Behaviour-based Classification of Encryption-type Ransomware using System Calls

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Christopher Chew Jun Wen

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2023

Abstract

The malware landscape is ever-changing, with threat actors utilising more sophisticated techniques to compromise data. As the usage of smartphones increases, more threat actors will turn their attention to capitalise on the popularity.

This thesis addresses this ongoing issue and focuses on encryption-type ransomware, which has been a rising malware threat in recent years, on the Android operating system. Many state-of-the-art anti-malware solutions have shifted away from static signature-based approaches as the techniques utilised by threat actors have become more advanced. Most newer solutions look towards the use of dynamic analysis to automatically identify malware. However, the large quantities of information required by dynamic analysis approaches often present a challenging task for developing robust automated anti-malware solutions and may be easily circumvented by future threat actors, which implies that more specialised automated solutions are required.

In the work presented in this thesis, we observe encryption-type ransomware behavioural patterns at a system call-level. We describe the Android Applications dataset on which a large portion of this work is based. By utilising the created dataset and the behavioural patterns, this thesis presents solutions using Finite State Machines (FSM) and supervisor reduction to quickly detect Android encryption-type ransomware. Furthermore, the solutions are evaluated on Linux encryption-type ransomware to show its transferability and generalisability.

We measured the success of our techniques by using the following accuracy metrics: true positive rates, false negative rates, true negative rates, false positive rates, and achieved an F1-score of up to 93.8%.

Acknowledgements

This journey has truly been the most challenging and memorable endeavour of my life thus far. I am thankful for this once-in-a-lifetime opportunity, and honoured to have met talented and exceptional individuals who have taught and supported me so much over the years. This journey would not have been possible without the continuous support of many individuals.

I would like to express my gratitude to The School of Graduate Research, the head of Cybersecurity Researchers of Waikato (CROW) Dr Vimal Kumar, and The Department of Computer Science, for funding my PhD. Thank you all for your generous support and allowing me to pursue this incredible opportunity without the worry of financial difficulties.

Thank you to my supervisors, Dr Vimal Kumar, Dr Robi Malik, and Dr Panos Patros. The completion of this research would not have been a reality without the three of you. Through our successes and my failures, you have all encouraged, supported, and guided me throughout this journey without falter. I truly cannot express enough how grateful and fortunate I am to be mentored by such knowledgeable, and respected individuals. I will never forget the knowledge you have all passed on to me and the guidance you have all given me. Thank you for the memorable journey, and thank you for all that you have done for me.

To the friends I have made in CROW. You have all facilitated a kind, supportive, and welcoming environment where I can always feel at ease and relatable. CROW gave me a place of belonging in my academic career and has kept me grounded and sane throughout my research. You are all incredibly talented individuals and have taught me so much over the years. Thank you for the unforgettable and fantastic memories we have all shared together, and for encouraging me in the most difficult of times during my research.

My heartfelt gratitude goes out to my closest friends in FaM for their continuous support. You have all made such an incredible and positive impact on my life. Thank you for always lending me a hand in times of hardship and making me feel appreciated. The good memories and hilarious banters we shared together are truly one of the most joyful moments of my life. I am forever thankful and grateful to you all.

To my dearest and most beloved family. I greatly appreciate your support and care throughout my entire life. The three of you have held me up in the toughest of times. Mom, thank you for your unyielding support and always working hard to ensure that life necessities were never an issue for the family. Sis, you are the best sister anyone can ask for; always encouraging and supporting me throughout my life. Your constant display of hard-work, thoughtful, and tenacious nature always encourages me to work my hardest. Dad, you are the pillar who upheld me the most through my trials and tribulations, not just in this journey, but also my life. You always believed in me, even when I did not, and you always go above and beyond to provide solace for me and the family in the most difficult of times. Thank you for being my biggest inspiration, you are the reason I strive to do better in every aspect of life.

I dedicate this thesis to those who have offered me their unwavering support and encouragement throughout the years. From the very bottom of my heart, I express my utmost and sincerest gratitude to you all, thank you.

Contents

List of Figures

List of Tables

Nomenclature

AI	Artificial Intelligence
ALBF	Average Logarithmic Branching Factor
AOT	Ahead of Time
API	Application Programming Interfaces
APK	Android Package Kit/Android Application Package
ART	Android Runtime
ASEP	Alternative System Call Execution Path
C&C	Command & Control
CLI	Command-line Interface
CSAM	Child Sexual Abuse Material
DAC	Discretionary Access Control
DEX	Dalvik Executable Format
DFS	Depth-First Search
FSM	Finite State Machine
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System

IPC	Interprocess Communication
JIT	Just in Time
LCS	Longest Common Subsequence
MAC	Mandatory Access Control
MRA	Maliciously Repackaged Application
MSA	Multiple Sequence Alignment
OS	Operating System
PHA	Potentially Harmful Application
RBFS	Reversed Breadth First Search
SSG	Sequential System call Graphs
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UID	User ID

Publications

The work done for this thesis has resulted in the following articles, which are in various stages of the publication process.

- The work described in Chapter 5 has been published in the proceedings of the 14th International Conference on Network and System Security (NSS).
- The work described in Chapter 6 is currently under preparation for submission to a journal.
- The work described in Chapter 7 is currently undergoing review after 1st revision at Elsevier's Computers & Security.

Chapter 1

Introduction

1.1 Ransomware

Ransomware poses a critical risk to all businesses and organisations in multiple sectors, such as healthcare, insurance, and entertainment, which have caused severe disruption of services, and compromised sensitive data (Burrell and RNZ, 2021; Gatlan, 2021; Toulas, 2022). The increasing frequency and sophistication of ransomware attacks have negative financial implications. According to Sophos (2022) an average of $1.4 million was required to recover from a ransomware attack in 2021. Furthermore, the cost and frequency of ransom demands have increased so much that it has even led some insurance companies to refuse cover for ransomware attacks (Umawing, 2023; Dissent, 2023). In addition to the financial implications, ransomware attacks can also have adverse ramifications on society, such as causing long-term psychological effects (Help Net Security, 2022) and, in some cases, leading to the loss of lives (Collier, 2021; Miller, 2022).

The criticality of the threat of ransomware has lead to a substantial body of

work in the area of ransomware detection and mitigation (Scalas et al., 2019; Alsoghyer and Almomani, 2019). There is, however, a scarcity of work focusing on encryption-type ransomware detection that attempts to address the resource constraints on mobile devices. This is a concern as the usage of mobile phones in our daily lives has increased significantly, with a total of 14.91 billion mobile devices worldwide surveyed in 2021. This number is expected to reach 18.22 billion by 2025 (Statista, 2021). The increase in usage of mobile phones is likely attributed to the conveniences available, such as communication applications, digital wallets, and entertainment applications. As these devices continue to become more pervasive in our lives, we are not only amassing private information on them, such as personal photos, credit card information, contacts, but also business and organisational information. Consequently, mobile phones have become a valuable and portable data storage and therefore targets of ransomware and malware attacks. The frequency of these attacks is predicted to rise as the usage of mobile phones continues to become more prominent (Awan, 2023).

1.2 Ransomware Mitigation

Malware analysts and researchers have used a variety of techniques to address and mitigate the threat of ransomware specifically and malware in general. These techniques fall into two main categories, which are *static* and *dynamic*. *Static analysis* techniques analyse static elements of applications, such as permissions, Application Programming Interfaces (APIs), and bytecode without executing them (Li et al., 2017; Bakour et al., 2018). Conversely, *dynamic analysis* observes applications during runtime to obtain behavioural information. This can be done, among

others, through the use of system calls or API calls (Bhandari et al., 2018; Tam et al., 2015).

Many state-of-the-art anti-malware solutions leverage artificial intelligence (AI) or machine learning (ML) techniques to automatically classify malware. One of the issues facing current state-of-the-art anti-malware solutions that utilise AI and machine learning is the significant resource investment required, which is often not feasible to implement on resource constrained devices. Furthermore, as the arms race between malware researchers and malware authors continues, malware authors will eventually develop methods to circumvent such AI and machine learning based anti-malware solutions. Therefore, developing more diverse techniques without relying on AI and machine techniques is essential for deterring future malware attacks.

1.3 Challenges of Anti-Malware Methods

The landscape of malware is constantly changing, posing many challenges for the development of anti-malware systems. Although improvements have been made over the years for static signature-based methods, the use of static analysis methods is no longer sustainable as malware developers utilise obfuscation techniques to evade detection (Moser et al., 2007; Yan and Yin, 2012; Or-Meir et al., 2019). Anti-malware solutions have trended towards the development of dynamic analysis approaches, which are resilient to simple obfuscation (Guerra-Manzanares et al., 2022). These dynamic analysis approaches range from varying levels of sophistication and robustness. One of the more prominent dynamic analysis techniques used in recent literature is behaviour-based analysis (Sekar et al., 2000; Isohara

et al., 2011a; Tam et al., 2015; Bhandari et al., 2018), which is a technique that monitors applications to identify malicious or benign behaviours. Other examples of dynamic analysis techniques, such as real-time malware analysis (Alam et al., 2015; Mehnaz et al., 2018), and self-protection systems (Skandylas and Khakpour, 2021; Iannucci et al., 2018), offer capabilities of early to immediate detection and consistent active monitoring.

Mobile phones have advanced significantly throughout the years, but they are often restricted by limited resources due to their portability and compactness. A large body of work in the anti-malware and anti-ransomware literature relies on the use of AI and machine learning based techniques. Such techniques often require considerable resources to perform well. As such, effective and automated solutions are often difficult to apply in a realistic scenario whilst conforming to the limited resources of mobile devices.

As previously mentioned, the use of static signature-based methods is no longer sustainable due to the constantly evolving threat landscape. Hence, some researchers have leaned towards dynamic analysis approaches that utilise behavioural patterns. Behavioural patterns provide more flexibility in detecting malicious activity as compared to traditional static signature-based methods. However, general observation of behavioural patterns can result in lower accuracy as benign and malicious applications can share similar behavioural patterns. For example, benign applications, such as cache-cleaning applications, can replicate similar systematic file removal behaviours to ransomware. Conversely, some malware variants can mimic benign applications to avoid the detection by anti-malware solutions (Aboaoja et al., 2022). Therefore, developing a reliable detection method using behavioural patterns can present a challenging task to achieve.

Some dynamic analysis approaches utilise system calls. For example, Tam et al. (2015) employ various techniques to extract high-level behaviour from system call sequences. Alternatively, Bhandari et al. (2018) adopt a different approach by using probabilistic analysis and machine learning to build a Markov chain model based on system call traces. System calls offer a balance between user-level and kernel-level analysis. User-level analysis is often unable to capture the behaviour of more sophisticated malware variants and is prone to tampering. Kernel-level offers more depth and resilience. However, the devised approaches can often result in a complex design. The large quantity of information produced by system calls can offer valuable information about an application's behaviour. This also presents a challenging task for creating automated anti-malware solutions as some of this information might not be relevant in discerning benign from malicious behaviour.

Of particular interest in this work is the utilisation of Finite State Machine (FSM) techniques and algorithms. FSMs are often fast and can be used to generalise observed patterns of malicious and benign behaviour for malware detection. Furthermore, FSMs can be used to compliment specific dynamic analysis approaches, such as the use of behavioural patterns. For example, a behavioural pattern can be converted and minimised into an FSM, which can be used to quickly identify malicious activity in linear time. As such, the use of FSMs is advantageous for developing anti-malware solutions that focus on early to immediate detection, which is crucial for more destructive types of malware, such as ransomware. Recent works have attempted to use the benefits of FSMs to detect malicious activity, but none have focused on applying specialised FSM simplification techniques on mobile devices to detect encryption-type ransomware. One of the main challenges facing the use of FSMs and other malware detection techniques is the represen-

tation of large quantities of information produced with dynamic analysis (e.g., system calls). Such information can be difficult to process and may not produce useful results.

Ransomware has been established as a prevalent issue and can affect multiple operating systems. Many anti-malware solutions are often devised for specific operating systems. This raises concerns as future malware developers will continue to develop malware, particularly ransomware, with cross-platform capabilities (Glazova, 2022; McDermott, 2023). Hence, an area of research to explore is the transferability of methodologies to detect specific malware types on different operating systems. The information gained from exploring this avenue can be used to aid in the development of more robust anti-malware solutions or improve existing anti-malware solutions.

1.4 Summary of Research Contributions

The previous section explored the landscape and challenges of anti-malware methods. The following contributions of this thesis apply dynamic analysis methods to detect Android and Linux encryption-type ransomware.[1]

- **Research Contribution 1:** The first research contribution identifies system call-level behavioural patterns for encryption-type ransomware. Chapter 5 explores this avenue by presenting a set of behavioural patterns, such as file encryption and tampering with user files through the use of system call logs. By assessing the efficacy and feasibility of the behavioural patterns to

[1]The scripts and regular expressions defined in this thesis can be accessed in the following repository: https://github.com/cc246/behavioural-ransomware-detection

detect encryption-type ransomware, shared common behaviours from different ransomware families are identified. Additionally, a dataset is generated to examine and understand the behaviour of Android encryption-type ransomware [2]. This dataset was also made available to aid researchers in their malware research. Chapter 6 expands on the first contribution by applying the advantages of real-time malware analysis to detect encryption-type ransomware. To achieve this, a streaming approach was designed using token FSMs. The detection rates of this proof-of-concept implementation was evaluated and demonstrates that it is feasible to run on an Android operating system with acceptable overhead.

- **Research Contribution 2:** Chapter 7 proposes an alternative approach using supervisor reduction (Vaz and Wonham, 1986; Su and Wonham, 2004), which deviates from traditional machine learning and AI approaches. Supervisor reduction is an FSM simplification technique used in supervisory control of discrete event systems (Ramadge and Wonham, 1989). This thesis employ this approach to automatically classify encryption-type ransomware on Android using system call data. The extensive experiments conducted in this chapter indicate that the detection rates of this approach are comparable to other anti-malware solutions with a promising avenue for detecting unknown variants.

- **Research Contribution 3:** The aforementioned contributions outlined the success of using behavioural patterns and FSMs to detect encryption-type

[2]This dataset can be accessed by request from vimal.kumar@waikato.ac.nz or info@crow.org.nz.

ransomware on Android devices. Chapter 8 demonstrates the generalisability of applying the aforementioned methods of behavioural patterns and FSMs to identify Linux encryption-type ransomware. Experiments in this chapter evaluated the detection rates and further identified differences and similarities between Android and Linux encryption-type ransomware.

1.5 Thesis Structure

The thesis is structured as follows. Chapter 2 describes the background followed by Chapter 3, which describes the related work along with the research gaps identified in recent literature. Chapter 4 provides an overview of the dataset and methodology used throughout this thesis. Chapter 5 describes a methodology, which utilises predefined behavioural patterns for identifying encryption-type ransomware on Android devices using system call data. The following chapter, Chapter 6, expands on the work of Chapter 5 by adopting a 2-layer FSM approach to detect Android encryption-type ransomware in real-time. Next, Chapter 7 proposes an automated approach using FSM algorithms and supervisor reduction to classify Android encryption-type ransomware. Chapter 8 examines the generalisability of the methodologies defined in Chapter 5 and Chapter 7 on Linux encryption-type ransomware as well as identifies the differences and similarities between Android and Linux ransomware behavioural patterns. Finally, Chapter 9 explores future work and concludes with closing remarks.

Chapter 2

Background[1]

The first section provides an overview of Android's architecture and its security history, followed by an introduction to ransomware and its behaviour. The next section describes system calls and the different categories. The following section introduces Finite State Machines (FSMs) along with the formal definition and expands on FSMs by explaining the problem of supervisor reduction and the algorithms to solve it in the tool Waters/Supremica (Åkesson et al., 2006).

2.1 Android Platform Architecture

Android is a prominent open-source, Linux based mobile operating system, which was introduced in 2008 and held a 71.62% of the operating systems market worldwide in September 2022 (StatCounter, September 2022). This section provides a description of its platform architecture. Similar to other operating systems, An-

[1]Most of this chapter is reproduced from previous publications with minor modifications. Section 2.2 is taken from (Chew et al., 2020), Section 2.3 appeared in (Chew et al., 2020, 2022), and Section 2.5 is taken from (Chew et al., 2022).

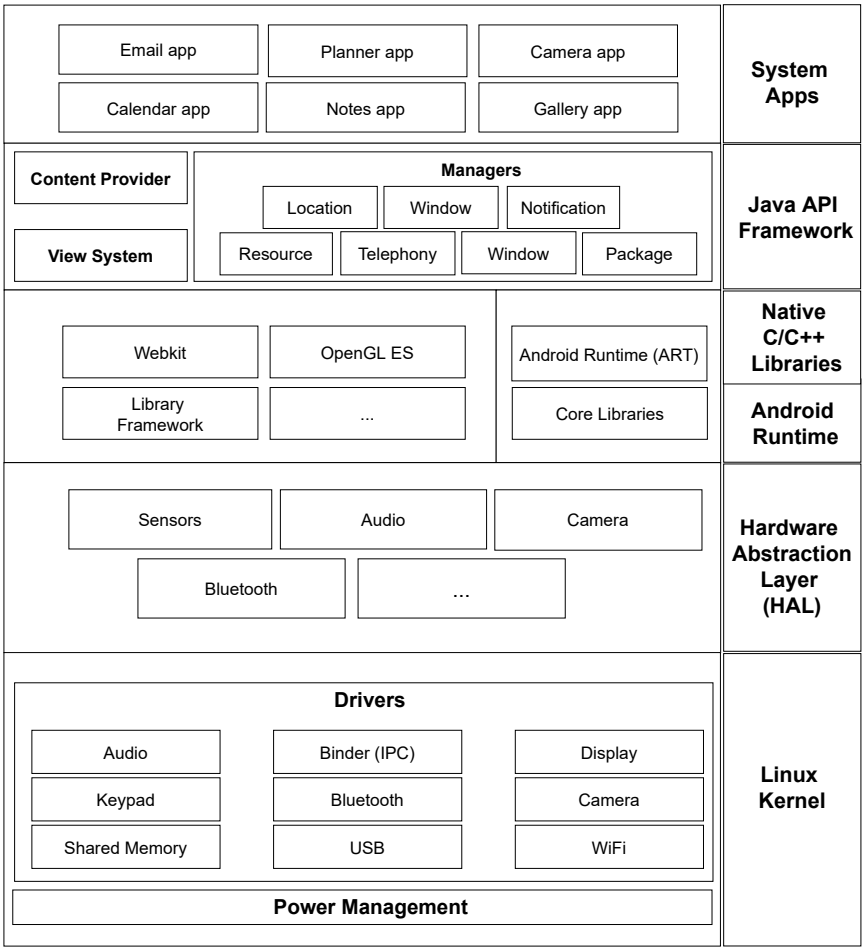

Figure 2.1: Android software stack reproduced from Google (2021) *Platform Architecture.* Retrieved from: https://developer.android.com/guide/platform

droid leverages a platform architecture which consists of different component layers that work in cohesion with each other to create the system. Figure 2.1 depicts the general architecture of the Android software stack.

System Apps is the topmost layer, where applications are installed and coexist with each other. Following that layer is the Java API layer, which provides access to the set of Android Operating System features in the form of APIs written in Java, and simplifies core modular components and services for developing appli-

cations. For example, `NotificationManager` can be used for custom notifications on the status bar, or `ActivityManager` for managing application lifecycle and navigation to the backstack.

The third layer consists of the Native C/C++ Libraries and Android Runtime (ART). The native libraries enable access to the core system components and services to applications. For example, the `OpenGL` API, can add support for drawing and manipulation of 2D and 3D graphics in applications. The native libraries can only be accessed via the Java framework APIs, and cannot be accessed directly by applications.

The second component that coexists in the same layer as Native Libraries is Android Runtime (ART). This component manages the run-time of Android applications and core system services. Originally, the Android run-time compiler was known as Dalvik. However, in 2014, this was replaced by ART (Petrovan, 2014), which offers a few improvements compared to the predecessor, such as an improved garbage collection and more extensive debugging features for applications.

The Android operating system utilises two process virtual machines (VMs) known as Dalvik and Android Runtime (ART). Dalvik, incorporates a just-in-time (JIT) compiler that dynamically converts code as it is required for execution. However, its successor, ART, replaced Dalvik and introduced notable enhancements in performance, memory management, and application execution. The primary distinguishing feature of ART lies in its adoption of an ahead-of-time (AOT) compilation approach where code is compiled during installation or updates, in contrast to the JIT compilation method employed by the Dalvik VM. Additionally, ART can utilise both AOT and JIT compilation techniques.

The next layer is the Hardware Abstraction Layer (HAL), comprising a set

of interfaces that exposes hardware capabilities to the Java API framework such as camera, and sensors. Each interface contains library modules that are loaded when the Java API framework requests to access hardware-specific components.

The last layer is the Linux kernel which is the core structure for Android that many components rely on, for example, ART relies on the kernel to do threading, and low-level memory management. The kernel also allows Android to make use of the security features that exist within the Linux kernel, such as user based permission model, and process isolation.

The Linux kernel allows the usage of application sandboxing (Google, 2022b), which isolates and separates applications from each other, preventing malicious applications from infiltrating the system and other applications. To achieve this, Android uses Discretionary Access Control (DAC) and assigns a unique user ID (UID) for each application, thus allowing them to run in their own space, isolated from each other. Overtime, this idea was further improved such as, in Android 5.0 where Mandatory Access Control (MAC) separation between the system and applications was introduced, with subsequent Android versions improving on the security of application sandboxing. Furthermore, the application sandbox leverage process isolation and per-process security provided by the Android MAC framework, which is based on SELinux (Security-Enhanced Linux) and SEAndroid (Security-Enhanced Android). Each application process is assigned a unique security context within this framework. This security context encompasses security identifiers that define the permissions and privileges associated with the respective process.

2.2 Android Security History

Since the introduction of Android, there have been many updates and improvements to its security. In 2012, Bouncer was released in an effort to combat the upsurge of Android malware in the preceding year (Micro, n.d.). Bouncer targeted pre-existing applications as well as new applications. The approach that Bouncer took was sandboxing (Lockheimer, 2012) where applications were executed and scanned for malware in an isolated environment on a cloud infrastructure; this was devoid of any access to the users' real data.

However, Bouncer's vulnerabilities were quickly identified. Oliva Hou from Trend Micro (Hou, 2012) noted that researchers were able to acquire specific details of the runtime environment, such as the duration of Bouncer's testing phase (which was five minutes), and the phone contents used in the simulated environment (two photos, one contact and the Google account). These details could easily be exploited by attackers through the use of simple obfuscation techniques to avoid detection by Bouncer.

A few years later in May 2017, a more robust approach known as Play Protect was introduced. Play Protect is the current system for defending users from malicious applications that may be lurking in the official or third-party app stores. Play Protect performs security scans periodically on the user's mobile device, and before an application is installed on the user's mobile device (Google, 2019a). If Play Protect finds a Potentially Harmful Application (PHA) from a scan, then one of the three mitigation actions are carried out depending on the severity of the application's malice.

