment	Claimed Overhead
Bao et al [22]	N/A	76%
Zhang et al [23]	Android	32.294%
Enck et al. [8]	Android	14%
Wei and Lie [33]	Android	10-20%
Fu et al. [30]	WebAssembly	5-12%

Table 6.2: Dynamic Taint Analysis Overhead

6.3 Signature Detection

Signature detection methods are being used in malware detection and are the process of generating signatures that represent applications or segments of applications based on specific features, such as code structure, behaviour or other identifiable aspects. These can then be used to compare against signatures that have already been generated for already existing applications that are known to be malicious. If there is a match then the new application is also identified as malicious and appropriate action can be taken.

6.3.1 Signature Detection and Code Obfuscation

Since changing one feature can lead to a new signature, a variety of methods can be employed to obfuscate code including; inserting redundant code, shuffling code ordering, replacing functions with new ones that still get the required result, and much more [34].

Apposcopy presented a way of using taint tracking to develop an application signature based on the semantic nature of its code instead of the syntactic, as a way of getting around some forms of obfuscation [27]. However, it was pointed out by others that Apposcopy was still vulnerable to the more complex forms of obfuscation [35].

Schrittwieser et al. performed an analysis of the existing methods for obfuscation, and obfuscation detection finding techniques that relied on pattern recognition, like in signature detection, were still highly susceptible to obfuscation [36].

Wang and Rountev later presented techniques for detecting obfuscated code in Android applications by the application of machine learning techniques to train a model for identifying features of the application that would indicate that it had undergone some form of obfuscation process [37].

6.3.2 Machine Learning in Signature Detection

Many of the existing implementations of machine learning take various ways of using the API calls of an application to generate signatures for training [38, 39, 40, 41, 42].

Yuan et al. highlighted the effectiveness of machine learning methods in the detection of malware that is the repackaging of existing malware features and identified how current popular anti-virus software did not handle repackaging as well [43].

Mariconti et al. presented MAMADroid, a method that involved constructing Markov chains of abstracted API calls as representation of application behaviour, both benign and malicious, then training a machine learning model to identify malware on them and then demonstrated how effective it was in detecting newer forms of malware without any new training over different year sets of training data [44].

6.4 Evaluation

6.4.1 Taint Propagation

At present, no taint analysis method has demonstrated a way to propagate taint that results in 100% accuracy. All cases seem to result in over-tainting or under-tainting.

Sometimes these issues are deliberately for the sake of reducing overhead, such as the case of array over-tainting discussed in Section 6.2.2 on dynamic taint overhead.

Many current taint analysis methods rely on the manual definition of sources and sinks which leaves analysis vulnerable to the introduction of new unforeseen sources and sinks. Machine learning methods have been employed, as seen in SUSI by Rasthofer et al., to try and account for the introduction of new taint sources and sinks that could be introduced through new or unfamiliar APIs [45].

6.4.2 Machine Learning & Signature Detection Methods

A major issue that is apparent from the methods involving machine learning and signature detection is that of new methods of malicious activity for which there is no current signature built or machine learning model trained on, either as a result of deliberate obfuscation or normal replacement of old API functions with new ones.

It is demonstrated in some cases that it is possible to protect against new malware, as demonstrated by Mariconti et al. with their malware detection system, MAMADroid, but it still demonstrated a drop in detection accuracy across each following year of malware samples after training on a given year's training data [44].

A clear focus for improvement is the vulnerability to the new signatures not seen before. Significant work has been done to account for this but it is clear that further work is required to improve accuracy duration.

6.5 Conclusion

In this literature review, the current state of taint analysis methods for information leakage (both malicious or otherwise) and other methods of malware detection based on signature detection has been discussed.

The current state of taint propagation logic was reviewed and the still present inaccuracies of over-tainting and under-tainting that exist in current solutions, resulting partially from issues such as overhead and focusing on efficiency, was identified. Signature detection and machine learning methods in malware detection were considered. Challenges faced in the area of identifying new threats that do not share features with samples of malware already detected and categorised continue to be a concern.