On a low severity application, the users are warned of an application that

could be malicious via a notification; the user can interact with the notification to uninstall the application. With a medium severity application, Play Protect disables the application until it is uninstalled from the mobile device. On the highest severity, Play Protect removes the application from the user's mobile device entirely.

Play Protect determines if an application is deemed as a PHA based on a few criteria; the first one is if the application violates Google's Unwanted Application policy (Google, n.d.b), the second one is if the application is purposefully omitting critical information, and thirdly, if the application encroaches on the privacy of users by violating the Developer Policy (Google, n.d.a). Furthermore, Play Protect utilises machine learning algorithms to identify PHA applications (Tetali, 2018). The algorithms aim to observe common PHA behaviour detected in applications, such as, accessing other applications on the user's device, and accessing or sharing private user data. These observations are used to determine if an application is malicious. If the algorithm detects that applications contain similar malicious behaviours, then it will group them into their respective families (i.e., *backdoors*, and *click frauds*). New unidentified PHAs are reviewed to determine if the application is malicious or a false positive.

In addition to the introduction of Play Protect, a security Application Programming Interface (API) called SafetyNet Verify Apps was introduced in September of the same year. This API aimed to address three key goals: to help further protect users from malicious applications, determine if a user's device is protected by Play Protect, and prompt users to enable Play Protect if it is disabled. SafetyNet achieves the aforementioned goals through a combination of different security measures and checks. One of them is through device attestation, which verifies the

integrity and authenticity of an Android device.

2.3 Ransomware

The definition of *Ransomware* is broadening and ever-evolving, such that threat actors are employing all available tactics to obtain ransom. The work described in this thesis limits the definition of ransomware to a malware, that aims to hold the users' device or data at ransom, often for monetary gain. Ransomware uses a variety of methods to gain access to computers and devices, ranging from exploit kits and email phishing links to being incorporated with other types of malware such as trojans and botnets (Al-rimy et al., 2018a; Kumar and Ramlie, 2021; Hull et al., 2019; Aurangzeb et al., 2017). Newer variants and iterations have appeared over the years, adopting more sophisticated techniques, such as self-propagation, stronger encryption, and alternative infection vectors (Richardson and North, 2017; O'Kane et al., 2018).

With the growing numbers of mobile devices, ransomware, such as WannaLocker, SimpleLocker, Filecoder, and Black Rose Lucy (Goud, n.d.; Mana et al., 2020), have found their way into the mobile ecosystem. Ransomware are generally of two types: locker ransomware and encryption ransomware (Mohammad, 2020). Locker-type ransomware traditionally displays a persistent screen that prevents the user from interacting with the rest of the system. This screen will often display the ransom note demanding monetary payment. On mobile devices, specifically Android, locker-type ransomware makes the application persistent by displaying a perpetual alert dialog or activity, or disabling interactions with the navigation bar (Andronio et al., 2015). Another technique used is altering users' lock screens,

thus preventing access to their devices (Al-rimy et al., 2018b; Kanwal and Thakur, 2017).

Encryption-type ransomware manipulates information on the user device by encrypting their files to prevent them from accessing any of their data (Kok et al., 2019; Al-rimy et al., 2018b). Similar to locker-type ransomware, a ransom note is often displayed after the encryption phase has been completed. Typically, for encryption-type ransomware, the process begins by scanning the user's personal directories, such as *Documents*, and *Pictures* for files. Once the scanning phase has completed, the ransomware often identifies files containing specific extensions, such as, *.docx*, *.png*, and *.jpg* to encrypt. This method is normally used to accelerate the encryption process, and efficiently determine the important user files to encrypt (i.e., the files most important to a user) (Gazet, 2010). For the encryption process, the data of the identified files are read, and written to a new encrypted file with a non-standard file extension. The original file is then removed or overwritten (Chen et al., 2017).

The ever shrinking cost of malware production as well as better chances of ransom payment have resulted into a shift towards encryption-type ransomware (Alzahrani and Alghazzawi, 2019; McConnell, 2017; Richardson and North, 2017). This trend raises further issues due to the low mitigation rates for encryption-type ransomware. For example, according to the 2021 State of Ransomware report by Sophos (Sophos, 2021) only 34% of cross-sector and 39% of retail sector organisations (from surveyed retail IT managers) successfully prevented their data from being encrypted by ransomware attacks. All of these put together indicate the need for preventative measures, specifically aimed at combating encryption-type ransomware.

2.4 Introduction to System Calls

System calls are used to communicate and request actions to specific resources between a user application and the kernel of an operating system (Kerrisk, 2021a,h). System calls on the Linux operating system can be classified into six main categories (Silberschatz, 1998). As Android was built upon the same operating system, the same categorisation can be applied.

- **Process Control** – *Process Control* is the first category of system calls. System calls associated with this category handles process creation and terminations, such as the `exit()` system call on Android, which is used to terminate a process (Kerrisk, 2021c).

- **File Management** – The second category of system calls is *File Management*. System calls in this category are used for file related operations. For example, `close()` can be used to close an open file descriptor and `read()` can be used to read the contents of a file (Kerrisk, 2021b,g).

- **Device Management** – Another category of system calls is *Device Manipulation*. In this category, system calls are used for manipulating device-specific interactions. An example of a *Device Management* system call is `ioctl()`, which is used for handling special interactions related to the device (Kerrisk, 2021e).

- **Information Maintenance** – The *Information Maintenance* category are system calls that can transfer information between the user application and the operating system. For example, `getpid()` can be used to acquire the process ID of the requesting process (Kerrisk, 2021d).

- **Communication** – Some system calls can be classified in the *Communication* category. This category of system calls are used for interprocess communication. An example of a *Communication* system call is `mmap2()`, which maps a file or device into memory (Kerrisk, 2021f).

- **Protection** – The last category of system calls is *Protection*. System calls in this category are often used for controlling access to specific resources, such as `umask()`, which is used for changing the default permissions on newly created files or directories (Kerrisk, 2021i).

System calls have been used for kernel-level malware analysis in recent literature. Many authors have used this approach as precise information can be obtained from monitoring the system call operations that occurred during the execution of an application, which can help identify malicious activities. For example, `openat()` represents a file was being opened, followed by accompanying arguments and flags. The flags can be used to determine the type of file open operation (e.g., `O_CREAT` is used to create a new file if the file does not already exist). Due to this, system calls can often provide useful behavioural information of an application from a single operation.

To intercept and log system calls, authors often rely on debugging tools, such as `strace` (Levin, n.d.). `strace` is a Linux command-line tool, which can intercept the system calls of a given process ID. On Android, the standard emulator (created using Android Studio) can utilise `strace` through Android Debug Bridge (ADB) to intercept the system calls of an application. Some researchers have deviated from the usage of debugging tools and implemented their own logging mechanisms, such as Tam et al. (2015), which utilised a modified emulator to intercept and analyse

system calls. Alternatively, instrumentation methods can also be used to intercept system calls, such as (Beaucamps et al., 2010), which relies on `Pin` (Intel, 2020), a tool for instrumenting a program, that can be used to acquire system calls.

Intercepted system calls can be used to generate a system call trace. A system call trace is a collection of system calls captured from a user application. System calls can generate large quantities of information due to the interactions that occur, which often results in large system call traces. Large system call traces can present a challenging task for pre-processing, particularly if they are used for streaming data.

2.5 Introduction to Finite State Machines

Finite State Machines (FSMs) are used in both hardware and software implementations, to indicate a sequence of logical statements. The main components consist of states, that represent the different traversal locations, and state transitions, that link the different states together with a set of possible transition rules (Hopcroft et al., 2001). The use of FSMs is advantageous in ransomware detection it provides fast detection times, which is useful for more destructive variants of ransomware, particularly, encryption-type ransomware. Furthermore, FSMs can be implemented in a lightweight monitoring system, offering benefits for resource constrained devices. The following section describes in more detail the formal definition of FSMs in relation to system call traces.

2.5.1 Languages and Finite-State Machines

There are two distinct types of FSMs; deterministic and nondeterministic. A Deterministic FSM has a unique target state for a given source state and event. Conversely, a nondeterministic FSM can contain more than one target state for a given source state and event.

System call traces can be described as traces of *events* taken from a finite *alphabet* Σ. The set Σ^* contains all finite *traces* of the form $\sigma_1 \cdots \sigma_n$ of events from Σ, including the *empty trace* ε. A subset $L \subseteq \Sigma^*$ is called a *language*. The *concatenation* of two traces $s, t \in \Sigma^*$ is written as st. A trace $s \in \Sigma^*$ is called a *prefix* of $t \in \Sigma^*$, written $s \sqsubseteq t$, if there exists a trace $u \in \Sigma^*$ such that $su = t$. The *prefix-closure* of a language $L \subseteq \Sigma^*$ is the set of all the prefixes of its traces, written $\mathrm{Pre}(L) = \{\, s \in \Sigma^* \mid s \sqsubseteq t \text{ for some } t \in L \,\}$.

Definition 2.1. *A (nondeterministic)* finite-state machine (FSM) *is a tuple* $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ *where* Σ *is a set of* events*,* Q *is a finite set of* states*,* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the* transition relation*, and* $Q^\circ \subseteq Q$ *is the set of* initial states*.*

The transition relation is written in infix notation $x \xrightarrow{\sigma} y$ and extended to traces $s \in \Sigma^*$ in the standard way. For state sets $X, Y \subseteq Q$, the notation $X \xrightarrow{s} Y$ means $x \xrightarrow{s} y$ for some $x \in X$ and $y \in Y$. For a state or state set x and y and $s \in \Sigma^*$, the notation $x \xrightarrow{s}$ means $x \xrightarrow{s} y$ for some $y \in Q$. The *language* $\mathcal{L}(G)$ of an FSM G is the set of all traces that can be executed from an initial state,

$$\mathcal{L}(G) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} \,\} \,. \tag{2.1}$$

An FSM G is *deterministic* if it has exactly one initial state, $|Q^\circ| = 1$, and the transition relation allows at most one successor state for any given source state and event, i.e., $x \xrightarrow{\sigma} y_1$ and $x \xrightarrow{\sigma} y_2$ implies $y_1 = y_2$.

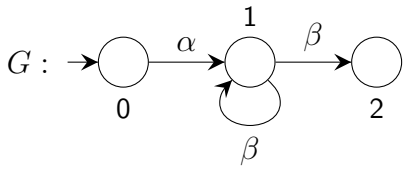

Figure 2.2: Nondeterministic FSM example

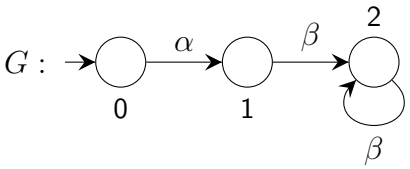

Figure 2.3: Deterministic FSM example

Example 2.1. *Consider a nondeterministic FSM G consisting of states* 0*,* 1*, and* 2 *with the events* $\{\alpha, \beta\}$ *in Figure 2.2. The FSM is nondeterministic as it consists of two* β *transitions that occur from state* 1*. Conversely, Figure 2.3 is the deterministic equivalent of the same FSM G using subset construction (Hopcroft et al., 2001). The main distinction for determinism is the singular initial state, labelled as state* 0*, and the transition relation has at most one successor state.*

2.5.2 Supervisor Reduction Problem

Supervisor reduction (Vaz and Wonham, 1986) is an FSM simplification method used for supervisory control of discrete event systems (Ramadge and Wonham, 1989). It can be characterised based on two subsets of an FSM's state set Q, the set of *positive* states $Q^+ \subseteq Q$ and the set of *negative* states $Q^- \subseteq Q$. The sets of

traces leading to these states are

$$\mathcal{L}^+(G) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} Q^+ \,\} \;; \tag{2.2}$$

$$\mathcal{L}^-(G) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} Q^- \,\} \;. \tag{2.3}$$

In supervisory control, Q^+ may represent a set of states where a particular control action is to be enacted, while the same action is to be prevented at states in Q^-. When classifying system call traces, positive traces in $\mathcal{L}^+(G)$ are known *malicious* traces, while traces in $\mathcal{L}^-(G)$ are *benign*.

It is assumed that no trace is both positive and negative, i.e., $\mathcal{L}^+(G) \cap \mathcal{L}^-(G) = \emptyset$. On the other hand, the sets of positive and negative states do not necessarily cover the complete state set, and likewise the combined positive and negative languages do not cover the full language of the FSM. That is, $Q^+ \cup Q^- \subseteq Q$ and $\mathcal{L}^+(G) \cup \mathcal{L}^-(G) \subseteq \mathcal{L}(G)$ where equality does not necessarily hold. States or traces that are neither positive nor negative are called *"don't care"* states or traces. For supervisory control, "don't care" states are states where it is safe to enable or disable a control action, for example because the action has no effect in that state. When classifying system call traces, "don't care" traces are not known to be malicious or benign, their classification is yet to be determined.

The objective of supervisor reduction is to replace a deterministic FSM G by a smaller deterministic FSM G' while preserving the positive and negative traces and still not classifying any trace as both positive and negative. That is,

$$\mathcal{L}^+(G) \subseteq \mathcal{L}^+(G') \quad \text{and} \quad \mathcal{L}^-(G) \subseteq \mathcal{L}^-(G') \quad \text{and} \quad \mathcal{L}^+(G') \cap \mathcal{L}^-(G') = \emptyset \;. \tag{2.4}$$

Every trace classified as positive or negative by the original FSM G must be classified in the same way by the reduced FSM G'. This reduction problem is different

from standard FSM minimisation problems (Hopcroft et al., 2001) because of the "don't care" traces, which the reduced FSM may classify as positive or negative in ways that allow for a smaller number of states.

2.5.3 Supervisor Reduction Algorithm

Given a deterministic FSM with positive and negative states, the usual approach to supervisor reduction is to *merge* states, grouping them into a single state in the reduced supervisor. Positive states can be merged with other positive states or with "don't care" states, and likewise negative states can be merged with other negative states or with "don't care" states; it is also possible to merge two "don't care" states. However, it is not possible to merge a positive state with a negative state, because this leads to violation of the requirement $\mathcal{L}^+(G') \cap \mathcal{L}^-(G') = \emptyset$ in (2.4), meaning that the reduced supervisor cannot separate the positive and negative traces accurately.

Additionally, the transition relation needs to be taken into account. When merging two states that both have outgoing transitions with the same event, their successor states must also be merged. The successors must not be in direct conflict, i.e., it is not possible for one of them to be positive while the other is negative, and the outgoing transitions of the successors must also be checked. Once all successors have been checked without ever having to merge a positive state with a negative state, the original two states can be merged to form a reduced supervisor, provided that all the checked pairs of successors are also merged.

Example 2.2. *Consider states* 0 *and* 1 *of FSM G in Figure 2.4. There is no immediate problem to merge these states as the negative state* 0 *can be merged with*

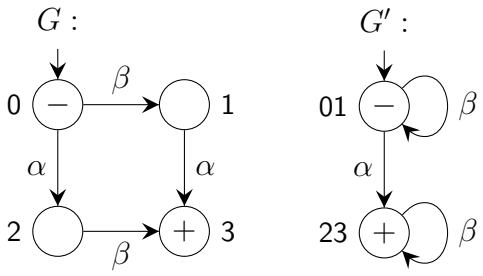

Figure 2.4: Example of supervisor reduction. States 0 and 1 of G are merged to form a new state 01 of G', and likewise states 2 and 3 are merged to 23.

the "don't care" state 1*. Yet, both states have outgoing transitions with event* α*. If* 0 *and* 1 *are merged, then their successors* 2 *and* 3 *must also be merged. This is possible here, as* 2 *is a "don't care" state that can be merged with the positive state* 3*. As neither* 0 *and* 1 *nor* 2 *and* 3 *have other common successors, they can be merged.*

The result G' *is shown in Figure 2.4. It classifies traces that do not contain the event* α *as negative and traces that contain the event* α *as positive. In terms of states, the don't care state* 1 *of* G *is classified as negative by* G' *while* 2 *is classified as positive. With this assignment, the number of states is reduced from 4 to 2 while still classifying all the positive and negative traces of* G *correctly.*

In general, supervisor reduction algorithms seek to establish a *cover* of the state space, i.e., a set $\mathcal{C} = \{C_1, \ldots, C_k\}$ of subsets $C_i \subseteq Q$ of the state set Q that covers the entire state set, i.e., $C_1 \cup \cdots \cup C_k = Q$. The subsets C_i are called *cells*, and each cell becomes a state of the reduced supervisor. For the cover to be feasible for supervisor reduction, none of its cells can contain both a positive and a negative state, and the cover must be *dynamically consistent* (Su and Wonham,

2004), i.e., it must respect the transitions as explained above.

This thesis uses an algorithm described by Su and Wonham (2004) to compute such a cover and reduce an FSM. This algorithm repeatedly selects two states and attempts to merge them. To do so, it explores all successor pairs to determine whether a positive state needs to be merged with a negative state, collecting additional pairs that need merging in the process. Any overlapping cells encountered are merged into a single cell, so that the computed cover has no overlapping cells and becomes a *partition*. If the two states selected are found to be mergible, they are merged, otherwise the pair is skipped. In this way, the algorithm systematically tries to merge all state pairs.

The supervisor reduction algorithm by Su and Wonham (2004) is arguably the most popular supervisor reduction algorithm currently used in supervisory control due to its trade-off between speed and reduction effectiveness. Its time complexity is $O(|\Sigma||Q|^4)$ where $|\Sigma|$ and $|Q|$ are the numbers of events and states of the FSM to be reduced. The result of the algorithm is sensitive to the order in which the state pairs are processed and merged. The implementation in Waters/Supremica (Åkesson et al., 2006) supports different orderings of states and pairs to allow for experimentation.

The supervisor reduction algorithm can be combined with a pre-processing step of *projection* (Malik, 2020). The idea is that many supervisory control and classification problems contain more events than necessary to distinguish positive and negative traces. It can be determined with a relatively quick *verifier algorithm* whether or not an event or a set of events can be removed from an FSM while still allowing for the distinction of all positive and negative traces. Using this repeatedly, the method of Malik (2020) searches for a set of events that removes

as many transitions as possible. Once an event set is found, the events are removed, the resulting nondeterministic FSM is determinised using subset construction and minimised (Hopcroft et al., 2001), and the result of this simplification is passed to supervisor reduction with the algorithm of Su and Wonham (2004). While the removal of events is usually beneficial, this cannot be guaranteed due to the exponential worst-case of subset construction.

2.6 Chapter Summary

This chapter presented the relevant background information for the works to follow in this thesis. The following chapter presents the related work and identifies the research gaps in recent literature.

Chapter 3

Literature Review

This chapter explains the four different categories of malware analysis techniques and describes the state-of-the-art malware analysis research in the respective categories.

3.1 Static Analysis

Static analysis observes static components of a given application to determine if it is malicious or benign, and gains insight on its functionality without executing the application. Many Android-based static analysis methods extract static features, such as permissions, Application Programming Interface (API), or bytecode, from an application. These static features are then used with other techniques, such as machine learning algorithms, to classify applications as malicious or benign.

An example is DroidDet (Zhu et al., 2018), which uses machine learning on extracted static features consisting of permissions, system events, frequency of permissions, sensitive APIs and URLs, to detect Android malware. Similarly,

Maiorca et al. (2017), focuses on extracting invoke-type instructions from the Application Package Kit's (APK) bytecode. Another work that observes Android bytecode is DroidAPIMiner (Aafer et al., 2013), which mines for specific API calls and parameters. MaMaDroid (Onwuzurike et al., 2019), observes sequences of abstracted API calls represented as Markov chains and classified through machine learning to identify Android malware. Alternately, DroidNative's (Alam et al., 2017) approach utilises both bytecode and native code to detect Android malware.

Traditional code and signature analysis techniques have been shown to be capable of detecting some known malware. The main challenges facing *static analysis* is the use of obfuscation techniques, such as binary/code/control flow obfuscation, and polymorphic coding (Gandotra et al., 2014; Moser et al., 2007; Faruki et al., 2014), which are often employed in more sophisticated malware to avoid detection. Researchers, such as, Alam et al. (2017) have attempted to alleviate some of the effects of obfuscation by observing native code. One of the core limitations of utilising signature and code analysis stems from the inability to detect newer and unknown variants of malware, which is an issue as the malware landscape continues to evolve with more destructive and unique variants.

3.2 Dynamic Analysis

The second category of malware analysis techniques is *dynamic analysis*, which monitors and observes the behaviour of applications while they are executing, such as tracking file system access, network activity, or memory contents. Such information can be acquired by using tools to log or intercept system calls and other API calls using debugging tools (e.g., `strace` (Levin, n.d.)), implementing

logging interfaces (Sekar et al., 2000; Tam et al., 2015), or by using instrumentation (Beaucamps et al., 2010; Xu et al., 2012; Abbasi et al., 2022). Dynamic methods are more resilient to common static obfuscation techniques (Hou et al., 2016; Guerra-Manzanares et al., 2022) because they observe the behaviour of an application rather than its static elements.

Similar to the approach used in Arzt et al. (2014), taint analysis can also be used in *dynamic analysis*. Enck et al. (2014) uses this approach along with variable-level tracking of native methods within the Dalvik VM interpreter, which contains taint markings in a taint map. These taint markings propagate through the Android Inter-Process Communication Binder, based on the defined data flow rules on how the application uses the tainted data, to the untrusted application's taint map. If the untrusted application makes a library call deemed as a taint sink (e.g., *network send*), then the application is considered malicious.

Behavioural pattern detection at a system call-level is another *dynamic analysis* technique, which has often been used in kernel-level malware analysis. System calls are useful in determining the precise operations that occurred during the execution of an application, which can help identify malicious activities or behaviours. Works, such as Isohara et al. (2011b); Tam et al. (2015); Lin et al. (2013) leverage system calls to identify malicious applications effectively on mobile operating systems.

System call monitoring often generates large quantities of information, some of that information might not be relevant in identifying the behaviour of an application, such as `clock_gettime()` that periodically records the system clock time. Works, such as Isohara et al. (2011b) attempts to mitigate this issue by using a filtering process. The filtering process selects categories of system calls that are

important for identifying malicious activity, and uses a process tree to remove processes, which are considered unrelated to the application.

CopperDroid (Tam et al., 2015) adopts a dynamic analysis approach, which utilises value-based data flow analysis on system call sequences and Interprocess Communication (IPC) unmarshalling to reconstruct the high-level behaviour of Android malware. Similarly, Lin et al. (2013) uses the Longest Common Subsequence (LCS) algorithm to extract potentially malicious patterns from system calls and utilises the Bayes theorem to determine if an application was a Maliciously Repackaged Application (MRA) based on the extracted patterns.

While there have been recent works on pattern detection on system call logs (Isohara et al., 2011b; Lin et al., 2013), none has specifically focused on patterns produced by encryption-type ransomware at a system call-level. Hence, one of the research gaps is to discover a set of common behavioural patterns for encryption-type ransomware, such as file encryption and tampering with user files by utilising the system call logs. Furthermore, many malware datasets have been made available for research purposes throughout the years. A few of which focus on specific types of malware, such as spyware, and botnets. While there exist malware collections, such as malware zoo and malware datasets, which target specific types of malware, like spyware and botnets, none of them have specifically focused on a dataset encompassing Android-based encryption-type ransomware along with accompanying system call logs. This thesis addresses this by building a collection of Android based encryption-type ransomware as well as a system call dataset to experiment with system call-based detection techniques.

3.3 Hybrid Analysis

Hybrid analysis is a combination of both static and dynamic analysis. The idea of hybrid analysis is to combine the advantages that each analysis technique has to offer (Gaikwad et al., 2015), and use them to counteract the drawbacks, where applicable. For example, static analysis alone is incapable of detecting unknown/undiscovered malware. The use of dynamic analysis can be used to counteract this issue by first capturing the behaviour of the application before passing the results to a static component for further analysis.

A work that focuses on a hybrid approach is DNADroid (Gharib and Ghorbani, 2017), which adopts a real-time malware detection approach to detect Android ransomware. In the static module, features are extracted from the Android Application Package (APK), such as permission requests, words, terms, and images commonly used in ransomware screens. These features are then processed by machine learning models and given a malware score (between 0 and 1). Whereas, the dynamic component utilises a sandbox environment to capture the API call sequences, which are pre-processed by removing common API calls sequences that appear in both benign and malicious applications. After pre-processing, DNADroid utilises Multiple Sequence Alignment (MSA) for aligning multiple extracted strands of API call sequences to acquire the common malicious subsequences.

These modules are utilised by the real-time detection module, which determines if an application is malicious or benign. To achieve this, the static classifier scores the application between 0-1 (benign or malicious) based on the trained model. If an application contains a score higher than the threshold (1-confidence score of

application), then the dynamic component extracts the common API subsequences using MSA. These extracted subsequences compares against other previously extracted subsequences using Binary Subsequence Alignment (similar to MSA except the comparison is only between two sequences). If the sequence matches, then the application is deemed malicious. Otherwise, the application continues to execute within the dynamic environment in 5-minute intervals until a malicious sequence match is detected.

Tong and Yan (2017) provide a good illustration of a hybrid approach used in Android malware analysis. The implementation begins with a dynamic component that compiles individual and sequential system call data acquired from applications through the modification of the Android operating system. This data is then passed through to a static component that generates malicious and benign patterns, and performs pattern comparison to determine the true nature of the application.

Hybrid analysis combines the benefits of static and dynamic methods together. Often time, this alleviates the limitations of each respective approaches. Unlike static or dynamic analysis alone, the limitations of hybrid analysis are not as clearly defined. One notable limitation of dynamic analysis is its susceptibility to potential false positives, coupled with the inherent risk of triggering infections while executing applications.

Conversely, static analysis is less effective in detecting unknown variants and is susceptible to simple obfuscation techniques. In the case of hybrid analysis, the limitations can vary depending on the methods employed, but resource consumption emerges as a common constraint. It is important to note that the precise limitations of hybrid analysis can vary based on the applied methodologies and

implementation strategies. While resource consumption is the most common limitation, further exploration is warranted to identify other limitations associated with hybrid analysis techniques.

3.4 Real-Time Analysis

With the dynamically changing and ever-evolving threat landscape, malware has become more sophisticated and cunning. Hence, to counteract this rapidly changing landscape, some malware detection systems have trended towards real-time malware analysis, which provide benefits, such as early to immediate detection and consistent active monitoring. Encryption-type ransomware attacks currently poses as one of the more challenging types of malware to recover from once a user has been affected, and adopting a real-time approach would be beneficial in minimising or mitigating its effects.

Hofmeyr et al. (1998) is one of the earlier works that adopted a real-time approach. Hofmeyr et al. (1998) work focus on the detection of anomalous intrusions through the utilisation of system call sequences, employing a distinction between self-discriminating and non-self discriminating system call sequences. A self-discriminating system call sequence refers to one that is deemed benign, while a non-self sequence is characterised as malicious.

Similarly, Maggi et al. (2008) propose a host-based intrusion detection system (HIDS) using system call sequence clustering and Markov Chains for modelling system call sequence to detect anomalous activity, specifically focusing on buffer overflow attacks. Their work expands and improves on a pre-existing Intrusion Detection System (IDS) known as SyscallAnomaly (Kruegel et al., 2003), which

generates profiles of system calls based on the arguments to identify the normal behaviour of a program. The methodology of Maggi et al. (2008) clusters same system calls based on the arguments to identify the different ways the same system calls can be used (i.e., an open system call can be used to read a file with the read-only flag (`O_RDONLY`) or read and write to a file with the read-write flag (`O_RDWR`). To model the program flow, they utilise Markov Chains to observe sequences of clustered system calls, enabling them to identify and characterise the program's behaviour. Maggi et al. (2008) applied their methodology in a prototype implementation to show the feasibility of the proposed approach as an IDS. This prototype was further improved on in later works, which focuses on reducing the false positive rates (Maggi et al., 2009). However, the clustering of system calls generates noticeable performance issues with 700 MB of memory usage in the worst-case scenario.

Sun et al. (2018) adopts a similar approach, which uses systems calls for real-time malware detection. The first process is initialisation, which generate resource files upon the first execution of the Android application. The second process, dynamic behaviour detection, adds a hook to the kernel to acquire system calls. The application's permissions and APIs are extracted with the decompiler tool known as ApkTool (Wiśniewski, 2010). These are utilised in a preparation phase, where applications statistics are acquired, such as the number of permissions used. The final process is *Malware Application Identification*, which implements a classifier with naive Bayes to identify if an application was benign or malicious. One of their limitations due to the usage of dynamic analysis approaches is the potentially extensive analysis time, which is more evident in larger applications.

Based on the surveyed literature of real-time approaches, we conclude that none

have specifically developed a real-time streaming approach focusing on Android encryption-type ransomware at a system call-level. This thesis explores this research gap by proposing a proof-of-concept real-time streaming approach to detect Android encryption-type ransomware.

3.5 Automaton-based Approaches

Some malware analysis approaches monitor behavioural aspects and use the information to build a *finite-state machine (FSM)* model of malicious behaviour. One of the earlier works of automaton-based approaches is Erlingsson and Schneider (1999), which propose a method to merge security policy enforcement code into the object code for a targeted system, in their work they evaluated on x86 architecture and Java JVML (Java virtual machine language). This was achieved using a security automaton where the alphabet is the events that the reference monitor would see (e.g., read or send), and the transition relation encodes a security policy. Simiarly, Beaucamps et al. (2010) utilises trace abstraction to model behaviours in programs. Their approach achieves this by extracting the execution trace of programs and later abstracted to indicate a high-level behaviour represented through a trace automaton. Using the trace automaton, they acquire behavioural patterns from observing malicious execution traces and basic execution sequences that could correlate to malicious behaviour. Alternatively, other works focus on utilising Markov Chains and machine learning to classify malware, such as Onwuzurike et al. (2019), which observe sequences of API calls represented as Markov Chains and classified through machine learning to detect Android malware.

Similarly, other works have also adopted the use of system calls in an automaton-

Table 3.1: Comparison of Static (SMA), Dynamic (DMA), Hybrid (HMA), and Real-time (RMA) malware analysis approaches

SMA	DMA	HMA	RMA
Safely identify known malware	Can identify unknown malware	Combines benefits of static and dynamic	Quickly prevent malicious activity
Quickly identify known malware	Resilient to common obfuscation	Often counteract drawbacks of static and dynamic	Constant and active monitoring
Ineffective to unknown malware variants	Often long analysis times	Often resource intensive	Effective for detecting evolving threats
Vulnerable to common obfuscation techniques	Can produce higher false positives and negatives		

based approach such as, Sekar et al. (2000) developed an FSM and machine learning approach based on system calls sequence and program counters for intrusion detection. Whereas, Semantic aWare andrOid malwaRe Detector (SWORD) (Bhandari et al., 2018) generates sequential system call graphs (SSG) through the use of Markov Chains to acquire the expected paths exhibited by malware. From the derived paths, they apply Average Logarithmic Branching Factor (ALBF) to acquire numerical representations of the typical paths.

Automaton-based approaches, such as the aforementioned literature, constructs an FSM model from observed system call traces. However, none have focused specifically on classifying encryption-type ransomware for resource constrained devices. Hence, one of the research gaps is to develop an alternative automated solution using an FSM simplification technique for identifying Android encryption-type ransomware, without the reliance on machine learning or AI, which often requires demanding resource investment.

3.6 Chapter Summary

This chapter explored the state-of-the-art malware analysis techniques in different categories.The main points of each malware analysis approaches explored in this

chapter have been summarised in Table 3.1. Furthermore, based on the literature surveyed, different research gaps were identified. The following chapter describes the dataset acquisition process and main methodology used in this thesis.

Chapter 4

Methodology

This chapter describes the data collection methodology used in this thesis, and also the challenges encountered during the collection process. The first section outlines the process of acquiring an Android Applications dataset with the following section describing the acquisition of a Linux dataset. The subsequent section describes how the two datasets are used to acquire system call logs, which will be utilised for various experiments in the following chapters of this thesis.

4.1 Android Applications Dataset Acquisition

The Android Applications dataset compromises of 715 cumulative samples, 502 were benign and 213 were malicious. The 502 benign samples were acquired through web-scraping from APKPure (n.d.). The samples include a variety of categories such as entertainment, utility, and productivity applications. Two distinct benign cache cleaning applications were included in the dataset to simulate similar behaviours resembling that of encryption-type ransomware, such as the sys-

tematic removal of files. The primary objective of a collecting a benign dataset is to ensure that there is diversity in the applications observed, and that each benign sample was capable of executing on the emulated device, which was not always feasible with some benign samples due to errors occurring, such as the usage of an incompatible API level.

For malicious samples, 213 Android encryption-type ransomware were acquired by retrieving the hash or package name published from established anti-virus vendors, such as Avast Blog (n.d.) and ESET (WeLiveSecurity, n.d.), and relevant search tags, such as family name. Furthermore, hashes were validated utilising VirusTotal (Sood, 2017) to identify and classify the ransomware by family. The validated hashes were used to download the associated application from Koodous (n.d.), which contains a repository of packages often used for malware analysis.

One of the motivations of developing our own dataset was due to the observation of illegal explicit content in existing dataset. Some malware samples in the existing datasets were found to contain Child Sexual Abuse Material (CSAM). The use of illegal explicit content is a common tactic used by malware developers to threaten and scare the users into paying the ransom. As dynamic analysis was the primary method utilised in the thesis, the execution of ransomware was required. This posed a challenging issue as often times the explicit content was displayed during the execution of the ransomware, which was deemed inappropriate for viewing. To alleviate this issue, we developed a dataset that deviated from such content.

Initially, the malicious dataset consisted of 500 applications from 10 different ransomware families. However, we required malicious ransomware samples, which had functioning encryption capabilities. Hence, after further analysis, it was dis-

covered that several ransomware applications did not encrypt the users' files. This was likely due to a few samples requiring a connection to a C&C server that was no longer active. Additionally, some samples were unable to be observed due to issues, such as crashes occurring during start-up, or missing manifest files. Hence, the final malicious dataset contains 213 malicious samples from 6 distinct families.

The six distinct families are shown in Table 4.1. These are families of known malware that first appeared during 2014–2020, and all were still circulating in 2020, which was the year our data collection began. The applications in each family can be traced to the same malware core, albeit with slight variations and different packaging (e.g., different application type, or name). Except for Wipelocker, the malicious applications exhibit typical ransomware behaviour of encrypting and replacing files. Wipelocker only deletes the user's files with no evidence of encryption occurring (Chew et al., 2020). It is nevertheless included in the dataset because the deletion of files is still malicious, and the behaviour is similar to that of encryption-type ransomware.

4.2 Linux Dataset Acquisition

Linux is the operating system of choice for a number of users who typically work in core ICT fields. The environment, applications, and the toolchains used, however, vary significantly from user to user. As such, acquiring benign samples was a challenging task as tools and applications used by the average Linux user cannot be easily defined. Hence, to mitigate this issue, the selected samples attempt to conform to a particular category of applications, specifically, applications used by developers for software development.

Table 4.1: Distribution of Android ransomware based on family

Ransomware Family	Sample Size
Simplelocker	64
Pletor	6
Filecoder	5
Black Rose Lucy	17
Wipelocker	70
Wannalocker	51
Total	213

Based on the aforementioned category, we have selected 170 benign samples to reproduce a similar distribution of samples to the Android Applications dataset. Similarly, two cache-cleaning applications were included in the 170 samples. The acquired dataset contains command-line interface (CLI) and graphical user-interface (GUI) applications, which were specifically selected to replicate different operations. It should be noted that each operation performed is counted as an individual sample (i.e., one application can be represented as multiple samples based on the operations performed.) For example, Docker has multiple command-line options available, such as `docker create`, `docker pull`, and `docker remove`. Similarly, Visual Studio Code (vscode) (Microsoft, n.d.), a GUI application for editing code and text, can perform operations, such as *file open*, *file delete*, and *file rename*. The different operations often present a distinct system call trace. Hence, the operations of applications with multiple options were also evaluated. Some applications

have been utilised multiple times to replicate different operations. For example, Visual Studio Code (VSCode) was used to replicate different file operations, such as file reading, file deletion, and file opening. Within the Linux benign dataset, different operations were treated as separate samples. The list of applications and their application type is as follows:

- Apache CLI
- apt CLI
- bash CLI
- bleachbit GUI
- cat CLI
- conda CLI
- cp CLI
- diff CLI
- docker CLI
- emacs GUI
- enpass GUI
- find CLI
- fslint GUI
- gcc CLI
- gedit GUI
- gimp GUI
- git CLI
- keepass GUI
- ls CLI
- make CLI
- mv CLI
- mysql CLI
- netbean GUI
- netcat CLI
- nimstall CLI
- nmon CLI
- npm CLI
- ping CLI
- python CLI
- rename CLI
- rm CLI
- ssh CLI
- tar CLI
- vagrant CLI
- vim CLI
- virtualbox CLI
- vscode GUI
- wget CLI

To acquire the malicious encryption-type Linux samples, two malware repositories were utilised, VirusShare (VirusShare, n.d.) and Malware Bazaar (Malware Bazaar, n.d.). VirusShare provides over 45 million malware samples identified in the wild and relies on VirusTotal (Sood, 2017) reports to categorise the samples. Furthermore, VirusShare provides a curated dataset consisting of ELF binaries (Linux's equivalent to Window's executables), observed from the years 2014 to 2020. This dataset consists of 43,553 samples, which was also considered. Whereas, Malware Bazaar has a corpus of over 500,000 malware samples with associated classification tags, such as family name or malware type.

Both VirusShare and Malware Bazaar repositories were searched for Linux bi-

naries and bash scripts. The repositories' search functions were used to conduct the search by finding potentially malicious hashes (SHA256 or MD5) associated with specific samples from different anti-virus vendors, specific ransomware family names, or unique naming conventions used by anti-virus vendors to identify ransomware. From this search, 46 Linux binaries and bash scripts with working encryption capabilities were identified.

For the VirusShare dataset, regular expressions were utilised on common substrings found in the classification names of anti-virus vendors, such as `rans`, `coder`, and `crypt` to automatically identify potentially malicious encryption-type ransomware. After searching through 20,000 samples using this method, one new sample was discovered. Hence, the dataset was no longer considered as the probability of identifying newer samples in the remaining dataset was very low.

The aforementioned searches produced a total of 47 samples consisting of Linux binaries and bash scripts that exhibited working encryption capabilities. The limited number of encryption samples was a result of several issues encountered while testing the samples. Similar to the Android Applications dataset, some samples were unable to run due to corrupted files or other code issues. Whereas, other samples, which specifically target ESXi servers, did not encrypt the emulated environment as the specific file or key was not found in the system. By further observing the 47 samples, it was discovered that the samples were split between 15 different families. This can be seen in Table 4.2, which shows the distribution of samples based on family. It is important to note that some families shown in Table 4.2 can often be classified as the same family due to the stark similarities they share with each other, such as Sodinokibi and REvil. The classification of families was derived from various anti-virus engines using VirusTotal. Hence, in

our Linux malicious dataset, these families were considered separate.

Due to the limited number of samples acquired from the aforementioned repositories, open-source ransomware projects on GitHub (n.d.) were also included to increase the sample size of the dataset. In total, 11 open-source ransomware projects were explored. Similarly, Table 4.3 shows the distribution of open-source ransomware by project name.

A notable open-source project was RAASNet (leonv024 and HugoLB, 2019), which can be configured differently based on the functionalities selected. In total, RAASNet contained four different forms of encryption, which are Ghost, Wiper, Pycrypto, and PyAES. The open-source project also provides two different file removal behaviours, which are *unlink and remove* and *overwrite and rename*. In addition to the different behaviours, there is an option to automatically remove the payload after the encryption process. A total of 16 different payloads were generated using different configurations of these functionalities. With the inclusion of the 16 samples from RAASNet and the aforementioned open-source ransomware the number of samples further increased by 26, which yielded a dataset consisting of 73 ransomware samples from 26 different families exhibiting encryption behaviours. The presence of the large number of RAASNET variants may introduce bias in the dataset. However, these samples were included to address the limitation of the limited sample size.

4.3 Acquisition of System Call Logs

Having obtained the samples, the next step is to execute them and observe the generated system calls. To achieve this, an emulated environment was used. Em-

Table 4.2: Distribution of Linux ransomware based on family

Ransomware Family	Sample Size
Babuk	12
Avoslocker	6
Dark Radiation	6
Sodinokibi	5
TellYouThePass	4
BlackCat	4
Cryptor	2
Polaris	1
REvil	1
Conti	1
Cerber	1
DarkSide	1
Generic.223935	1
Generic.220825	1
BlackMatter	1
Total	47

Table 4.3: Distribution of open-source ransomware based on project

Ransomware Family	Sample Size
RAASnet	16
Jimmy-ly00	1
Hash-glitch	1
Kishenkumar345	1
Gonnacry	1
Crypy	1
Bware	1
Wasper	1
Gowther	1
Ransom0	1
Cryptsky	1
Total	26

ulated environments have been a staple of many works surrounding dynamic malware analysis (Tam et al., 2015; Hull et al., 2019; Abbasi et al., 2022) as they can create a simulated computer system in an isolated environment. The isolated environment can safely analyse malware without the risk of compromising the host machine.

The work described in this thesis utilises Android and Linux emulated environments. Both emulated environments attempt to replicate a user file system. This was achieved by inserting trap files of common file extensions, such as `.txt`, `.png`, and `.jpg` stored within the user directory. As established, encryption-type ransomware aims to compromise the user's files through encryption. Hence, the trap files also function as an indicator of compromise, which can aid in capturing the malicious behaviour at a system call-level.

Having set up the emulated environments, each sample was executed and `strace` (Levin, n.d.) was used to capture system call logs. While each sample was running, simulated user interactions were inserted automatically and manually to replicate a more realistic user environment.

Having identified the similarities of both Android and Linux emulated environments, we now shift our focus to distinct details of each emulated environment. Starting with the Android emulated environment, the trap files included in the Android emulated environment initially covered a few common file extensions. However, the included trap files do not necessarily completely represent a realistic user environment as users often have more files of varying sizes and file types. Hence, the emulated environment was further improved in Chapter 6 and Chapter 7 by including a total of 105 trap files obtained from Kaggle (n.d.), which contains various user-created datasets often used for machine learning and data

science. The additional trap files have more varying file sizes with the additional file types `.mp3`, `.mp4`, `.cfg`, and `.xml`, which are also commonly encountered on real user smartphones (Du et al., 2021).

The process ID observed on `strace` for the Android samples, was the `Zygote` process, which is the main parent process on Android. This ensures that system calls from all running processes are captured and makes it possible to detect malware that uses multiple processes to avoid behaviour-based detection (Bidoki et al., 2017).

For the collection process of Android system call logs, Android Debug Bridge (Google, 2020a) was used to send commands to the emulator. Each sample was installed from its Android Package Kit (APK) file, under elevated privileges. Additionally, Android Monkey (Google, 2020b) was used to simulate user interaction.

The collection of system call logs for Linux used two emulated environments. One of those emulated environments was used to execute the sample, whereas the second emulated environment was used to capture the system calls via `strace`. All Linux samples were executed and interacted manually as specific arguments and parameters were required for the ransomware to run.

As previously established, both Android and Linux emulated environments use `strace` to capture system calls. However, the process observed was different. On Linux, the process observed was the bash terminal that was executing the sample. By observing the bash terminal, it is feasible to achieve a similar result to the Android `Zygote` process, which can capture the malicious process that was running.

4.4 Chapter Summary

This chapter described the dataset acquisition for Android and Linux encryption-type ransomware and the challenges encountered during the collection phase. Additionally, this chapter presented the different methodologies for acquiring system call data. The acquired data will be utilised for different experiments in the following chapters of this thesis.

Chapter 5

Detection of Encryption-type Ransomware using System Call based Behavioural Patterns[1]

Ransomware has been established as a prevalent issue and poses a serious cyber threat to society. Some researchers have trended towards the use of dynamic analysis with system call data to develop anti-malware and anti-ransomware solutions. However, none have focused on developing behavioural patterns for encryption-type ransomware on a specific platform. Hence, one of the research contributions outlined in Section 1 is to identify behavioural patterns at a system call level for encryption-type ransomware on Android devices.

This chapter addresses the aforementioned research contribution and presents a

[1]The material in this chapter is reproduced from a co-authored conference paper entitled ESCAPADE: Encryption-type-ransomware: System Call based Detection (Chew et al., 2020) with minor modifications to Section 5.2 and Section 5.3.4.

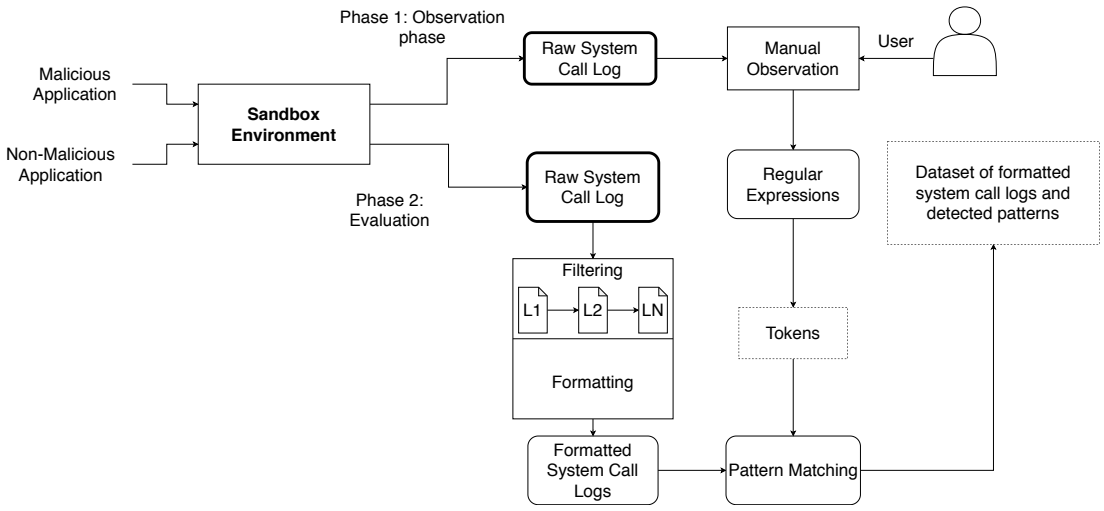

Figure 5.1: Methodology process overview

ransomware detection technique based on behaviours observed in the system calls performed by the malware and identifies a set of common behavioural patterns for encryption-type ransomware. Furthermore, the behavioural patterns are evaluated to assess the viability and efficacy of these patterns at detecting encryption-type ransomware behaviours from different families, to discover the shared common behaviour among encryption-type ransomware.