Chapter 7

Conclusion

7.1 Thesis Summary

This thesis has demonstrated how the problem of information leakage is best addressed through the development of formal definitions of what constitutes a leak. The method of specifying leaks through the use of predicate annotation is shown to allow for the verification of whether leaks are occurring in a program when used with current software model checking tools (such as Skink).

This method's effectiveness has been demonstrated against constructed examples, examples from published literature and taint analysis benchmark cases.

Working with examples from current taint analysis literature show how the technique could be applied in these cases to generate the same results, and potentially detect leak cases that other methods considered not to be worth the effort.

Examples of programs have been provided that could be annotated with different leak predicates, depending on factors such as what was being observed at an observation point and what could be considered a leak for a program in that specific context.

When looking at benchmarks, there was a focus on demonstrating predicate annotation with examples in which current methods commonly result in false positives and false negatives due to complications in their techniques or trade-offs made for efficiency purposes.

The benefits of the predicate annotation method can be seen in the removal of the need for step by step tracking of secret information through a program's flow and the potential errors that come with that process. These benefits the of annotation method address issues with taint tracking methods that result in false positives with arrays and false negatives with some implicit flow cases.

Drawbacks with the method currently include the effort that can be involved in designing predicates for each given scenario and how false negatives and false positives can result if predicates are not defined precisely.

7.2 Final Conclusions

This thesis has demonstrated how information leakage can be addressed with formal definitions of leaks and demonstrated a way to do so with leak predicates and program annotation. This provides a precise and accurate way to specify leaks in their given contexts and verify that they are occurring using embedded definitions within the program code.

The predicate annotation method has been proven in a variety of scenarios where its benefits over taint analysis methods have been shown to avoid common false positive and false negative scenarios.

This thesis provides a basis for more accurate leakage detection that can now be applied in other contexts. There is further work that can be done on this technique to make it easier to define leaks in different scenarios of much more intense complexity. This thesis provides the groundwork upon which this future work can build upon.

7.3 Potential Future Research

The application of the predicate annotation method to a range of actively used programming languages is an important focus for future work. This would allow for application of this method in detecting leaks in programs such as those designed for mobile devices. This is where the main focus of current information leakage detection work is focused.

Java is a primary language this could be applied to since this is the language used for android device applications and the primary focus of much of the current published literature in taint analysis. A benefit of this is the ability to test this method against the benchmarks written in Java more directly and no longer need to convert them to C for testing. There are existing Java verification tools that could be used in applying this method to the language, such as Java PathFinder [46].

Exploring automating the annotation method, potentially developing a tool that can analyse the given CFG of a program and provide a list of suggested points for annotations and suggested variables to include in the leak predicate.

Complex cases of leaks involving encryption need to be addressed in future work. Predicate annotation has not been applied in cases involving encryption and would present significant challenges due to the nature of encryption's deliberate difficulty in reversing it without the right information.

Work can also be done to extend the evaluation of the predicate annotation method by comparing to traditional encodings of noninterference analysis with product programs such as those presented by Eilers et al. [47].

7.3.1 Multiple Points of Observation

A more complex case for consideration in future work is a case in programs where more than one point of observation occurs within a program and the leak occurs only if the observer can make a calculation using values from multiple observation points.

Figure 7.1 has two points of observation at locations 3 and 5, where the leak is only compromising if the observer can perform a calculation using the information from both points. In this example, it is possible for the observer to discern the precise value of the s_1 by observing the change in the public variable p_1 at the end of the program. v_1 is assumed to be a constant value that is the same for any program execution.