5.1 Methodology

Figure 5.1 provides an overview of the process followed in this work. The sandbox environment component is our run-time environment where applications are examined, which was previously described in Section 4.3. The sandbox environment was created using Android Studio, on a Google Pixel 2 (API level 24) with 2048MB internal storage, 512MB SDCard storage, and 1536MB of RAM. Whereas

the host machine was running MAC-OS, Intel Core i5 2.3 GHz Quad Core, with 8GB RAM.

The output of the sandbox environment splits into two phases. The first phase is the observation phase where applications are observed for their behaviour during runtime. After which, regular expressions are created based on the benign and malicious behaviours observed during that phase. These regular expressions are then converted into our token representation for pattern matching.

These tokens are used in the second phase, labelled as *Evaluation*. This phase starts with the extraction of the raw system call logs (similar to the observation phase), then applies multiple layers of filtering to abstract and remove repetitive or unrelated system calls. After which, the filtered log is formatted for pattern matching using our created tokens. This process is repeated for all unique variants containing a unique hash—also known as a sample—resulting into the final dataset, which contains the formatted system call logs and detected patterns. Examples of a true positive pattern match and false positive pattern match are provided in Listing 5.1 and Listing 5.2 in the filtered log format. The following subsections extensively describe our methodology of formatting and categorising system call data for detection of encryption-type ransomware in more detail.

Listing 5.1: Filecoder true positive pattern match for Write To File Unknown Extension

```
4179;23:18:22;openat;(AT_FDCWD, "U_DIR/large_text.txt.seven",
   O_WRONLY|O_CREAT|O_TRUNC|O_LARGEFILE, 0666 <unfinished ...>
4179;23:18:22;openat;( ) = 36
4179;23:18:22;fstat64;(36, <unfinished ...>
4179;23:18:22;fstat64;( {st_mode=0, st_size=1, ...}) = 0
4179;23:18:22;write;(36,"\10\261{H|\254\226\32\202\342\322\222\230
```

```
    \376c\256h\347\253\347v\271\"\303\265W\203\"\203\244\265T"...,
    148720 <unfinished ...>
```

Listing 5.2: Benign cache-cleaning application false positive pattern match for Read User File

```
4436;21:30:03;openat;(AT_FDCWD, "U_DIR/large_image.jpg",
    O_RDONLY|O_LARGEFILE) = 119
4436;21:30:03;fstat64;(119, {st_mode=0, st_size=1, ...}) = 0
4436;21:30:03;read;(119, <unfinished ...>
4436;21:30:03;read;( "\377\330\377\340\0\20JFIF\0\1\1\1\0d\0d\0\0
    \377\376\0LFile sou"..., 8192) pow2
4436;21:30:03;read;(119, "\344\6Q,\24\266\325j\333\244\312N\371#
    \2\247\236*\244\363Bx\2\356\f\235\205(\266\360.\7"
    ..., 8192) pow2
4436;21:30:03;read;(119, <unfinished ...>
4436;21:30:03;read;( "l\214\254\223\10\250\222H\356\304\366\2\275\
    r-\251S\342\273t\357\177\336\306\376\33G\315\225p
    \272\276"..., 8192) pow2
```

5.1.1 Detection of Behavioural Patterns

To acquire a set of high-level common behavioural patterns for encryption-type ransomware, we conducted an evaluation with 10 encryption-type ransomware samples from five families obtained from CICAndMal2017 (Lashkari et al., 2018) and Koodous (n.d.). Each application was executed 10 times and manually observed during runtime to comprehensively acquire their malicious behaviour. Additionally, 10 benign samples were also analysed to observe the differences in behaviour.

The five ransomware families used for our pattern observation phase consisted of: WannaLocker, DoubleLocker, SimpleLocker, Filecoder, and Wipelocker. The

selection of these five families were used as a pilot test aimed at acquiring system call-level behavioural patterns exhibited by encryption-type ransomware. All samples were evaluated from each of these families to acquire our common high-level behaviours. The samples used within our pattern observations phase are excluded from our dataset of malicious applications to avoid any potential bias within our evaluation phase in Section 5.3. However, all families, excluding Doublelocker, are still included in our malicious dataset. During the observation phase, we were able to discover 12 behavioural patterns. We classified the behavioural patterns in three categories, five of these patterns are classified as *Malicious*, four are classified as *Suspicious*, and three are *General* behavioural patterns.

5.2 Token Description

Table 5.1 shows the 12 patterns we identified and created. Within this table, the use of `>` is to concatenate each token. Additionally, Table 5.2 provides each token's objective. The following paragraphs provide more details of the representation of each pattern.

1 **Rename & Unlink File** – The first pattern created was *Rename & Unlink File*, the following pattern combination begins with a `renameat` system call. As this pattern observe multiple lines of system calls, the two tokens `N` and `ON` were used to capture the next line. The component argument of this system call requires the location of a file to rename, therefore the token `UD` was used to capture any renames within the user directory followed by the exclusion of the `Android` directory as this directory does not often contain user created files. The next system call observed was `fstat64`, which returns

Table 5.1: List of common behavioural patterns discovered and their token representation

Pattern Name	**Pattern Combination**
Rename & Unlink File	OP(renameat)>UD>\\(?!Android)>N>ON >OP(fstatat64) >N>ON>OP(unlinkat)>UD>A
Unlinking User Files	OP(unlinkat)>UD>MF
Unknown File Ext Created	OP(openat)>UD>UFC>A
Read User File	OP(openat)>UD>MF>(>AL>OP(read)>N>){3}
Write File Unknown Extension	OP(openat)>UD>UFC>AL>OP(write)>A
IPv4 Connections	OP(connect)>DQ
Directory Search	OP(openat)>MD>N>N>(>OP(getdents64)>N>)* >OP(close)>A
URL to Obfuscated Filename	OP(openat)>OF>(>OP(openat>)?>AL >OP(pwrite64)>AD
Socket Create and Connect	OP(socket)>SF>N>(>OP(socket)>N>)?>A >OP(setsockopt) >N>(>OP(setsockopt) >N>)>?>OP(connect) >N>(>OP(connect) >N>)>?>OP(fcntl64) >N>(>OP(fcntl64)>N>) >?>OP(fstat64) >N>(>OP(fstat64)>N>) >?>OP(write)>GA
File Write	OP(openat)>AL>OP(write)>A
File Read	OP(openat)>AL>OP(read)>A
Generic File Unlink	OP(unlinkat) (.*?(.(\w+)(\bflock|xml|bak|db-wal\b)\").+)

Note: Some sub-patterns were retained as a regular expression as certain parts are too specific to be represented as tokens.

Table 5.2: List of token names and their respective pattern

Token	Pattern Purpose
OP	System call operation
AL	All including newline
UD	User directory
N	Newline
ON	Optional match newline
A	Match all
UFC	Unknown file creations
DQ	Dotted quad formats (i.e., IPv4)
AD	URL address
OF	Obfuscated file
SF	Socket flags
GA	Get address info
MD	Match directory
MF	Match file (regular file with one extension)

information regarding a file; finalising with the `unlinkat` system call and `UD` token, which represents the unlinking of a file in the user directory.

2 **Unlinking User Files** – *Unlinking User Files* is a pattern that observes a single line of system call. This pattern contains three core tokens, which represent the system call operation, directory location, and file observed. This is represented with the pattern combination beginning with the `unlinkat` system call followed by the token `UD` and `MF`, which represents the unlinking of a file with any file extension within the user directory.

3 **Unknown File Ext Created** – The *Unknown File Ext Created* pattern also

observes a single line of system call. This pattern begins with the `openat` system call followed by the user directory token `UD`, which represents the observation of open operations in the user directory. The token is subsequently followed by `UFC`, which observes the generation of unknown file extensions (e.g., excluding common extensions like `.jpg`, `.png`, and `.txt`). Concluding the sequence is the token `A`, designed as a wildcard to match any content within the line.

4 **Read User Files** – Pattern *Read User Files* observes multiple lines of system calls, which begins with the `openat` system call followed by the `UD` and `MF` system call, which denotes the opening of a file containing a single extension within the user directory. This is then followed by the tokens `AL` for matching all lines, the system call `read`, and the token `N` for matching a newline. These three tokens are subsequently repeated 3 times, which is annotated with the brackets followed by the number 3 encapsulated within braces. The 3 subsequent `read` tokens were derived from observing the way that benign and malicious applications read files. Generally, from the observation of the system calls logs, malicious applications read user files in specific block sizes, whereas this occurrence is less frequent in benign applications.

5 **Write File Unknown Extension** – The following pattern, *Write File Unknown Extension*, observe two lines of system calls. However, it follows the same set of token combinations as *Unknown File Ext Created* with the addition of a `write` system call followed by the wildcard token `A`.

6 **IPv4 Connections** – *IPv4 Connections* observes the `connect` system call, which is used to establish a connection to a socket. This connection is estab-

lished by specifying an address in the argument. Hence, the token `DQ` was used to match an IPv4 address in a dotted quad format.

7 **Directory Search** – *Directory Search* matches multiple lines of system calls beginning with the `openat` system call, this system call is accompanied by the tokens `MD` and `N`. This combination of tokens is used to denote the opening of a directory in any location. Subsequently, a group of system calls with `getdents64` by the token `N` is used to capture the directory entries. The final system call observed in this pattern is `close` followed by the token `A` to signify the closing of the directory search.

8 **URL to Obfuscated Filename** – The pattern *URL to Obfuscated Filename* observes the `openat` system call with the `OF` token, which represents an opening of an obfuscated file. This is followed by an optional match for any `openat` system call, which is signified by encapsulating the system call in brackets followed by the symbol `?`. Following the optional match is the system call `pwrite64` with the token `AD`, which reads or writes to a given file descriptor in the form of a URL address.

9 **Socket Create and Connect** – Another pattern related to network connection behaviours was *Socket Create and Connect*. This pattern observes multiple system calls. The pattern starts with the `socket` system call followed by tokens `SF` and `N` to signify an established communication with a specified endpoint. This is followed by an optional match for the `socket` system call. The next set of system calls observed in this pattern was `setsockopt` with the token `N`. This system call was used to set or get the socket options. Similarly, the system call is followed by an optional match. The next se-

quences observe the `connect`, `fcntl64` and `fstat64` system calls with each system call containing an optional match, finalising with the `write` system call and the token `GA`, which is used to represent the acquisition of network information.

10, 11 **File Read and File Write** – *File Read and File Write* share similar pattern combinations with both of the pattern starting with an `openat` system call with the token `AL` and subsequently end with the system calls `read` or `write` with the token `A`.

12 **Generic File Unlink** – The final pattern created is *Generic File Unlink*. This pattern observes the `unlinkat` system call and a regular expression to match any file that matches the common file extensions `flock`, `xml`, `bak`, and `db-wal`, which were often removed from benign applications based on observations in the system call logs.

5.2.1 Pattern Acquisition and Classification

The method of acquiring the patterns was based on manual deduction in the observation phase. This was achieved by going through each application and identifying malicious (or potentially malicious) behaviour and its respective high-level system call counterpart via the captured log. For example, if an application encrypted the user's files then the high-level behaviour at a system call level would translate to `openat` - open user file, one or multiple `read` system calls, `openat` - create new encrypted file, one or multiple `write` system calls.

We aim to observe common high-level behavioural patterns specifically focusing on encryption-type ransomware. However, not all captured behavioural patterns

correlate to malicious behaviour.

For example, consider the creation of a socket to connect to an external URL to transfer specific resources. This type of behaviour occurs in both benign and malicious applications. However, the usage will differ. A malicious application often uses that connection to contact a Command and Control (C&C) server (Lipovský et al., 2016) to download the payload, whereas a benign application would use the connection to download resources; often occurring in applications requiring frequent updates, such as online mobile games, or linking accounts such as social media accounts. Therefore, to aid in distinguishing the behaviour of patterns, we created a classification to better represent the patterns detected.

Patterns in the *Malicious* category are explicitly classified as malicious behaviours. Applications that contain *Malicious* patterns contain malicious segments that resemble behaviour of encryption-type ransomware, such as unlinking or renaming a file within the user directory.

Behavioural patterns classified in the *Suspicious* category are deemed as potentially malicious. These types of patterns can lead to malicious behaviour. However, the behaviour by itself does not indicate any malice. An example of a *Suspicious* pattern is the connection of an IPv4 address.

Patterns in the *General* category are common benign behaviours that exist in malicious and benign applications with low indication of malicious behaviour, such as a file read or file write from any directory.

Note: *Suspicious* and *General* patterns are not used in our evaluations for this work. These patterns were primarily identified and created to aid future detection systems that utilise common high-level behaviour. Furthermore, encryption-type ransomware exhibits distinct malicious behavioural patterns unlike other types of

malware, such as Adware and Trojans, where the malicious behaviours are not always immediately evident. The inclusion of these two pattern categories will be more beneficial in those types of malware.

5.2.1.1 Malicious Patterns

Our first malicious pattern observed from the logs was related to file renaming and unlinking within the user's main directory (*Rename & Unlink File*). This behaviour was observed in the WannaLocker/Slocker sample, which renamed the initial encrypted file using an unknown file extension. Once the file extension has changed, the ransomware proceeded to unlink the user's original file that was related to the encrypted file. We only looked for this pattern in files within the user directory or external directory (SDcard) as these directories are the points of interest for encryption-type ransomware due to the importance of the files residing within them (often important to the users, such as photos, notes, and other important documents, but not required for the system to work) (Song et al., 2016). Additionally, during our observation phase, the folder *Android* was also within the user directory. Hence, we added a condition to exclude that specific directory.

The next malicious pattern from our observations was unlinking of users' files. From our analysis, we were able to find consistent occurrences of this pattern in the ransomware samples and there were no traces of this pattern occurring in the 10 benign samples during our observation phase.

Another malicious behavioural pattern discovered was the creation of files with unknown file extensions within the user's main directory (*Unknown File Ext Created*). From the different samples observed, this was a prevalent file encryption behaviour of ransomware where a new file was created to hold the encrypted data

of the original user's file. This encrypted file was in a nonstandard file extension and the file name consisted of the original file's name including its original file extension.

The last two common malicious patterns discovered were reading of user files and writing to a file with an unknown file extension. These two behavioural patterns represented the encryption segment of an encryption-type ransomware. This was a common behaviour that occurred in all of our ransomware logs.

The first pattern that represents the encryption component is *Read User File*. This pattern focuses on capturing the behaviour of applications reading three times from a file within the user directory. From our observation phase, some of the malicious variants observed read the contents of files within the user directory over multiple `read` operations in a specific block size, unlike the benign samples, which read the file contents in one single block. Hence, the inclusion of three read operations; this is to filter out apparent benign applications.

The second pattern of the encryption component is *Write File Unknown Extension*. This pattern observed the behaviour of applications writing data to a newly created file with an unknown file extension. This pattern, together with *Read User File*, represented the encryption behaviour seen from the various encryption-type ransomware in our observation phase. Figure 5.2 provides an abstracted example of our process for modelling the aforementioned malicious behavioural patterns using regular expressions. We utilised a similar process for Suspicious and General patterns.

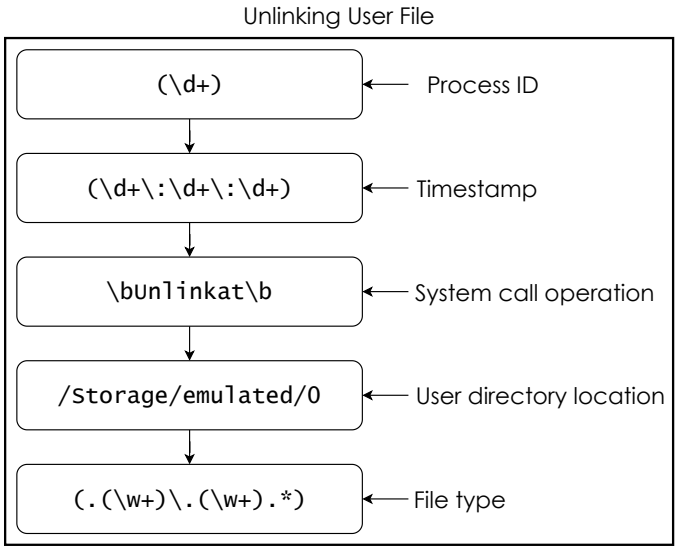

Figure 5.2: Abstract view of representing 'Unlinking of user files' malicious pattern using regular expressions

5.2.1.2 Suspicious Patterns

The first suspicious pattern we noted was applications making connections to an external IPv4 address. This could mean the malicious app making connection to a C&C server. However, this can also just be a non-malicious app connecting to the outside internet. We, therefore, classified as suspicious but not malicious. IPv6 addresses were not considered as IPv4 addresses were the most prevalent occurrences observed in our initial observation phase. Furthermore, our main methodology of pattern matching utilises regular expressions as such, it would be challenging to represent IPv6 addresses as regular expressions. The sequence of this pattern observes any `connect` system call followed by an IPv4 address.

Another suspicious behavioural pattern was directory searching. This behaviour is traditionally exhibited by encryption-type ransomware, which searches

for user files within the device to encrypt. However, this behaviour does not inherently signify malicious behaviour as there are benign applications that can exhibit the same behaviour, such as cache-cleaning applications.

The next notable suspicious pattern discovered in some ransomware samples, was the creation of an obfuscated file. This file had no file extension and the content contained an external URL. Similar to the first suspicious pattern, we were unable to validate the legitimacy of the URL address. However, many of the ransomware logs observed, contained URL addresses that were related to C&C servers.

The last suspicious pattern was the acquisition of network information via `getaddrinfo`. From our observations, the majority of the ransomware logs attempted to acquire network information, such as socket addresses, and socket types from unknown domains via `getaddrinfo`. However, this does not necessarily indicate malice as we discovered legitimate trusted domains in benign applications such as, `googleadservices`.

5.2.1.3 General Patterns

There are three patterns in the *General* category. These patterns consist of simple file I/O operations, read and write file behaviour, and generic file unlinking (targets known file extensions in any directory location), such as temporary files (`.tmp`, `_tmp`), backup files (`.bak`), or file locks (`.flock`).

The patterns in the *General* category aim to provide more detailed information regarding an application's behaviour regardless of whether the application is malicious or benign.

For *File Read*, and *File Write*, the sequence started with an `openat` system

call, then a read/write operation. The last pattern *Generic File Unlink* matches any *unlinkat* system call with any file matching *.flock*, *.xml*, *.bak*, or *.db-wal*.

One of the objectives of this work was to identify common high-level behavioural patterns for encryption-type ransomware at a system call level. To satisfy this requirement, we identified 12 different behavioural patterns, represented as tokens, and categorised them into three severity levels based on our observations. The use of behavioural patterns can be a challenging task for binary classification as benign and malicious applications can often exhibit similar behavioural patterns. Hence, the severity levels aimed to address this issue by categorising the identified behavioural patterns based on their relevance to encryption-type ransomware behaviours. By utilising these patterns with the methodology for collecting and extracting system calls in Section 4.3, we were able to devise a meta language for detecting malicious encryption-type ransomware behavioural patterns. This approach presents an easily reproducible testbed for researchers to create behavioural patterns based on system call logs.

5.3 Evaluation

This section details the method of evaluation used in this work, which includes the evaluation of detected patterns identified in a set of encryption-type ransomware and benign applications. These evaluations are conducted to identify shared commonalities that exist between different encryption-type ransomware families as well as assessing the viability against a benign set of applications.

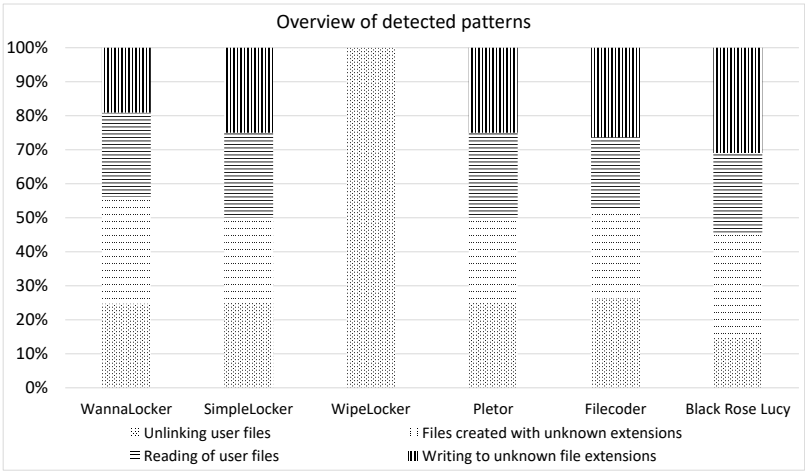

Figure 5.3: Overall results of each detected pattern

5.3.1 Evaluation Method

We ran each application for two minutes using our automation script. Based on our initial observations, all malicious Android applications evaluated were able to exhibit malicious activity within the two minute threshold. Once all the system calls were extracted, we put them through our detection program, and calculated the number of all detected patterns for the different severity levels.

For our ransomware dataset, we identified different malicious patterns for all six ransomware families. Any application whose log contained a match for at least one malicious pattern was classified as malicious. Any falsely identified malicious patterns were noted within this evaluation.

5.3.2 Detected Patterns

This section details our evaluation of the six different encryption-type ransomware families. Figure 5.3 illustrates the results of our evaluation for the malicious

dataset. The following paragraphs provide a thorough elaboration of each family and their discovered patterns.

WannaLocker: For WannaLocker we acquired 51 samples; from these 51 samples, we detected 850 malicious patterns. *Unlinking User Files* and *Read User File* were detected 211 times each, *Write File Unknown Extension* was detected 162 times, and 266 patterns were *Unknown File Ext Created*.

SimpleLocker: We acquired 64 encryption-type ransomware samples of SimpleLocker, and out of these 64 samples, we were able to discover 1280 malicious patterns. Within the 1280, we detected an even split of 320 *Unlinking User Files*, *Unknown File Ext Created*, *Read User File*, and *Write File Unknown Extension*. However, we were unable to detect any *Rename & Unlink File* as this behaviour did not occur in any of the samples.

WipeLocker: For WipeLocker, 70 samples were acquired for evaluation. All 70 samples detected 5 *Unlinking User Files* with no other malicious patterns detected. This led to a total of 350 malicious patterns detected. Although WipeLocker did not indicate any behaviour of file encryption (even after re-evaluating the applications manually), the attributes this family exhibited were similar to encryption-type ransomware such as, the detected pattern of unlinking user files, and directory searching (a suspicious pattern, which we were able to detect a total of 733 occurrences); hence, the inclusion of this family within our evaluation.