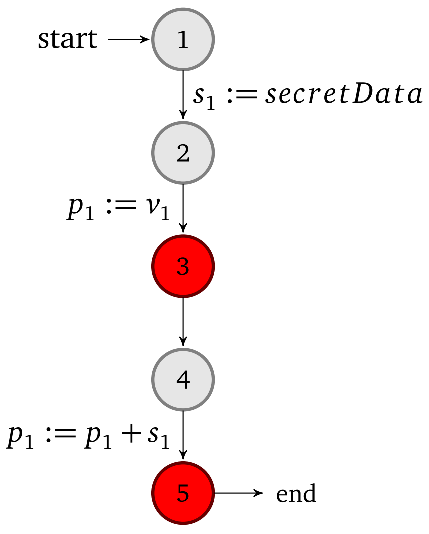

Figure 7.1: Program with two observation points

Node	Observation	Leak Predicate
3	$\{p_1\}$	
5	$\{p_1^{'}\}$	$s_1 = p_1^{'} - p_1$

Table 7.1: Observation and Leak Predicate Maps for Figure 7.1

Here the predicate to determine if there is a leak would be the difference between the value of p_1 at the first observation point subtracted from its value when it is observed again at the second observation point. To define the leak and make predicate checking easier, the annotation process could benefit from some simplification of variables for this context and the shifting of observation points.

Note that in the annotation table there is only an annotation of a leak predicate for the second observation point as there is not a concern with the first observation of p_1 leaking s_1 at that point. The leak predicate for $L(5)$ can be defined as $s_1 = p_1^{'} - p_1$ where $p_1^{'}$ is the changed value of p_1 at the second observation of location point 5.

This example assumes that the observer has the capability to remember two different outputs from the program while it is running, as it needs both the value of p_2 and p_1 to calculate the original value of s_1. If p_1's output overwrites that of p_2 then both can be remembered at the same time, and without the ability to remember both for calculation, no leak can occur as it is not possible to reconstruct s_1. The following sections will begin to look at how the capabilities of an observer can influence how a leak can be defined also.

Bibliography

[1] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pp. 317–331, IEEE Computer Society, 2010.

[2] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, The Internet Society, 2011.

[3] C. Fritz, S. Arzt, and S. Rasthofer, "DroidBench 2.0," *Online: https://github.com/secure-software-engineering/DroidBench*, 2015.

[4] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1* (A. Bertolino, G. Canfora, and S. G. Elbaum, eds.), pp. 280–291, IEEE Computer Society, 2015.

[5] M. Boreale, "Quantifying information leakage in process calculi," *Inf. Comput.*, vol. 207, no. 6, pp. 699–725, 2009.

[6] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in Lo *et al.* [48], pp. 250–261.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United*

Kingdom - June 09 - 11, 2014 (M. F. P. O'Boyle and K. Pingali, eds.), pp. 259–269, ACM, 2014.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014.

[9] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pp. 186–199, IEEE Computer Society, 2010.

[10] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pp. 146–160, IEEE Computer Society, 2011.

[11] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[12] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[13] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pp. 11–20, IEEE Computer Society, 1982.

[14] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[15] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[16] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Information Systems Security, 4th International Conference,*

ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings (R. Sekar and A. K. Pujari, eds.), vol. 5352 of *Lecture Notes in Computer Science*, pp. 56–70, Springer, 2008.

[17] N. Grech and Y. Smaragdakis, "P/taint: unified points-to and taint analysis," *PACMPL*, vol. 1, no. OOPSLA, pp. 102:1–102:28, 2017.

[18] B. Livshits, "Stanford SecuriBench Micro," *Online: https://suif.stanford.edu/~livshits/work/securibench-micro/*, 2006.

[19] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo Marugán, "Skink: Static analysis of programs in LLVM intermediate representation - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II* (A. Legay and T. Margaria, eds.), vol. 10206 of *Lecture Notes in Computer Science*, pp. 380–384, 2017.

[20] E. Siever, S. Spainhour, and N. Patwardhan, *Perl in a nutshell - a desktop quick reference*. O'Reilly, 1999.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[22] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010* (P. Tonella and A. Orso, eds.), pp. 13–24, ACM, 2010.

[23] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13,*

Berlin, Germany, November 4-8, 2013 (A. Sadeghi, V. D. Gligor, and M. Yung, eds.), pp. 611–622, ACM, 2013.

[24] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, “Copperdroid: Automatic reconstruction of android malware behaviors,” in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[25] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli, “Formal characterization of illegal control flow in android system,” in *Ninth International Conference on Signal-Image Technology & Internet-Based Systems, SITIS 2013, Kyoto, Japan, December 2-5, 2013* (K. Yétongnon, A. Dipanda, and R. Chbeir, eds.), pp. 293–300, IEEE Computer Society, 2013.

[26] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli, “Protection against code obfuscation attacks based on control dependencies in android systems,” in *IEEE Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, CA, USA, June 30 - July 2, 2014 - Companion Volume*, pp. 149–157, IEEE, 2014.

[27] Y. Feng, S. Anand, I. Dillig, and A. Aiken, “Apposcopy: semantics-based detection of android malware through static analysis,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014* (S. Cheung, A. Orso, and M. D. Storey, eds.), pp. 576–587, ACM, 2014.

[28] W. You, B. Liang, J. Li, W. Shi, and X. Zhang, “Android implicit information flow demystified,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS ’15, Singapore, April 14-17, 2015* (F. Bao, S. Miller, J. Zhou, and G. Ahn, eds.), pp. 585–590, ACM, 2015.

[29] F. Wei, S. Roy, X. Ou, and Robby, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications*

Security, Scottsdale, AZ, USA, November 3-7, 2014 (G. Ahn, M. Yung, and N. Li, eds.), pp. 1329–1341, ACM, 2014.

[30] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *CoRR*, vol. abs/1802.01050, 2018.

[31] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (J. Jung and T. Holz, eds.), pp. 65–80, USENIX Association, 2015.

[32] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "Straighttaint: decoupled offline symbolic taint analysis," in Lo *et al.* [48], pp. 308–319.

[33] Z. Wei and D. Lie, "Lazytainter: Memory-efficient taint tracking in managed runtimes," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014* (C. Wang, D. Huang, K. Singh, and Z. Liang, eds.), pp. 27–38, ACM, 2014.

[34] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proceedings of the Fifth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2010, November 4-6, 2010, Fukuoka Institute of Technology, Fukuoka, Japan (In conjunction with the 3PGCIC-2010 International Conference)*, pp. 297–300, IEEE Computer Society, 2010.

[35] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Trans. Software Eng.*, vol. 43, no. 6, pp. 492–530, 2017.

[36] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. R. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 4:1–4:37, 2016.

[37] Y. Wang and A. Rountev, "Who changed you? obfuscator identification for android," in *4th IEEE/ACM International Conference on Mobile Software Engineering*

and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017, pp. 154–164, IEEE, 2017.

[38] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, “Using spatio-temporal information in API calls with machine learning algorithms for malware detection,” in *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, AISec 2009, Chicago, Illinois, USA, November 9, 2009* (D. Balfanz and J. Staddon, eds.), pp. 55–62, ACM, 2009.

[39] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu, “Droidmat: Android malware detection through manifest and API calls tracing,” in *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, August 9-10, 2012*, pp. 62–69, IEEE Computer Society, 2012.

[40] N. Peiravian and X. Zhu, “Machine learning for android malware detection using permission and API calls,” in *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pp. 300–305, IEEE Computer Society, 2013.

[41] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, “Droid-sec: deep learning in android malware detection,” in *ACM SIGCOMM 2014 Conference, SIGCOMM’14, Chicago, IL, USA, August 17-22, 2014* (F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, eds.), pp. 371–372, ACM, 2014.

[42] Y. Aafer, W. Du, and H. Yin, “Droidapiminer: Mining api-level features for robust malware detection in android,” in *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers* (T. A. Zia, A. Y. Zomaya, V. Varadharajan, and Z. M. Mao, eds.), vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 86–103, Springer, 2013.

[43] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[44] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *CoRR*, vol. abs/1612.04433, 2016.

[45] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, The Internet Society, 2014.

[46] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[47] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," in *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* (A. Ahmed, ed.), vol. 10801 of *Lecture Notes in Computer Science*, pp. 502–529, Springer, 2018.

[48] D. Lo, S. Apel, and S. Khurshid, eds., *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, ACM, 2016.