Pletor: We were able to acquire six samples from Pletor and from those six samples, a total of 120 malicious patterns were discovered; with even split of 30 between *Unlinking User Files*, *Read User File*, *Write File Unknown Extension*, and *Unknown File Ext Created*.

FileCoder: For Filecoder, we were only able to acquire five samples. However,

out of these five samples, we were able to discover 95 malicious samples. From those 95 patterns, *Unlinking User Files* and *Write File Unknown Extension* were split evenly with 25 total detected samples each,20 were classified as *Read User File*, whilst the remaining 25 were classified as *Unknown File Ext Created*.

Black Rose Lucy: For Black Rose Lucy, we acquired 17 samples for our evaluation. Out of these 17 samples, 307 malicious patterns were detected. Out of these, we identified 45 instances of *Unlinking User Files*, 95 *Unknown File Ext Created*, 72 *Read User File*, and 95 *Write File Unknown Extension*.

Unlike other encryption-type ransomware, we noticed that Black Rose Lucy specifically targeted the user's external storage directory (`/sdcard/`) rather than the user's internal directory during our evaluation. Additionally, we required manual interaction with each of the samples as Android Monkey was unable to detect the package name of the application.

One of the objectives of this work was to evaluate the viability of the devised patterns for behavioural pattern detection against a set of encryption-type ransomware. Within our evaluation, we were unable to discover any patterns for *Rename & Unlink File* as this behaviour was likely tied to a specific variant of WannaLocker or SimpleLocker. However, from the overall results of our evaluation, seen in Figure 5.3, there is clear indication of shared common behaviour among encryption-type ransomware regardless of the family, with the only exception of WipeLocker, which is known to only remove user files. Through the patterns detected and shared commonalities identified, we have validated the viability of these common high-level behavioural patterns for detection of encryption-type ransomware.

Table 5.3: Summary of all benign applications evaluated

Benign Samples	Percentage	Sample Size
True Negative	98.6%	495
False Positive	1.4%	7

5.3.3 Benign Applications Results

Table 5.3 contains a summary of our results where we evaluated the efficacy of our patterns on our benign dataset. The *Percentage* column provides the percentages of true negatives and false positives detected for all benign samples evaluated. The *Sample Size* column denotes the numerical value of true negatives and false positive samples detected.

To evaluate the efficacy and viability of our patterns, we tested our approach on a dataset consisting of 502 benign applications. Two of those are the cache-cleaning applications discussed separately below. Out of the other 500 benign applications, we encountered six falsely classified applications. This was due to a mismatch of four different patterns, specifically, *Unlinking User Files*, *Read User File*, *Unknown File Ext Created*, and *Write File Unknown Extension*.

For our pattern matching results, two applications incorrectly matched *Read User File*; this was due to the applications creating and reading application related files within the user directory, such as `dslv_state.txt`. To mitigate this issue, `openat` system calls with the flag `O_CREAT` could be excluded. This would ensure that only user created files were captured within this pattern.

The third benign application that was falsely classified incorrectly matched

the patterns *Unlinking User Files* and *Read User File*, due to the application creating and utilising temporary files within the user directory. This was one of the drawbacks of capturing high-level behaviour. For most cases, these patterns would capture unlinking of user created files and existing user file access and reads, which is a behaviour often exhibited by encryption-type ransomware as part of the file encryption process. However, in the case of an application creating and utilising a file within the user directory, it would be classified as a false positive. A potential solution is to exclude files created by the application within the user directory, as previously suggested, or reduce and combine the behavioural patterns related to file encryption.

The last three benign applications falsely classified were incorrectly matching two behavioural patterns: *Unknown File Ext Created* and *Write File Unknown Extension*. These patterns were falsely classified due to the applications creating an application folder within the user directory and a file with an unknown file extension within the application folder.

Similar to the proposed solution for the aforementioned third application, combining behavioural patterns related to file encryption could provide a more accurate representation. Alternatively, the pattern could be altered to only check for primary directories (i.e., directories not created by the application), such as *Photos*, *Documents*, and *Downloads*.

5.3.3.1 Cache Cleaning Applications

For the two cache cleaning applications, one of them resulted in a false positive. There were four total malicious patterns matched and all four of those patterns were linked to *Read User File*.

From the examination of the patterns file and system call log file, these four patterns were deemed as irregular behaviour as it was unusual for a benign application to be reading the contents of user created files (i.e., pre-existing files, not created by the application).

5.3.4 Discussion

This section explores some of the limitation of the aforementioned work. One of the limitations relates to the generation of regular expressions. Currently, we require manual observation and interaction to create regular expressions. This process can often be tedious and difficult. As we continue to develop our approach, we intend to automate this process.

Another limitation is our approach of identifying behavioural patterns. The patterns identified were based on our observations from various applications. As a result of this, there may have been some behaviours that were not captured. In future, we would like to introduce a more formalised and robust methodology of identifying behavioural patterns at a system call-level ensuring that all behaviours are captured without any uncertainty.

Additionally, we intend to introduce more behavioural patterns capable of detecting other types of malware, such as Backdoors and Trojans, which were identified as two of the most prominent types of infections for third-party apps (Google, 2019b). This enables us to expand our dataset and evaluate the efficacy of our methodology on a larger sample size consisting of different types of malware.

Many dynamic anti-malware solutions often utilises isolated environments, most commonly through the use of Virtual Machines (VMs), to contain and analyse

malware. This presents an issue as more contemporary malware are equipped with the capabilities to detect and evade those environments (Gadhiya and Bhavsar, 2013; Uppal et al., 2014). Although the experiments conducted in this thesis were evaluated on VM, the work has shown to be feasible to implement on a real device. In future, a potential avenue for improvement is to develop a system that incorporates the concepts proposed in this thesis, which can efficiently capture system calls in real time on a real user device whilst adhering to the resource constraints of mobile devices.

As previously mentioned in Section 5.2.1, *Suspicious* and *General* patterns were not utilised in our evaluations. However, these patterns were still identified and created to lead into future work. These patterns can be expanded to create a more robust real time malware detection model for Android devices, or aid current and future anti-malware solutions in detecting and deterring malware.

The use of system calls enables the ability to capture large quantities of information, which can often be used to associate behaviours exhibited by an application. However, by further exploring system calls, it was observed that more complex behaviours are difficult to capture and understand at a system call level. One of the examples was locker-type ransomware behaviour and SMS Trojans. The core mechanism of a locker-type ransomware is to restrict the users' ability to access their devices, often times, this restriction is imposed by a perpetual overlay or window, which cannot be closed. Conversely, SMS Trojans send or intercept SMS messages for malicious purposes (e.g., stealing of credentials and involuntary subscription to premium services). At a system call-level, these types of behaviours are often handled by the `ioctl` system call, which observes binder transactions. Due to the complexity of binder transactions, the `ioctl` system calls are not easily

understandable.

Based on the understanding of this limitation, it would not be feasible to accurately identify and understand the complex behaviours of specific types of malware by solely relying on system calls alone. Further additions, such as frequency analysis (Bhatia and Kaushal, 2017), specialise decoding of Binder transactions (Tam et al., 2015), or additional observable features, such as permissions (Ferrante et al., 2017), would be necessary to produce a more descriptive overview of an application's behaviour. Nevertheless, solely utilising system calls and its sequence does not inhibit the ability to observe behavioural patterns in general. However, this is a limitation that should be considered for more complex behaviours or specific malware types.

The use of dynamic analysis methods are more resilient to common static obfuscation techniques, such as code obfuscation, and junk code insertion. However, it should be noted that obfuscation techniques have been effectively explored to invalidate existing dynamic analysis methods. A dynamic analysis obfuscation technique of particular interest is system call obfuscation. Srivastava et al. (2011) proposed an *Illusion* attack that utilises an Alternative System Call Execution Path (ASEP) and the `ioctl` system call to obfuscate malicious behaviour. The proposed method showed that it was possible to masquerade the behaviours performed by malicious applications as the system calls invoked through the use of `ioctl`, which is difficult to discern from benign applications due to the marshalling process, unless a specialised decoding process was implemented.

In this work, the use of regular expressions to devise behavioural patterns have been shown to effectively detect encryption-type ransomware. However, regular expressions have its limitations, particularly relating to more complex pattern

matching, such as nested parentheses, and counting or checking for balanced sets of characters. In the following chapters of this thesis, this issue is addressed by deviating from the prevalent use of regular expressions in the behavioural patterns.

5.4 Chapter Summary

In this chapter, we identified and explored different system call level behavioural patterns for encryption-type ransomware. To achieve this, we presented an extensive methodology for collecting and identifying behavioural patterns at a system call level. Using this methodology, we were able to discover a set of common high-level behavioural patterns at a system call level.

Additionally, the effectiveness of behavioural patterns identified was evaluated. This was achieved by creating 12 behavioural patterns for detecting encryption-type ransomware. Consequently, this chapter evaluated the patterns against a set of encryption-type ransomware to identify shared commonalities between different families using pattern matching. The methodology and behavioural patterns presented in this chapter contributed and further extends to dataset mentioned in Chapter 4 by including the formatted system call logs of encryption-type ransomware.

Chapter 6

Real Time System Call based Ransomware Detection

The threat landscape of malware is ever-evolving, and the techniques used by malware developers have become more intricate over the years. To counteract the evolving landscape, some researchers have shifted towards the use of real-time malware analysis as it can offer the benefit of immediate and active monitoring of malware. One of the research contributions outlined in this thesis is to leverage the benefits of real-time malware analysis and system call level analysis to dynamically identify behavioural patterns of encryption-type ransomware on mobile devices.

This chapter further extends the work mentioned in Chapter 5 by adopting a proof of concept real time streaming approach. The following sections describe the revised methodology for a real-time streaming approach to detect encryption-type ransomware. This is achieved by using a 2 layer token Finite State Machine (FSM) approach, which distinguishes between individual and sequential behavioural patterns. The new methodology was compared to the previous methodology in Chap-

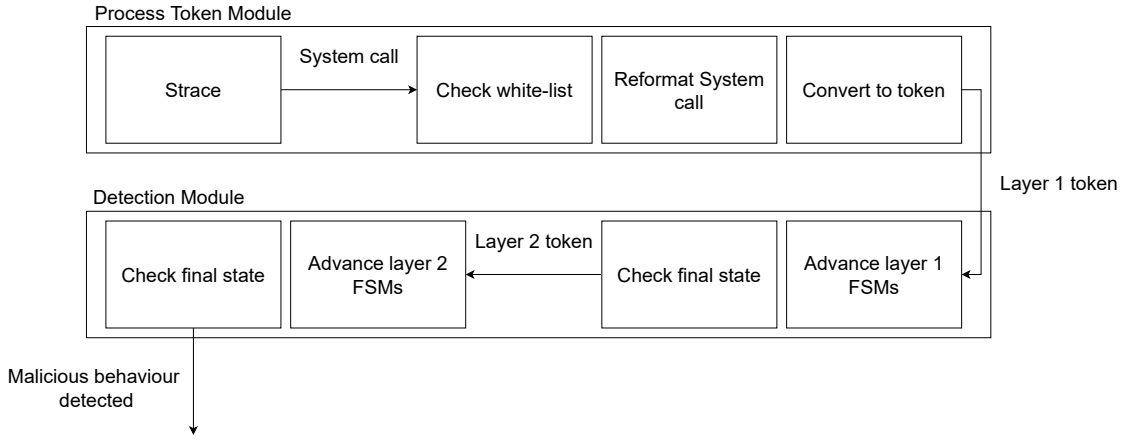

Figure 6.1: Block diagram for streaming approach

ter 5 to evaluate its efficacy. Furthermore, the performance of the new approach was evaluated and shows that it is feasible to run on an Android operating system.

6.1 Implementation with streaming system calls

In the previous chapter (Chapter 5) an offline methodology for detecting encryption-type ransomware was proposed, which utilises system call data. The main limitation of the approach is the offline data collection process, which is not scalable and not indicative of a real-world scenario where data and information is constantly generated in real time. We improved this through a new streaming architecture, where each line of system call generated by `strace` is processed in real time. This approach consists of two primary modules, Process Token Module, and Detection Module. Figure 6.1 provides an abstract overview of our proposed approach, with the following subsections further elaborating on each module.

6.1.1 Process Token Module

To stream the system call data (i.e., capture the system call data in real time), we used a similar environment as previously mentioned in Chapter 4, Section 4.3, which adopts Android Debug Bridge (ADB) and `strace` on an Android emulator running Android 7.0 Nougat (API level 24). The process observed using `strace` was the parent process (Zygote), which allows us to capture a broad range of behaviours, such as the application's behaviour and application to Operating System (OS) interactions occurring within the device. System calls produced by `strace` are sent to the Process Token module, which checks if it is a white-listed system call, then formats the system call with a separation character (`;`) and converts it into a unique token for the Detection Module. By adopting a streaming approach, we were able to provide a more realistic, real-world, evaluation of our offline approach of using system call behavioural patterns to detect encryption-type ransomware in real time.

Not all system calls recorded by `strace` are relevant to the behaviour of an application of interest. For example, `clock_gettime()` is used to obtain the system clock time and `gettimeofday()`, can acquire the current time and the timezone, irrespective of application behaviour. We filtered out system calls following a similar method of filtering unrelated system calls from our offline approach, which was mentioned in Chapter 5, Section 5.1, to the streaming process. We improved this process by white-listing a smaller subset of system calls used for encryption-type ransomware (e.g., open, write, read). By applying the white-list, the processing and detection time are further reduced as system calls that are not in the white-list will not be processed.

After the initial filtering process, each system call was formatted using the separation character ; for easier token conversion (e.g., <pid>;<timestamp>;<system call>;<arguments>), then converted into unique tokens to be utilised by the FSMs (i.e., token FSMs) in the Detection Module. This was done to reduce the number of state transitions required. The conversion process condensed each system call into a unique token. To convert system calls into tokens, we developed a set of unique tokens (provided in Table 6.1), derived from regular expressions, that matched each system call based on the operation and system call arguments.

6.1.2 Detection Module

The Detection Module utilises the behavioural patterns previously discussed in Chapter 5, Section 5.2.1. These behavioural patterns are converted into token FSMs, which are used in our detection phase. As each token is streamed from the Process Token module, the Detection module validates the current token against a set of FSMs. In this module, the proposed method includes two layers of finite state machines to acquire a more precise detection model for encryption-type ransomware. *Suspicious* and *General* patterns were not used in the Detection Module except for *Directory Search*, as those patterns did not provide additional benefits in the process of detecting malicious activity with this proposed implementation.

The first layer of FSM consists of individual behavioural patterns previously mentioned in Section 5.2.1. These behavioural patterns were converted into a more compact and generalised FSM to reduce the time taken to detect behaviour. It needs to be kept in mind that generalisations like this can increase the likelihood of false positives.

Table 6.1: Token representations of systems calls

Tokens	**Text Representation**
O_U_CREATE	Open create unknown file extension
O_UDIR_FILE	Open user file
O_UD	Open user directory
RD	Generic read
O_OBF	Open obfuscated filename
S	Generic socket
W	Generic write
O	Generic open
C_DQ	Connect to dotted quad address
SS	Generic setsockopt
FC_64	Generic fcntl64
W_GA	Write getaddrinfo
FS_64	Generic fstat64
GET_ENT64	Get entries in directory
U_UDIR	Unlinking file in user directory
G_U	Generic unlink
PW_64	Generic pwrite64
PW_64_AD	Pwrite64 URL
RN_UDIR	Rename file in user directory

Encryption-type ransomware follows a distinct and common sequence of behaviours. Hence, to further distinguish the differences between malicious and benign behaviours we have devised a second layer of FSMs, which determines if the sequence of matched patterns corresponds to the sequence of behavioural patterns exhibited by encryption-type ransomware. The second layer of FSMs represents

the sequential occurrence of behaviours observed in encryption-type ransomware (i.e., combination of layer 1 FSMs). The second layer FSM will only be checked if the first layer FSM matches a pattern (i.e., a layer 1 FSM has reached a final state). The state transition of a layer 2 FSM is the layer 1 FSM behavioural pattern name (e.g., `Unlink user file`, `General unlink`).

6.1.2.1 Creation of layer 1 FSMs

Layer 1 FSMs are based on previously discovered encryption-type ransomware behavioural patterns. However, as mentioned in Section 6.1.2 they were generalised and compacted through the utilisation of tokens. To acquire the token FSMs, we simplified the expanded regular expressions by removing fine-grain details, such as timestamps, newline matches (`\n`), and multi-line matches (`((.|\n)*?)`) as these matches were no longer due to the real time streaming approach, which processes one token at a time rather than iterating over multiple lines of system calls. After this simplification, the system calls and their respective arguments used in the regular expression were converted into a unique token as previously explored in Section 6.1.1. Through this process of generalisation and compaction, we acquired tokenised FSMs. Figure 6.2 shows an example of this process, which takes the offline tokenised regular expression and expands it to the full regular expression. This is done to remove the fine-grain details, thus resulting in a more compact regular expression. After removing the fine-grain details, the regular expression is converted into a unique token, which is then created into a layer 1 token FSM.

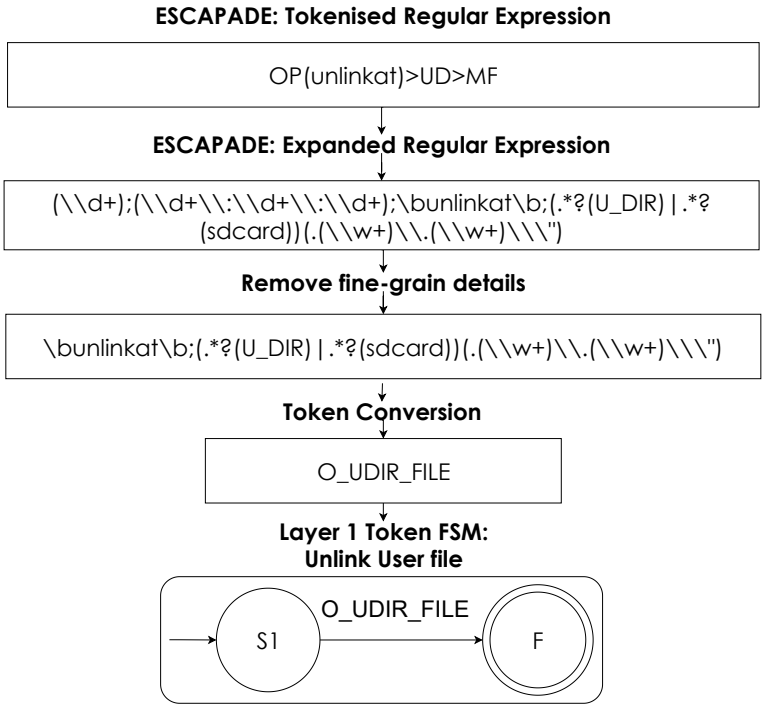

Figure 6.2: Transformation of behavioural patterns to layer 1 token FSM

6.1.2.2 Creation of layer 2 FSMs

Layer 2 FSMs focus on behaviour sequences (i.e., sequence of behavioural patterns from layer 1 FSMs). As previously mentioned, encryption-type ransomware exhibited distinct sequences of behaviours. To acquire the specific sequences of behaviours, we randomly selected six samples from six different ransomware family (one sample from each family) and manually observed the sequence of layer 1 FSMs detected. From this observation, we acquired 4 distinct sequences of behaviours commonly exhibited by encryption-type ransomware as shown in Table 6.2. The table shows the four distinct sequences of behaviours; the symbol `>` is used to show the concatenation of individual behaviours (e.g., `Directory Search > Unlink User File` means a directory search behaviour followed by another be-

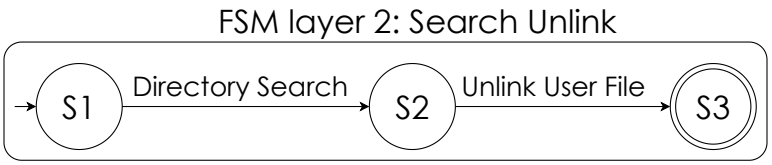

Figure 6.3: Layer 2 FSM example for search Unlink

haviour, which unlinks user files). If one of these sequences is discovered in the 2nd layer of FSMs, the application is considered malicious. Figure 6.3 shows an example of a layer 2 FSM.

The streaming approach described in this section addresses the limitations of the previous offline approach by establishing an improved processing and detection system. This approach adopted the previously defined behavioural patterns, and created a real time detection system utilising a 2 layer FSM, which observed individual behavioural patterns and sequences of behavioural pattern, thus further validating the first half of our fourth research objective. In the following section, we evaluate the improvements of this streaming implementation compared to the previously established offline approach.

6.2 Evaluation

In this section, we present the results of our comparison between the streaming implementation, which observed system calls in real time and utilised a two layer FSM approach to detect behavioural patterns, and the offline approach, which observed system call logs to detect behavioural patterns. Furthermore, this section describes the methods used to evaluate our approaches, and the results of our experimentation, which consist of the detected malicious patterns, false positives

Table 6.2: Sequence of common behaviours exhibited by encryption-type ransomware

Pattern Name	**Behaviour sequence**
Search Read Unknown Create Write	Directory Search > Read User File > Unknown File Creation > Write Unknown File Extension
Search Read Unlink	Directory Search > Read User File > Unlink User File
Search Unknown Create Write	Directory Search > Unknown File Creation > Write Unknown File Extension
Search Unlink	Directory Search > Unlink User File

within benign applications, and the overhead incurred by the streaming approach.

The environment used in our evaluation was running MAC-OS, Intel Core i5 2.3 GHz Quad Core, with 8GB RAM. The Android emulator was created using Android Studio, and the emulator environment was a Pixel 2 running API level 24, Android 7.0 (Google APIs), with 2048MB internal storage, 512MB SDCard storage, and 1536MB of RAM.

6.2.1 Evaluation Method

To evaluate the offline approach, we ran each application for two minutes using our automation script. As mentioned in Chapter 4, this work utilises the inclusion of additional trap files. Hence, the application runtime has been increased to two minutes as opposed to the initial one minute mentioned in Chapter 5. This automation script installs and starts the applications and utilises Android Mon-

key Google (2022a) to inject random events to simulate real user interaction. Once all the system calls were extracted, we put them through our detection program, and calculated the number of all detected patterns for the different severity levels. A similar method was utilised for our streaming approach. However, rather than collecting system call logs, we piped the output of `strace` into our implementation and measured the number of layer 2 FSM matches (i.e., sequential behavioural patterns). We identified various malicious patterns for all six ransomware families. Any application containing a match for at least one malicious pattern, for the offline approach or one layer 2 FSM match, for the streaming approach, was classified as malicious. Any falsely identified malicious patterns were noted within this evaluation.

This section details our evaluation of the six different encryption-type ransomware families. Figure 6.4, shows the individual malicious patterns detected in the offline approach and Figure 6.5 shows the sequence of malicious patterns detected using the streaming approach. Although different patterns were utilises in the detection process (offline uses individual behavioural patterns, whereas streaming uses sequences of malicious behavioural pattern), the two figures indicate a similar outcome in detected behavioural patterns for encryption-type ransomware. This similarity shows that the streaming approach with an altered detection method, using sequence of behavioural patterns, is capable of successfully identifying shared common behavioural patterns in encryption-type ransomware and is comparable to our offline approach.

One of our research objectives was to evaluate the feasibility of the devised patterns for behavioural pattern detection against a set of encryption-type ransomware. The results of our evaluation in Figure 6.4 and Figure 6.5, provide vis-

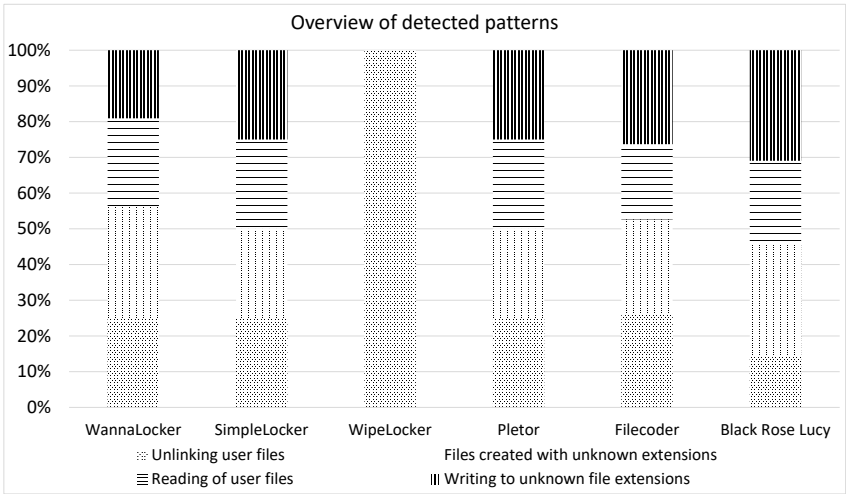

Figure 6.4: Offline: Malicious behaviour results

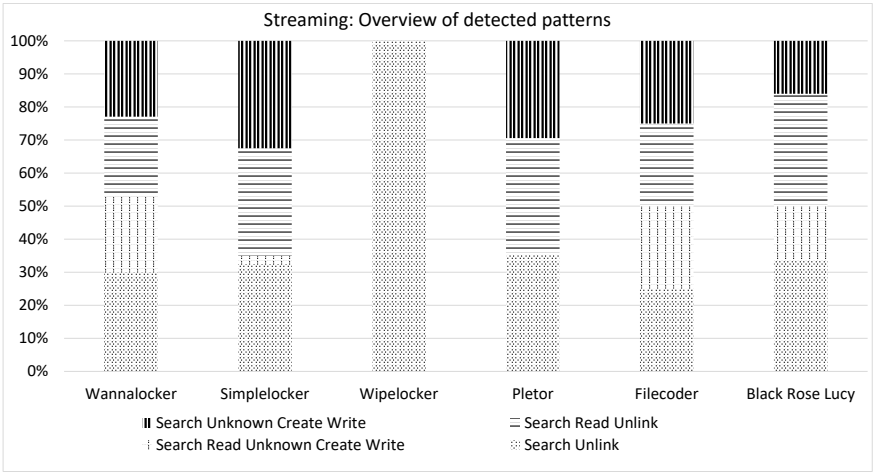

Figure 6.5: Streaming: Sequence of malicious behaviour results

ible indication of shared common behaviour among encryption-type ransomware regardless of the family. The only exception is of WipeLocker, which demonstrates a singular behavioural pattern. WipeLocker is known to only remove user files, without encrypting them. Although there have been different classifications for WipeLocker Chen et al. (2017), we chose to classify this specific family as an

encryption-type ransomware based on the observed system behaviour (unlinking files) rather than the user perceived behaviour, such as ransom notes or displaying a perpetual window, which may result in a different classification. Further, in our evaluation, we were unable to find any match for the *Rename & Unlink File* pattern as this behaviour was likely tied to a specific variant of WannaLocker.

The results shown in this evaluation have validated the feasibility of our discovered malicious behavioural patterns for detection of encryption-type ransomware. Additionally, we have shown the feasibility of our streaming approach for detecting malicious patterns by achieving similar successful results to our offline approach.

6.2.2 Benign Applications Test

We tested both approaches on a dataset consisting of 502 benign applications. Two of the benign applications were cache-cleaning applications, which are discussed in a separate section. In the following subsections, we explain the results of our experiments. The results of the benign applications were previously described in Chapter 5, Section 5.3.3. We further extended this evaluation on our streaming approach by utilising the same dataset. However, we applied incremental changes to refine the patterns. This is further elaborated in the next section.

6.2.2.1 Streaming Method

Our initial streaming approach contained one layer of FSMs where each pattern represented a behaviour, similar to the offline approach. As we evaluated this initial design on our benign dataset, we encountered 2.2% (11 out of 500) false positives and 100% true positives. To help alleviate the false positives, we applied

a second layer of FSM as mentioned in Section 6.1.2.2, which captured the sequence of behaviours.

After re-evaluating with the inclusion of layer 2 FSM, we encountered a much higher false positive rate of 4.2% (21 out of 500) with unchanged true positive rates. The increase in false positive rate was caused by the combination of the suspicious pattern *directory search* and *unlinking user file*, which was present in 17 out of 21 of falsely classified benign applications. This issue occurred because the initial *directory search* pattern matched all folders within the user directory. This included the `Android` folder where application-specific files were stored. The *unlinking user file* pattern also had the same issue where any file within the user directory was considered a match. To alleviate this issue, we restricted the *Directory Search* pattern to exclude the `Android` folder. This alteration significantly reduced the false positive rate to 1% (5 out of 500) whilst retaining the 100% true positive rate.

This method, however, can potentially produce false negatives, as applications may store valuable data for the user within the application-specific folders or users can also store their own files within the folder. To observe this, we tested the new pattern on 6 different encryption-type ransomware (from different families). Each sample was observed for 5 minutes in an emulated environment with trap files stored within the Android directory. The extension of time from 2 minutes to 5 minutes is to extensively evaluate each application and ensure that all behavioural patterns were captured. In this test, 5 out of 6 ransomware encrypted the files within the Android directory except for Wannalocker, which did not encrypt files within the Android folder. These results posed an issue as the exclusion of the `Android` directory limited the scope of our detection process. The Android di-

rectory is often used to store application related files. For example, gaming save files. Some ransomware samples might not consider specific directories to encrypt, but rather the file extensions. Hence, if there is a file extension of interest in the Android directory, then encryption will still occur.

To mitigate this issue without compromising on the detection rate, we observed the differences in behaviour between benign and encryption-type ransomware, specifically the behaviour of directory search. We noticed that with encryption-type ransomware, a directory search occurred for multiple folders within the user directory to ensure a widespread effect. However, for benign applications this search was less frequent, except for specific applications, such as cache-cleaning applications. To evaluate this theory, the *directory search* pattern was altered to detect directory searches that occurred two or more times in separate directories. With this alteration, the false positives rates were reduced to 0.4% (2 out of 500) with 100% true positives. This was a 250% reduction in false positives compared to the methodology of excluding `Android` directory without compromising on the scope, and accuracy of our detection. Hence, we utilised this methodology in our detection system.

Utilising the Altered Directory Search method, two false positives were detected. These two false positives consisted of `search_unlink` sequences. This was likely caused by the applications accessing the same user directory multiple times (i.e., Android directory) and unlinking application related files. As the systems calls were abstracted into tokens, the detection system was unable to identify fine-grain details, such as different user directories being accessed (i.e., if the same user folder was accessed twice, it would be considered a directory search pattern). This is one of the known limitations of our proposed streaming approach.

6.2.2.2 Cache-Cleaning Applications

As previously detailed in Chapter 5, Section 5.2.1.1, specific benign applications, such as cache-cleaning applications could produce behaviours, which can potentially be deemed as malicious if the context is not known (e.g., unlinking junk files within the user directory). Hence, we separately evaluated two cache-cleaning applications to evaluate the efficacy of our approaches. By utilising the offline methodology mentioned in Chapter 5, Section 5.1, one of the cache-cleaning application resulted in a false positive. There were four total malicious patterns matched, and all four of those patterns were linked to *Read User File*. From the examination of the patterns file and system call log file, these four patterns were reading the contents of the user created files (i.e., pre-existing files, not created by the application), which would be deemed as malicious behaviour as it is unusual for most benign application to be reading the contents of user created files.

Table 6.3 contains a summary of our results, which utilised the offline approach. The *Percentage* column shows the percentages of true negatives and false positives detected for all benign samples evaluated. The *Sample Size* column denotes the numerical value of true negatives and false positive samples detected, while Table 6.4 provides an overview of the true negatives and false positives of 502 benign applications for the streaming approach with the 4 aforementioned alterations. Additionally, the evaluation results for cache-cleaning application have also been included.

We can see that the false positive rates of our streaming approach have noticeably improved (using the Altered Directory Search method) compared to the offline approach. This was due to the introduction of a layer 2 FSM, which observed se-

quences of behaviours, thus further distinguished the differences between a benign and malicious application behaviour. Additionally, based on our observations, we made incremental alterations to the patterns based on the behaviours exhibited by benign and malicious applications to identify the best-fit method for our approach. The false positive rates show that detecting ransomware and malware in general through behaviours exhibited in system calls is feasible.

Table 6.3: Summary of all benign applications evaluated using offline approach

Benign Samples	**Percentage**	**Absolute Number**	**Sample Size**
True Negative	98.6%	495	502
False Positive	1.4%	7	

Table 6.4: Summary of benign evaluation with the streaming approach using aforementioned methods

Methodology	**True Negative**	**False Positive**	**Sample Size**
Layer 1 Evaluation	489 (97.8%)	11 (2.2%)	500
Layer 2 Evaluation	479 (95.8%)	21 (4.2%)	500
Restricting User Directory	495 (99%)	5 (1%)	500
Altered Directory Search	**498 (99.6%)**	**2 (0.4%)**	**500**
Incl. Cache-cleaning application	498 (99.2%)	4 (0.8%)	502

6.2.3 Performance Evaluation

A critical aspect of such a detection system is the time it takes to detect malicious activity, which affects its feasibility in a real-world environment. We tested both

Table 6.5: Average detection time for individual patterns in seconds

Pattern Name	**Offline**	**Single Match**	**Sequential Match**
Unlink User File	0.623s ± 0.0081s	0.026s ± 0.0171s	0.335s ± 0.0908s
Unknown File Ext	0.670s ± 0.0076s	0.175s ± 0.3068s	0.406s ± 0.1437s
Read User File	0.738s ± 0.0079s	0.021s ± 0.0112s	0.384s ± 0.0897s
Write to Unknown File Ext	0.661s ± 0.0027s	0.024s ± 0.0120s	0.454s ± 0.0982s

our offline and streaming approaches on this aspect.

To evaluate the pattern matching time, we executed a malicious ransomware variant 10 times on each approach for 120 seconds. For the offline approach, the log file was recorded once. However, the detection component was executed 10 times on the same log file. This was done to ensure consistent results. Table 6.5 shows a summary of our results. *Offline* indicates the offline approach, *Single Match* represents individual behaviours matched (i.e., layer 1 FSM), and *Sequential Match* is the combination of individual behaviours matched in sequential order (i.e., layer 2 FSM) in the streaming approach. To calculate the *Offline* time, we measured the average time taken to match a pattern using the regular expression. For *Single Match* and *Sequential Match*, we measured the average time from the first transition to the last transition of the FSMs (both layer 1 and layer 2 respectively). As seen in the results, the pattern matching times in the streaming approach are significantly lower compared to the offline approach. This was due to the change in the design of the architecture by introducing a tokenised FSM approach, which retained the current state without the intricacies of regular expression matching.

We conducted another evaluation to assess the efficacy of our streaming ap-

proach by measuring the number of system calls that can be processed per second (i.e., throughput). To achieve this, we observed 10 random benign samples for 120 seconds and measured the average CPU time (*user time+ system time*) of all samples. We then acquired the average number of system calls generated from all samples and computed the number of system calls that can be processed by our streaming approach per second (i.e., *Throughput = Number of system calls/ CPU time*). The throughput produced from our streaming approach can be compared to the number of system calls that can be produced by the application over 120 seconds (i.e., *Application run-time throughput = Number of system calls*/120s) to determine the feasibility of our approach. In our experiment, we found that the average number of system calls generated from our applications over 120 seconds was 134020 ± 96078, and the average CPU time for our streaming approach was $17.57\text{s} \pm 12.975\text{s}$. From these two values, the calculated throughput of our streaming approach was 7628 system calls/s. In comparison to the number of system calls produced by the application over 120s, which is 1117 system calls/s, the results indicate that our proposed streaming approach is feasible, as it is capable of processing more system calls than an application can generate.

6.2.4 Discussion

In this section, we discuss some of the experiences we had and the observations we made. As established by now, we were observing the behaviour of encryption-type ransomware on Android operating system, however, the samples of such malware are limited and hard to find especially for Android devices. Additionally, the process of acquiring and validating these samples was time-consuming as each

downloaded sample had to be manually checked against VirusTotal Sood (2017) to ensure that the malware was of an encryption-type ransomware family. Of the 500 samples we collected, 213 exhibited encryption-type ransomware behaviour. The remaining 287 samples were not utilised due to reasons such as,

- The application not executing due to missing manifest files
- The application not executing due to incompatible Android versions
- The applications not exhibiting encryption-type ransomware behaviour
- The application requiring a connection to C&C server

As a result of the dataset, we realised that our models could potentially lead to the issue of an overfitted solution due to the low malicious sample size. However, the samples collected covered the vast majority of encryption-type ransomware samples on Android devices; although limited, we believe this is close to the extent of the current encryption-type ransomware landscape on Android.

Relevant literature indicates that insufficient data can have an adverse effect on the results, leading to lower accuracy or overfitting. Of importance, Stockwell and Peterson (2002) investigated different sample sizes using different machine learning method. From the analysis, they concluded that some methods requires fewer data points to produce a model with acceptable accuracy. However, overall, the most accurate model was achieved using the most data points. Vabalas et al. (2019) observed the effects of different validation methods using a limited sample size. Based on their evaluation results, a regular K-fold cross validation method can produce optimistically biased results on relatively small sample sizes. Whereas, a nested K-fold cross validation was a more effective solution to produce

unbiased results irrespective of the sample size as the training and validation data are separated by two layers. Although the core limitation of the sample size is still existent. Many of the existing datasets generated were based on multiple types of malware. Hence, by generating datasets targeted at a specific malware type, it enables more avenue of exploration for specific malware types in the future.

We have specifically developed our approach for detecting encryption-type ransomware as it is more prevalent and destructive compared to locker-types in recent years. Due to the specificity of our approach, we do not believe it would be feasible to accurately detect locker-type ransomware using the current behavioural implementation without further significant adjustments. Hence, as part of our future work, we aim to explore the adjustments required and broaden our approach to include other types of malware, such as trojans, and spyware or introduce different variants of our dataset to counteract the aforementioned issues and concerns.

It needs to be noted that the intention of this work was the creation of FSMs models and behavioural patterns, which currently require manual observation and human interaction. This often makes the process time-consuming and difficult. For our future work, we intend to further develop our approach by automating the process of identifying behavioural patterns and FSM creation, thus alleviating the requirement of human interaction and enable us to create a fully automated self-protecting system. Additionally, as all experiments were conducted in an emulated environment, the performance evaluation results while indicative of acceptable performance do not truly reflect a real-world implementation. Hence, in future, we intend to implement the streaming approach on a real user device.

An astute reader would also make the observation that the sequence of events in the layer 2 FSM are allowed to occur in any order except for the last detected

behaviour, thus resulting in a partial shuffling of events. This provides flexibility in the detection process. However, a potential limitation of this partial shuffle is the last event in a layer 2 FSM, which always occurs in the same order (e.g., `Search Read Unlink` = `Directory Search` OR `Read User File > Read User File` OR `Directory Search > Unlink User File`). Even though our evaluation for detecting encryption-type ransomware was successful, there is potential for false negatives if a malicious application exhibits a malicious sequence of behaviour, which does not match the last occurring behaviour. In future, we would like to expand this work by utilising a full shuffle approach or a fixed sequence of occurring events and compare the differences in detection rates.

6.3 Chapter Summary

The work proposed in this chapter presents a real time behaviour-based ransomware detection method, which expanded on the methodology proposed in Chapter 5. This is achieved by adopting a 2-layer token-based finite state machine streaming approach. The evaluations conducted in this chapter demonstrate that our ransomware detection system can run on an Android operating system with acceptable overhead.

Chapter 7

Automatic Detection of Encryption-type Ransomware using Supervisor Reduction[1]

Many systems leverage machine learning or artificial intelligence techniques to automatically detect ransomware and malware. Such approaches can offer robust classification of malware. However, some of the devised solutions often result in hefty resource consumption. This presents a challenging obstacle for developing automated and robust solutions for malware detection on resource constrained devices. One of the research objectives of this thesis aims to address this issue by adopting an alternative approach, without the reliance of machine learning and artificial intelligence, to detect encryption-type ransomware on Android devices.

This chapter proposes a Finite State Machine (FSM) based approach to recog-

[1]This chapter is a co-authored journal paper, which is under review (Chew et al., 2022) with slight modifications in Section 7.2.2 and Section 7.2.8.

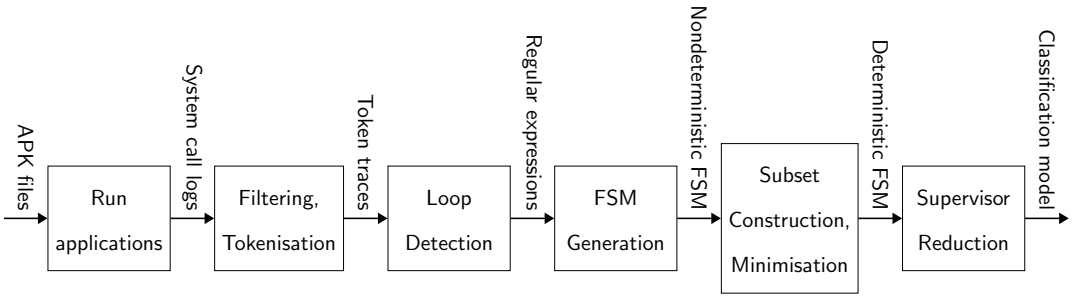

Figure 7.1: Steps of classification model generation.

nise encryption-type ransomware based on their behaviour. Malicious and benign Android applications are executed to capture the system calls they generate, which are then filtered and tokenised and converted to finite-state machines. The finite-state machines are simplified using *supervisor reduction*, which generalises the behavioural patterns and produces compact classification models. The classification models can be implemented in a lightweight monitoring system to detect malicious behaviour of running applications quickly. An extensive set of cross validation experiments is carried out to demonstrate the viability of the approach, which show that ransomware can be classified accurately with an F1 score of up to 93.8%.

7.1 Methodology

Figure 7.1 gives an overview of the proposed method to detect encryption-type ransomware at a system call level. In the first step, system call logs are acquired by *running applications* from a dataset of encryption-type ransomware. In the second step, the system call logs are *filtered and tokenised*, replacing the system calls by a small set of tokens suitable for an FSM model. Next, *loop detection* compresses the token traces to regular expressions by identifying repeated behaviour.

Then the *FSM generation* step combines the regular expressions for all traces in a nondeterministic FSM, which is then converted to a minimal deterministic FSM using *subset construction and minimisation*. Finally, *supervisor reduction* minimises the FSM further and produces a classification model, which can then be used to monitor running applications and check for malicious behaviour.

The following subsections explain these steps in further detail. Subsection 7.1.1 describes filtering and tokenisation, Subsection 7.1.2 describes loop detection, and Subsection 7.1.3 describes the process of converting regular expressions to FSMs and their minimisation, which covers the three remaining steps. Afterwards, Subsection 7.1.4 examines the computational complexity of the whole procedure.

7.1.1 System Call Filtering and Tokenisation

Applications typically generate a large number of system calls, not all of which are helpful to classify the behaviour as malicious or benign. An example is the system function `clock_gettime()`, which is frequently used to read the system clock time by both malicious and benign applications. Such irrelevant system calls increase the volume of the system call data significantly and make it difficult to identify malicious behaviour. To combat this issue, a white-listing approach (Isohara et al., 2011a; Chew et al., 2020) is used to filter out some system calls.

Furthermore, after filtering system calls based on the white-list, the raw data generated by `strace` still contains a lot of detail about each system call including timestamps, parameters, and return values, which needs to be abstracted from to build a useful FSM classification model. Therefore, the system calls that survive filtering are replaced by *tokens* selected from a small set, again similar to the

Table 7.1: Six possibilities to tokenise system calls.

	Tokens					
Operation	Set I	Set II	Set III	Set IV	Set V	Set VI
Open directory	O	OD	OD	OD	OD	OD
Open file for reading	O	ORF	ORF	ORF	OXF	ORF
Open file for writing (create)	O	OWF	OCWF	OCWF	OCWF	OCWF
Open file for writing (append)	O	OWF	—	OAWF	OXF	OXWF
Open file for writing (other)	O	OWF	—	—	OXF	OXWF
Rename file	RN	RN	—	RN	RN	RN
Unlink/delete file	U	U	U	U	U	U

approaches of Chew et al. (2020) and Isohara et al. (2011a).

Table 7.1 shows six different ways considered in this work to convert system calls to tokens. The defining behaviour of encryption-type ransomware is through the encryption process (Chew et al., 2020; Kok et al., 2019; Lemmou et al., 2021), and therefore the selected tokens are related to the specific system calls used when the ransomware searches for and encrypts files. The most relevant system call is `openat()`, which is needed to open directories in order to search them, to open files to read their contents, and to create files to write encrypted data. Additionally, the `renameat()` system call is represented by a token, because certain samples of Wannalocker exhibit behaviour where the encrypted file is renamed (Chew et al., 2020). Lastly, encryption-type ransomware often removes the original file after the encryption process or as part of the extortion process, and therefore the `unlinkat()` system call is represented by another token.

The encryption of files also generates a large number of calls to `read()` and

`write()`, but these were found to be less relevant, as the opening of a file is followed by these calls in benign applications as well. Instead, it is determined from the arguments to `openat()` how a file is opened, in the hope that the pattern of different types of `openat()` system calls provides more relevant information in a more concise form. Firstly, it is of interest whether a directory or file is opened, which is easily determined by the presence or absence of the `O_DIRECTORY` flag. When a file is opened, it is distinguished whether it is opened for reading (with the flag `O_RDONLY`) or for writing (with the flag `O_WRONLY`). In the case of writing a file, it is furthermore distinguished whether the system call allows for creation of a new file (with the flag `O_CREAT`), whether it tries to append to an existing file (with the flag `O_APPEND`), or whether neither of these flags is used. The captured system call logs also contain a large number of system calls to open files for reading and writing (with the flag `O_RDWR`), which are filtered out. The read/write access is common in malicious and benign applications, and including it was found to produce long token traces that are difficult to process without producing useful results.

The six token sets in Table 7.1 are defined to examine the effects of different levels of abstraction in the token traces. Set I is the coarsest set, which represents all `openat()` system calls with the same token `O` and makes no distinction based on the flags, while Set II distinguishes between opening directories and opening files for reading or writing, without separating the different modes of writing. Some token sets filter out certain system calls, which is shown by a dash in the table to indicate that no token is generated. Most of the evaluation presented in the following is based on Set IV, which is designed based on the understanding of ransomware behaviour in the hope of preserving a reasonable level of detail

while keeping the token traces small. The other tokenisations are evaluated in Section 7.2.4 below.

To measure the effectiveness of the filtering process, the average length of all unfiltered system call traces was calculated and compared to the average length of the filtered token traces. The average length of all unfiltered traces collected was 167489 system calls. After filtering and tokenisation using Set IV, the average trace length was 552 tokens, resulting in a 99.7% reduction. This is a significant decrease in the number of tokens, which greatly facilitates the following processing steps.

7.1.2 Loop Detection

System call traces can be long even after filtering, and several of the traces in the dataset contain significant subsequences of repeated tokens, which are likely produced by loops in the application code. To improve the classification models, a simple loop detection process is used to detect patterns of repeated tokens and compress them into loops. The method proposed here differs from related work (Gharib and Ghorbani, 2017; Bhandari et al., 2018) in that repeated subsequences are not removed, but replaced by loops. This reduces the size of the traces and helps to produce smaller state machines, while at the same time trying to infer a more general behavioural model from the trace.

A sliding window technique is used to combine sequences of repeated tokens into loops. The process begins by taking a token trace and a threshold $k \geq 1$ as input. The threshold signifies the number of repeated tokens required before a token sequence is replaced by a loop. For example, with threshold 2, a sequence

OO is replaced by O^{2+}, which indicates two or more occurrences of the token O. In general $E^{k+} = E^k E^*$ for $k \geq 1$ and regular expressions E, and at threshold k any sequence α^j with $j \geq k$ and $\alpha \in \Sigma$ is replaced by α^{k+}.

Example 7.1. *Assuming* $\Sigma = \{\mathsf{O}, \mathsf{R}\}$*, the token trace* OORRR *is replaced by* $\mathsf{O}^{2+}\mathsf{R}^{2+}$ *at threshold 2 and by* OOR^{3+} *at threshold 3.*

The loop detection process is repeated to detect longer repeated subsequences and nested loops. For the second step, each subterm of the form E^{k+} is considered as a single token, and the maximum size of the sliding window is increased by 1, thus searching for subsequences of length 2 that are repeated. This process is repeated for a set number of iterations, where the i-th iteration searches for repeated subsequences of any length up to i.

Example 7.2. *Given a threshold value of 2, the token trace* OOOROOROOOOR *is replaced by* $\mathsf{O}^{2+}\mathsf{RO}^{2+}\mathsf{RO}^{2+}\mathsf{R}$ *in the first iteration, and this is replaced by* $(\mathsf{O}^{2+}\mathsf{R})^{2+}$ *in the second iteration.*

Both the threshold and the number of iterations are adjustable parameters of the loop detection process. Section 7.2.3 contains the results of an experiment to examine how changing these parameters affects the accuracy of ransomware detection.

7.1.3 Classification Model Generation

Having captured and filtered samples of malicious and benign traces, the next step is to combine these traces in an FSM. Given a set Σ of tokens and two sets $B, M \subseteq \Sigma^*$ of benign and malicious token traces, an FSM G is constructed with

positive and negative languages

$$\mathcal{L}^+(G) = M\Sigma^* \; ; \tag{7.1}$$

$$\mathcal{L}^-(G) = \mathrm{Pre}(B) \setminus M\Sigma^* \; . \tag{7.2}$$

Every malicious trace and every possible continuation of a malicious trace is classified as positive (7.1). This represents the idea that, once malicious behaviour in M has occurred, damage has been done and cannot be reverted by further action. Furthermore, all prefixes of benign traces are classified as negative, so traces that can be continued to an observed benign trace are not considered as malicious (7.2). To ensure the requirement from Section 2.5.2 that the positive and negative languages are disjoint, traces that occur both in the malicious and benign sets are classified only as positive by removing the continuations of malicious traces from the negative language (7.2). That is, ambiguous traces encountered as both malicious and benign lead to false positives rather than false negatives. This situation did not arise during the experiments carried out for this work.

To construct the FSM G, first the regular expressions representing the reduced token traces according to Section 7.1.2 are converted to nondeterministic FSMs using a standard algorithm (Hopcroft et al., 2001). These FSMs are then combined in a single nondeterministic FSM with several initial states that includes the positive and negative languages (7.1) and (7.2). This nondeterministic FSM is converted to a minimal deterministic FSM with the same positive and negative languages using subset construction and Hopcroft's minimisation algorithm (Hopcroft et al., 2001). If the deterministic FSM from subset construction contains any states that are marked as both positive and negative, then the negative marking is removed from these states to ensure (7.2).

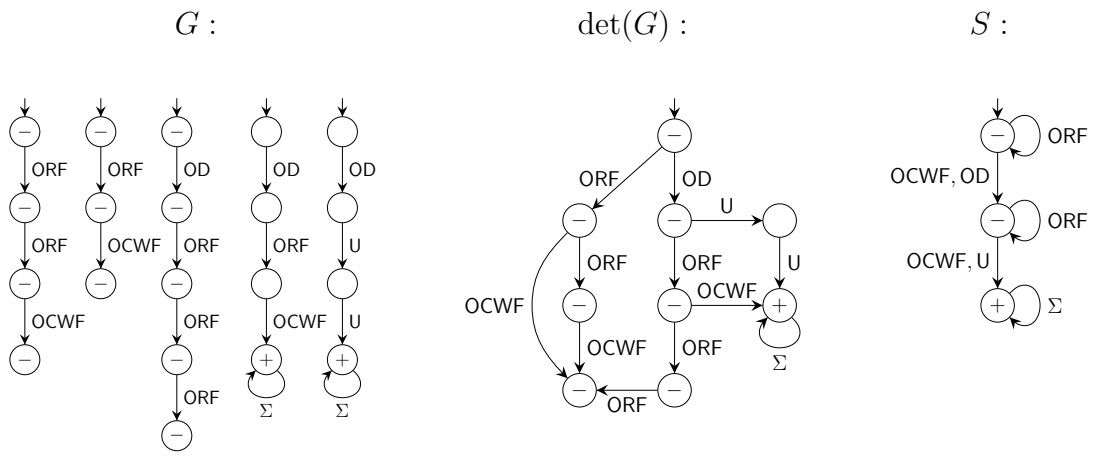

Figure 7.2: Constructing a classification model. The nondeterministic FSM G is constructed from the malicious and benign traces, converted to a deterministic FSM $\det(G)$, and minimised by supervisor reduction to obtain the classification model S.

Finally, the minimal deterministic FSM is passed to supervisor reduction (Section 2.5.3) to obtain the *classification model*. The classification model is a deterministic FSM that can then be used to classify new observed behaviour as malicious or benign as follows. Starting from the initial state, system calls are observed and converted to tokens, and each token is used to update the classification model's state. If a positive state is reached at some point, this means that malicious behaviour has been detected. It is also possible that a token is generated for which no transition is defined from the current state of the classification model—then no further tokens are processed and the behaviour is considered as benign.

Example 7.3. *Consider the token set* $\Sigma = \{\mathsf{OD}, \mathsf{OCWF}, \mathsf{ORF}, \mathsf{U}\}$ *and the benign*

and malicious languages

$$B = \{\mathsf{ORF\,ORF\,OCWF},\ \mathsf{ORF\,OCWF},\ \mathsf{OD\,ORF\,ORF\,ORF}\}\ ; \tag{7.3}$$

$$M = \{\mathsf{OD\,ORF\,OCWF},\ \mathsf{OD\,U\,U}\}\ . \tag{7.4}$$

Figure 7.2 shows the nondeterministic FSM G constructed from these traces (without loop detection). The three benign traces appear on the left with all their states negative. Thus, all prefixes of benign traces lead to negative states. The two malicious traces on the right have positive end states, and the selfloops labelled Σ indicate transitions with all events in Σ to represent that continuations of malicious traces remain malicious. Figure 7.2 also shows the minimal deterministic FSM $\det(G)$ *equivalent to G and a classification model S resulting from supervisor reduction. Here, comma-separated tokens on a transition indicate parallel transitions with each listed token.*

It is clear that the benign trace ORF ORF OCWF *takes S to a negative state, indicating classification as benign, while the malicious trace* OD ORF OCWF *takes S to its positive state and is classified as malicious. The trace* OD U*, which takes* $\det(G)$ *to a "don't care" state, takes S to its positive state and is classified as malicious. The trace* U U *is not accepted by S from the first step onward and is considered as benign.*

As shown in the example, supervisor reduction does not only reduce the number of states to produce a more manageable classification model, it also attempts to predict and generate a best-fit model based on the "don't care" states. The hope is that this way of minimising the number of states also leads to improved classification accuracy.

FSMs with positive and negative states as shown in Figure 7.2 cannot be submitted directly to supervisor reduction in a supervisory control tool such as Waters/Supremica. Therefore, the FSM model is modified to convert the classification problem to a supervisory control problem as follows. The nondeterministic FSM G is augmented with two states b and m, and with an event μ to represent the detection of malicious behaviour. For each negative state x, a transition $x \xrightarrow{\mu} \mathsf{b}$ is created, and for each positive state y, a transition $y \xrightarrow{\mu} \mathsf{m}$ is created, and all states except b are declared to be *accepting* states. This ensures that, if a *maximally permissive nonblocking supervisor* (Ramadge and Wonham, 1989) is synthesised from the resulting deterministic FSM, the event μ is enabled in positive states and disabled in negative states that are not also positive. That is, the event μ is enabled to signal the detection of malicious behaviour.

7.1.4 Computational Complexity

Considering the steps of classification model generation in Figure 7.1, the steps with the highest computational complexity are *Subset Construction and Minimisation* and *Supervisor Reduction*.

The worst-case time complexity of subset construction is exponential in the number of states of the nondeterministic FSM being converted to a deterministic FSM. Fortunately, this worst-case does not apply to the FSMs generated from the token traces by the proposed method. Without loop detection, the maximum possible number of transitions of the deterministic FSM is equal to the total number of tokens in all malicious and benign traces combined, and the maximum number of states is one more than the number of transitions. This means linear time

complexity in the size of the input.

The bound on the number of states increases with the number of loop detection iterations to a limit of $O(N^k)$ where N is the length of the longest trace and k is the number of traces, which is exponential in the number of traces. However, this limit can only be reached with a high number of loop detection iterations. Exponential blow-up at this stage was not observed in our experiments with up to four loop detection iterations. The exponential worst-case can be avoided by monitoring the number of states generated during subset construction and restarting with a lower number of loop detection iterations when the number of states exceeds a threshold. Then the time complexity remains linear in the size of the input, the threshold, and the number of loop detection iterations.

The next time consuming step is *Supervisor Reduction*, which has a worst-case time complexity of $O(|\Sigma||Q|^4)$ as explained in Section 7.1.4. The worst-case for the number of states is $|Q| = N^k$, which means exponential time complexity in the number k of traces. If the number of states can be kept linear in the size of the input, then the complexity of the supervisor reduction step is polynomial in the size of the input, $O(n^4)$. As supervisor reduction is the step with the highest complexity in Figure 7.1, this becomes the theoretical worst-case time complexity for the entire classification model generation process.

It is important to note that classification model generation is only performed a single time as an offline computation step. Once a classification model FSM has been generated, its state transition function can be implemented as a hash table, which can be used to process system calls very quickly. The worst-case time complexity for processing one system call while monitoring a running application is constant, $O(1)$.

7.2 Evaluation

This section evaluates the performance of the classification models obtained using the method described in Section 4 in terms of their size and accuracy. First, Subsection 7.2.1 describes the evaluation methodology, and the following Subsections 7.2.2–7.2.4 present experiments to determine how different parameters of the classification model generation process affect the results. Afterwards, Subsection 7.2.5 presents an experiment that attempts to detect malware from an unknown family, and Subsection 7.2.6 compares the proposed approach to related work.

7.2.1 Cross Validation Approach

K-fold *cross validation* is a statistical method that is commonly used to evaluate machine learning applications (Marsland, 2011). In this work, a ten-fold cross validation approach is used to measure the accuracy of classification. The traces obtained from 502 benign and 205 malicious applications are randomly split into ten sets. Nine of these sets form the *training data* and are used to generate the classification model following the steps outlined in Section 4. The tenth set forms the *testing data*: its traces are classified as malicious or benign using the classification model constructed from the training data, and the results are compared to the actual status of the application that generated the trace to determine whether the classification is correct or not. This process is repeated ten times by swapping the roles of the traces in the ten sets, so that each trace is used nine times as training data and once as testing data.

After classifying the testing data for all ten rounds of cross validation, the results are accumulated to determine the numbers of true positives, false positives,

true negatives, and false negatives. A *true positive* occurs if the classification model correctly identifies a trace from a malicious application as malicious. Otherwise, if a malicious trace is incorrectly classified as benign, it is called a *false negative*. Similarly, a benign trace is called a *true negative* if it is correctly classified as benign and a *false positive* if it is incorrectly classified as malicious. After the numbers of true positives etc. are determined, it is possible to calculate their *rates*:

$$\mathrm{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \quad \mathrm{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{7.5}$$

$$\mathrm{TNR} = \frac{\text{True Negatives}}{\text{False Positives} + \text{True Negatives}} \quad \mathrm{FNR} = \frac{\text{False Negatives}}{\text{True Positives} + \text{False Negatives}} \tag{7.6}$$

Here, "True Positives" is the absolute number of traces identified as true positives, etc. The *true positive rate* TPR is the ratio of malicious traces that are correctly classified as malicious, while the *false negative rate* FNR is ratio of malicious traces that are incorrectly classified as benign. Similarly, the *true negative rate* TNR and *false positive rate* FPR measure the ratio of benign traces that are classified as benign or malicious, respectively. It is clear that $\mathrm{TPR} + \mathrm{FNR} = \mathrm{TNR} + \mathrm{FPR} = 1$. Additionally, the *F1 score* is used as a combined measure:

$$\mathrm{F1} = \frac{2 \times \text{True Positives}}{2 \times \text{True Positives} + \text{False Positives} + \text{False Negatives}} \tag{7.7}$$

The F1 score combines the rates of false positives and false negatives into a single number between 0 and 1 such that a value of 1 indicates perfect accuracy. It is often used to measure the accuracy of classification models, although it can be criticised for its failure to take the number of true negatives into account.

7.2.2 Evaluation of Supervisor Reduction Options

In the first experiment, it is examined how different supervisor reduction settings affect the generated classification model. As mentioned in Section 2.5.3, the supervisor reduction algorithm (Su and Wonham, 2004) is sensitive to the order in which it attempts to merge state pairs. Therefore, the tool Waters/Supremica (Åkesson et al., 2006), which was used for all experiments, supports several options to control this algorithm.

Firstly, the states of the FSM being reduced can be ordered in different ways. Of particular interest is the *depth-first search* (DFS) state ordering, which arranges the states in the order in which they are visited by depth-first search traversal of the FSM. The initial state is the first state, and states that appear on paths without branching are kept together. Also considered is the *reversed breadth-first search* (RBFS) state ordering, which arranges the states in the reverse of the order in which they are visited by breadth-first search traversal. The states with the longest distance from the initial state appear first in this ordering, which are the end states of the longest trace or traces in the training data, whether these traces are malicious or benign. Four other state ordering options in Waters/Supremica were found to produce poor results throughout initial explorations, and are not considered in this work.

Having ordered the states of the FSM being reduced, there are different possible orders in which to consider state pairs for merging. The simplest strategy is the *Lexicographic* pair ordering, where the first state in the state ordering is first paired with the remaining states in the state ordering. After that, the second state is paired with the states that appear after it in the state ordering, etc. Alterna-

tively, the *Diagonal* pair ordering strategy starts by pairing the first and second states, then considers pairings of the third state with the states before it in the state ordering, before moving on to the fourth state, etc. There are two variants, *Diagonal1* and *Diagonal2* where the states each state is paired with are considered in ascending or descending order. The main difference is that lexicographic pair ordering starts pairing the first state with all others, while diagonal pair orderings give preference to pairs of states that are both close to the start.

Additionally, Waters/Supremica supports the pre-processing step of *projection*, which attempts to remove unnecessary tokens from the FSM before passing it to the supervisor reduction algorithm. Consistently with previous results (Malik, 2020), it was found sufficient to use a greedy search strategy to identify the events to be removed in all experiments. Therefore, the third option considered in the following experiments is whether projection is disabled or enabled with the *Greedy* search method.

For the first experiment, the system call logs in the training data are tokenised based on Set IV in Table 7.1 and subjected to one iteration of loop detection with a threshold value of eight, and the resulting regular expressions are used to produce a nondeterministic and then a minimal deterministic FSM. Then classification models are generated by supervisor reduction with all combinations of the options mentioned above, and used to classify the traces in the testing data.

Table 7.2 shows the results of this evaluation. For each combination of options, the table shows the true positive and associated rates according to (7.5) and (7.6) as well as the F1 score according to (7.7) from ten-fold cross validation. It also shows the number of states of the generated classification model FSMs and the time taken by the supervisor reduction algorithm to compute them, averaged over

Table 7.2: Classification models with different supervisor reduction options.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
DFS	Lexicographic	Off	0.444	0.556	0.984	0.016	0.599	2078	72.6 s
DFS	Diagonal 1	Off	0.585	0.415	0.954	0.046	0.690	742	1493.0 s
DFS	Diagonal 2	Off	0.620	0.380	0.930	0.070	0.692	581	528.2 s
DFS	Lexicographic	On	0.649	0.351	0.972	0.028	0.756	1281	32.7 s
DFS	Diagonal 1	On	0.673	0.327	0.902	0.098	0.704	909	175.0 s
DFS	Diagonal 2	On	0.702	0.298	0.896	0.104	0.718	1192	100.2 s
RBFS	Lexicographic	Off	0.951	0.049	0.948	0.052	0.915	91	502.5 s
RBFS	Diagonal 1	Off	0.951	0.049	0.946	0.054	0.913	91	535.0 s
RBFS	Diagonal 2	Off	0.312	0.688	0.998	0.002	0.474	13293	665.3 s
RBFS	Lexicographic	On	0.917	0.083	0.938	0.062	0.887	107	52.8 s
RBFS	Diagonal 1	On	0.922	0.078	0.942	0.058	0.894	114	54.8 s
RBFS	Diagonal 2	On	0.620	0.380	0.994	0.006	0.758	7665	84.9 s

the ten rounds of cross validation. The experiments were carried out on a PC running Ubuntu 18.04.6 LTS with a 3.8 GHz Intel Core i5-7600K CPU and 16 GiB of RAM.

The most effective classification models were produced by the RBFS state ordering and the Lexicographic or Diagonal1 pair orderings. With projection disabled, these models achieve false negative rates below 5% and false positive rates below 6%, which results in F1 scores above 91%. While a few other models have lower false positive rates, those are accompanied by high false negative rates of 35% or above, suggesting that those models have not changed much compared to the unreduced deterministic FSM and may be overfitted to the data. The models with the highest accuracy also have the lowest number of states, suggesting that

minimisation of the number of states leads to good classification accuracy. The combinations of RBFS with Lexicographic and Diagonal1 both start by pairing a state furthest removed from the initial state with the states preceding it, which seems to be most effective for the FSMs resulting from the construction in this work. Because of the strong performance of these strategies compared to all other combinations of options, which was also observed on several other occasions, the experiments presented in the following sections focus on these strategies.

As explained in Section 2.5.3, projection searches for events that are irrelevant for the classification task and removes them from the FSM prior to supervisor reduction. Table 7.2 shows that projection always changes the generated classification model. By closer inspection of the classification models, it is found that the tokens ORF and OAWF of Set IV are removed in all ten rounds of cross validation, and RN is removed in nine of the ten rounds. The token ORF indicates opening a file for reading and regularly appears in both benign and malicious traces. The token OAWF, which indicates opening a file for appending, is used less frequently, and RN, which indicates renaming of a file, is only observed in some Wannalocker traces. The removal of these tokens by projection suggests that the corresponding system calls are not needed to distinguish malicious and benign traces, and ransomware can be detected by only considering the pattern of directory accesses, files opened for writing, and file deletions. This makes sense given that the malicious activity of the applications considered involves the creation of encrypted files and the deletion of user files, usually in combination with directory searches.

While the use of projection often reduces the runtime of supervisor reduction significantly, it reduces the number of states only once and never leads to better accuracy in this experiment. Particularly with events that are used only occasion-

ally, although accurate classification may be possible without them, it may require a more complicated classification model. In this experiment, supervisor reduction seems to be able to find better classification models that use all events, although it takes longer to do so because the supervisor reduction algorithm has to process more states.

The supervisor reduction times in Table 7.2 are all below 25 minutes, with the better performing supervisor reduction options finishing in less than ten minutes. Notwithstanding the theoretical complexity analysis, the by far most time-consuming step when generating a classification model is running the applications. It takes 36 hours to run the 707 applications for three minutes each, although the process can be parallelised. System call logs can also be saved and reused, so when a dataset is extended with new applications, only the new applications need to be run. Still, the supervisor reduction times are comparatively small, even if several classification models are constructed to choose the smallest one. The times taken for the other steps in Figure 7.1 are negligible in comparison.

Additionally, the storage requirements to implement the classification models on a real device are acceptable. Such an implementation requires 16 bit for a source state, 3 bits to represent the number of events (i.e., tokens), and an additional 16 bits for the destination state. By utilising a half-full hash-table implementation, the sum is multiplied by 2, resulting in a requirement of 16 bytes per transition. For example, multiplying the largest average number of transition shown in Table 7.2, the resulting FSM would require 212.69KB. However, it is possible to achieve a more compact model by calculating the exact sizes of the source state, event, and destination state. The calculated value is significant smaller when compared to the recorded average of 23.46MB for Android smartphone applications

based on a study conducted by Chen et al. (2021) where they explored the differences and similarities for 223 application pairs between Android smartphones and smartwatch. The space complexity can be presented as $O(n)$ without the usage of loop detection and $O(n^2)$ with loop detection where n is the total number of system calls.

Overall, the runtimes suggest that it is feasible to generate a classification model offline for later use on a smartphone. Once a classification model has been generated, it can be used easily to monitor a running application with minimal processor or memory overheads on a real device. This is an important benefit as real-time and early detection approaches are critical for protecting devices of users.

7.2.3 Evaluation of Loop Detection

A series of experiments was carried out to evaluate the efficacy of the loop detection process described in Section 7.1.2. Loop detection has two parameters, the *number of iterations* which determines the length of repeated sequences and the potential nesting level of detected loops, and the *threshold* which determines the minimum number of repetitions before a sequence of tokens is replaced by a loop. In the experiment, the ten-fold cross validation process described above is repeated after performing 1–4 iterations of loop detection with threshold values in the range 2–12. All system call logs are tokenised using Set IV, and the supervisor reduction options are limited to the combinations found most effective in Section 7.2.2, namely reversed breadth-first search (RBFS) state ordering and Lexicographic or Diagonal1 pair ordering.

Figure 7.3 gives an overview of the accuracy and size of the classification models

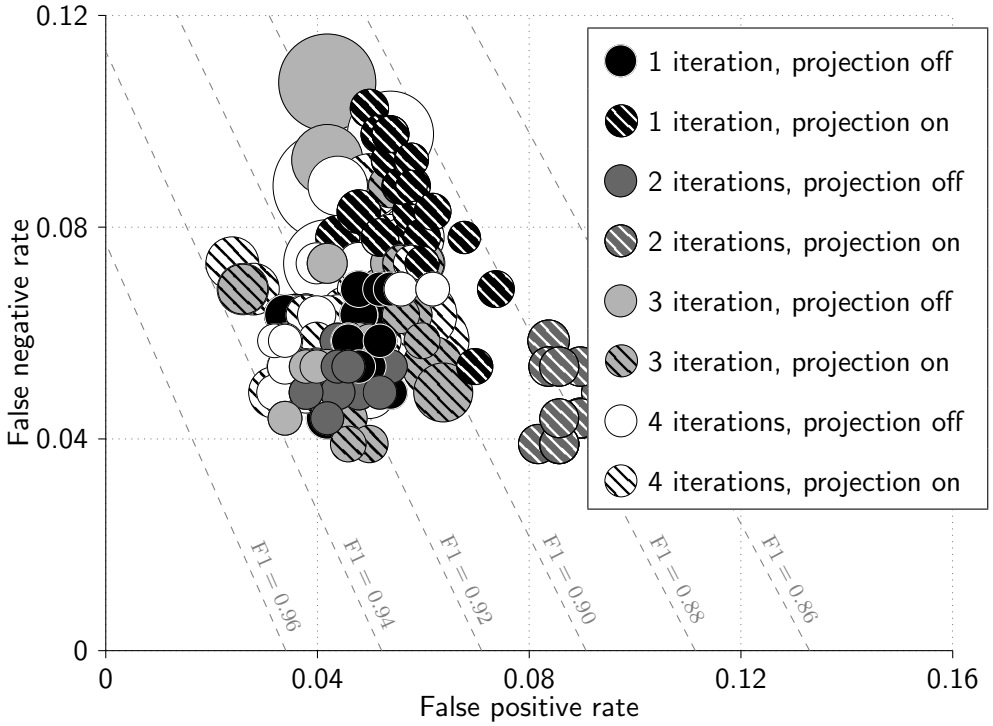

Figure 7.3: Classification model accuracy and size depending on loop detection parameters and projection.

produced in this experiment. Each circle indicates the results for one particular combination of loop detection and supervisor reduction options. The position of the circle indicates the false positive and false negative rates measured by ten-fold cross validation, and the size of the circle represents the average number of states of the classification models in the ten rounds of cross validation. The results can also be compared to classification models computed with loop detection disabled, which are shown in Table 7.3.

All the classification models in this experiment have F1 scores between 86.5% and 93.8%. Figure 7.3 shows that the majority of the models computed with loop

Table 7.3: Classification models obtained without loop detection.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
RBFS	Lexicographic	Off	0.912	0.088	0.946	0.054	0.893	176	2407.4 s
RBFS	Diagonal1	Off	0.912	0.088	0.946	0.054	0.893	176	2461.3 s
RBFS	Lexicographic	On	0.922	0.078	0.944	0.056	0.896	77	74.1 s
RBFS	Diagonal1	On	0.917	0.083	0.940	0.060	0.889	75	78.3 s

detection are more accurate than those computed without loop detection, which have F1 scores between 88.9% and 89.6%. This suggests that loop detection can help to find more accurate classification models.

Figure 7.3 suggests that more accurate classification models have fewer states, but with the poorly performing supervisor reduction options removed, this trend is less pronounced than previously observed in Table 7.2. The numbers of states obtained with the different loop detection options range from 78 to 930. The smallest model with 78 states, obtained with one loop detection iteration and a threshold of five, has an F1 score of 88.8% and a false negative rate of 8.8%. The most accurate model with an F1 score of 93.8% and a false negative rate of 4.4% has 91 states; it is obtained with three or four iterations and a threshold of eight.

Furthermore, there is a difference in accuracy between models obtained with projection enabled and disabled, which was already observed in Table 7.2. The models computed with projection and with one and particularly two loop detection iterations are among the least accurate. On the other hand, models computed with two loop detection iterations and without projection form a group of fairly accurate models. A higher number of iterations leads to a more irregular spread, including

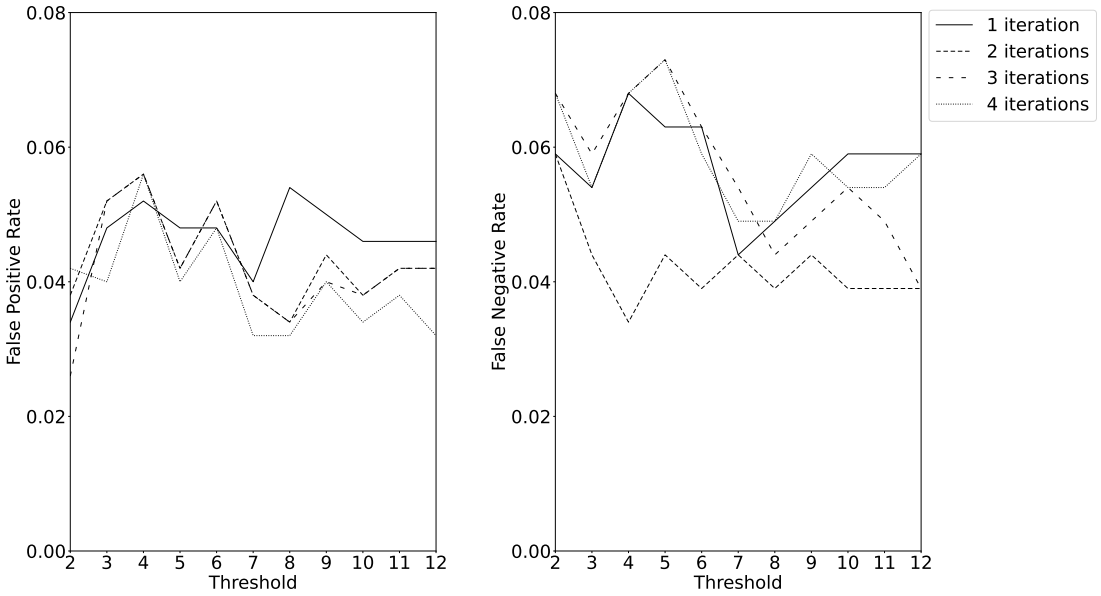

Figure 7.4: Lowest false positive and false negative rates depending on loop detection parameters.

the most accurate models as well as several large and poorly performing models both with projection enabled or disabled.

For a more detailed analysis of the effect of the number of loop detection iterations and the threshold, the graphs in Figure 7.4 show the lowest false positive and false negative rates among the four models obtained for each combination of these two parameters. These diagrams suggest that two iterations lead to the lowest false negative rates, while four iterations produce slightly lower false positive rates. Also, the false positive rates are relatively high for threshold values in the range 3–6, with a possible local minimum of the false positive rate at a threshold of 2, and another possible local minimum of both false positive and false negative rates at a threshold of 7 or 8.

These minima may be explained by the way how loop detection separates re-

peated token sequences that are shorter than the threshold from longer ones. A sequence of repeated tokens shorter than the threshold is left unchanged, while a longer sequence is modified with a loop at the end. As a result, the states corresponding to these two types of sequences are not merged when constructing the minimal deterministic FSM, leaving it to supervisor reduction to decide whether such states should be treated alike or differently. More uniform classification can be ensured when similar subsequences are treated alike, either converting them all to loops or not. This seems to be achieved best by thresholds of 2, 7, or 8, which produce the most accurate models in this experiment.

Figure 7.5 shows the supervisor reduction times measured in this experiment. The graphs show average runtimes over four combinations of supervisor reduction options and over ten rounds of cross validation, depending on the two loop detection parameters. The diagram shows that the runtime increases steadily with increasing threshold towards the average of 1255.3 s without loop detection, when using one loop detection iteration. Yet, this increase is not observed with more iterations. The runtime of supervisor reduction is mainly determined by the size of the deterministic FSMs, whose number of states appears to remain limited when two or more loop detection iterations are used. A secondary factor affecting the runtime is the structure of the FSMs, which become more complex as each iteration produces longer and more deeply nested loops. In this experiment, two iterations seem to result in the fastest runtime.

Loop detection broadens the sets of positive and negative traces in the training data. For each positive trace, a group of similar traces is identified that contain the same subsequences with more repetitions, and these similar traces are removed from the set of "don't care" traces and added to the set of positive traces; the set

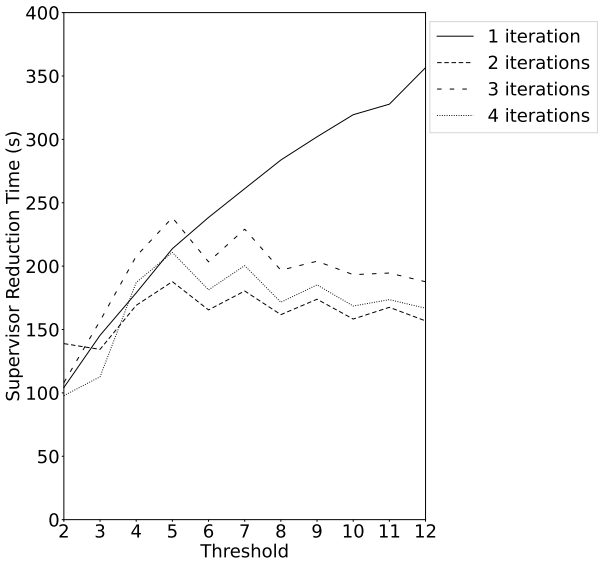

Figure 7.5: Average supervisor reduction times depending on loop detection parameters.

of negative traces is broadened likewise. This results in a more difficult supervisor reduction problem, which has the potential to provide a more accurate classification model provided that the broadening is accurate. This explains why most of the models computed with loop detection are more accurate than those without. Additionally, loop detection tends to produce smaller deterministic FSMs, which usually means faster supervisor reduction times.

7.2.4 Evaluation of Different Tokenisations

This section describes an experiment to determine how classification models are affected by different ways of filtering and tokenising system calls. This experiment again uses one iteration of loop detection with a threshold of eight, and the supervisor reduction options identified as most effective in Section 7.2.2, i.e., RBFS

Table 7.4: ESCAPADE tokenisation.

	Tokens	
Operation	User directory	Other directory
Open directory	$\mathsf{OD}_{\mathsf{udir}}$	$\mathsf{OD}_{\mathsf{other}}$
Open file for writing (create)	$\mathsf{OCWF}_{\mathsf{udir}}$	$\mathsf{OF}_{\mathsf{other}}$
Open file for reading or writing (excluding create)	$\mathsf{OXF}_{\mathsf{udir}}$	$\mathsf{OF}_{\mathsf{other}}$
Rename file	$\mathsf{RN}_{\mathsf{udir}}$	—
Unlink/delete file	$\mathsf{U}_{\mathsf{udir}}$	$\mathsf{U}_{\mathsf{other}}$

pair ordering and Lexicographic or Diagonal1 state ordering.

In addition to the six ways of tokenisation introduced in Table 7.1 in Section 7.1.1, a seventh tokenisation is considered that was originally proposed for ESCAPADE (Chew et al., 2020). This tokenisation, shown in Table 7.4, distinguishes system calls based on whether they access the *user directory* or another directory. The user directory on Android is a space for the user to create and store personal files such as photos, downloads, or multi-media, which are a primary target for encryption-type ransomware. The other directories predominantly contain system files, applications, and their configuration files. As shown in Table 7.4, the ESCAPADE tokenisation uses different granularities of tokens representing system calls in the user directory or in other directories, which was found effective for use with the hand-coded classification models of Chew et al. (2020).

Figure 7.6 shows the false positive and false negatives rates in relation to the average state numbers of the classification models, which is represented by the size of the circles in the same format as in Figure 7.3. Some tokenisations appear to have fewer than four classification models, which occurs when the same false

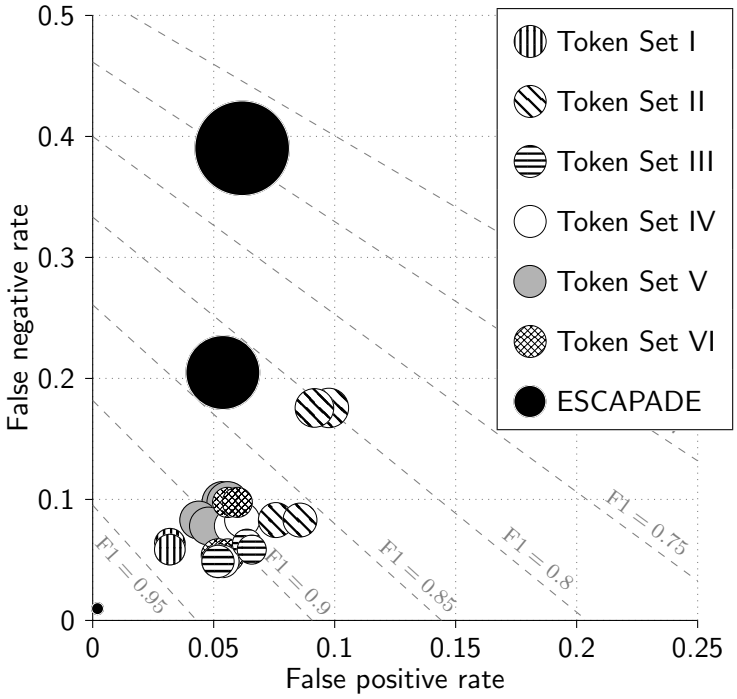

Figure 7.6: Classification model accuracy and size for different tokenisations.

positive and false negative rates are obtained using different supervisor reduction options. Additionally, Table 7.5 shows the classification models with the lowest false negative rates for each tokenisation.

Most of the tokenisations result in accuracy and state numbers comparable to those observed in Section 7.2.3, but there are outliers. The ESCAPADE tokenisation produces both the most and the least accurate classification models by large margins. With projection enabled, it produces a false positive rate of 0.2% and a false negative rate of 1.0%. The classification models have only 12 states on average and use only some of the tokens related to user directory access, while the other tokens (including all tokens related to non-user directories) have been projected out. These accurate and compact models are only found with projection enabled.

Table 7.5: Classification models with the lowest false negative rates for each tokenisation.

	Supervisor Reduction Options			Accuracy					Result	
Tokenisation	States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
Token Set I	RBFS	Diagonal1	Off	0.941	0.059	0.968	0.032	0.932	92	330.1 s
Token Set II	RBFS	Lexicographic	Off	0.917	0.083	0.924	0.076	0.872	106	554.0 s
Token Set III	RBFS	Lexicographic	Off	0.951	0.049	0.948	0.052	0.915	91	416.5 s
Token Set IV	RBFS	Lexicographic	Off	0.951	0.049	0.948	0.052	0.915	91	499.8 s
Token Set V	RBFS	Lexicographic	On	0.922	0.078	0.952	0.048	0.904	124	61.5 s
Token Set VI	RBFS	Lexicographic	Off	0.946	0.054	0.948	0.052	0.913	98	558.7 s
ESCAPADE	RBFS	Lexicographic	On	0.990	0.010	0.998	0.002	0.993	12	3.0 s

Without projection, supervisor reduction fails to identify a reasonable pattern, which may be caused by the large number of tokens or the coarse granularity of the non-user directory tokens.

Unfortunately, the compact and accurate models produced by the ESCAPADE tokenisation may be caused by a bias in the dataset. The user directory is accessed by all 205 malicious applications, but only by 49 of the 502 benign applications, and this explains why projecting out the non-user directory tokens produces so small classification models. Yet, the bias may not only be a reflection of the fact that the user directory is the primary target of ransomware, which has also been observed to encrypt files in other directories. Benign applications may access files in the user directory if requested to do so, but such behaviour is unlikely to be exposed when user interaction is generated randomly by Android Monkey. A different way of capturing traces may be needed to produce more reliable classification models

based on the ESCAPADE tokenisation.

Setting aside the ESCAPADE tokenisation, the lowest false negative rate of 4.9% in this experiment is shared by Token Set IV, which is the set used for all other experiment, and Set III. Set III differs from Set IV by the removal of tokens that have been removed by projection in Section 7.2.2, and this may explain the similarities. The best F1 score of 93.8% and the lowest false positive rate of 3.2% besides ESCAPADE is obtained by Set I. This may be a surprise, considering that Set I is the coarsest tokenisation that does not differentiate between the `openat()` system calls based on their flags. Yet, the small number of tokens makes it easy for supervisor reduction to minimise the FSM, while the low level of loop detection in this experiment seems to be enough to reduce misclassification. The slightly finer Set II does not achieve the same accuracy and produces some of the most inaccurate models. It seems that the increased number of tokens makes it more difficult to minimise the FSM, and the model does not benefit from the distinction between the write operations in the remaining tokenisations.

This experiment suggests that a tokenisation similar to Set III or Set IV works well, which recognises directory access, file read access, file creation, and deletion, while possibly filtering out other file access and renaming. A coarse tokenisation like Set I also is a strong candidate, but this may be less reliable as evidenced by the results for Set II.

7.2.5 Unknown Ransomware Detection

As the malware landscape continues to evolve rapidly, one of the main concerns for anti-malware products is the need to detect unknown malware. Therefore,

this section presents an experiment to evaluate the feasibility of the proposed approach to detect unknown encryption-type ransomware. This evaluation follows the same process for generating classification models as Section 4, except that six-fold cross validation is used instead of ten-fold cross validation. Each round of cross validation attempts to detect one of the six ransomware families (Wipelocker, Wannalocker, etc.). For example, a Wipelocker classification model is constructed using the malicious samples from the other five ransomware families, and then it is attempted to classify the Wipelocker samples with this model. This simulates a more challenging scenario where a classifier trained with old data is exposed to an unknown new generation of malware.

The benign samples are randomly split into six sets of equal size, with five of the six sets used to build each classification model. As in Section 7.2.3, the traces are tokenised using Set IV and simplified using one iteration of loop detection with a threshold of eight, and the supervisor reduction options are those identified as most effective in Section 7.2.2, i.e., RBFS pair ordering and Lexicographic or Diagonal1 state ordering.

Figure 7.7 shows the false positive and false negatives rates in relation to the average state numbers of the classification models, which is represented by the size of the circles in the same format as in Figure 7.3. The diagram shows fairly low false positive rates: the highest false positive rate is 20.2% while the majority of the data points falls below 8%. These rates are comparable to those observed in Section 7.2.3, which is expected given the random split of the benign samples.

The false negative rates in this experiment are higher and deserve a more detailed analysis. Hence, Table 7.6 shows the classification models with the lowest false negative rates for each family. These results suggest that the proposed ap-

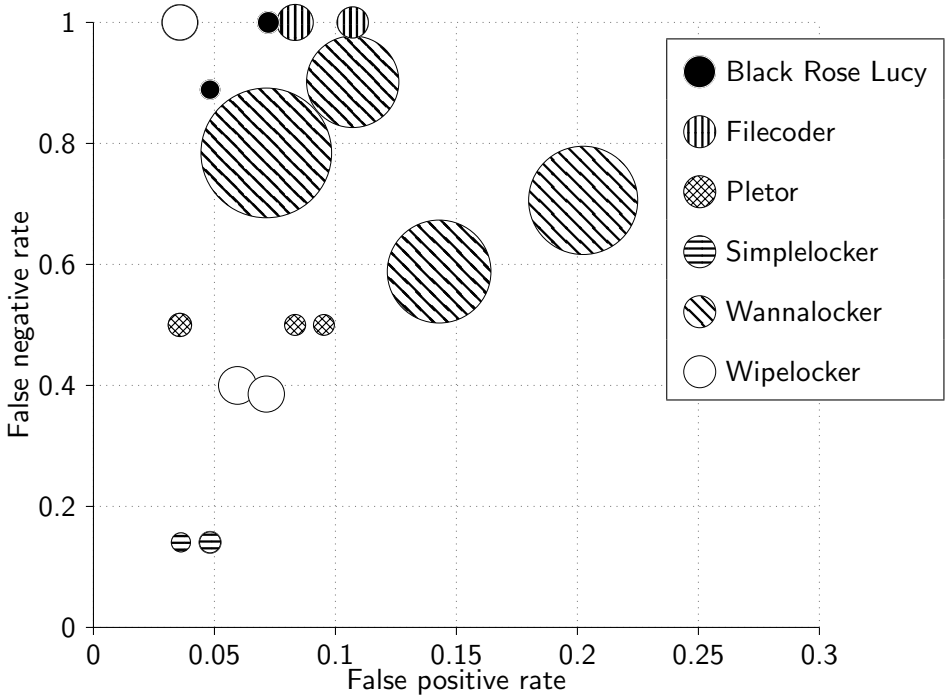

Figure 7.7: Classification model accuracy and size for different ransomware families.

proach is capable of accurately classifying the benign samples, whereas unknown encryption-type ransomware is only detectable in some cases. While up to 86% of Simplelocker samples are recognised as malicious, and some of the Wannalocker and Pletor samples are detected, the remaining families are only recognised in a few cases or not at all. Usually, the classification models with a small to medium number of states have lower false positive rates, but the smallest model is not always the best. The failure to detect some of Wipelocker, Filecoder, and Black Rose Lucy samples is likely due to the fact that these families exhibit unique behavioural aspects, in combination with the small number of only six families sampled.

Table 7.6: Classification models with the lowest false negative rate in each family.

Ransomware Family		Supervisor	Reduction	Options	Accuracy					Result	
Name	Size	States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
Black Rose Lucy	9	RBFS	Diagonal1	On	0.111	0.889	0.952	0.048	0.143	75	51.0 s
Filecoder	5	RBFS	Lexicographic	Off	0.000	1.000	0.917	0.083	0.000	222	53.4 s
Pletor	6	RBFS	Lexicographic	Off	0.500	0.500	0.964	0.036	0.500	92	494.5 s
Simplelocker	64	RBFS	Lexicographic	On	0.859	0.141	0.964	0.036	0.902	62	58.7 s
Wannalocker	51	RBFS	Diagonal1	On	0.412	0.588	0.857	0.143	0.500	1812	20.2 s
Wipelocker	70	RBFS	Diagonal1	On	0.614	0.386	0.929	0.071	0.656	223	18.5 s

7.2.6 Comparison to Related Work

This subsection compares the accuracy of the approach proposed in this work to various static and dynamic methods from relevant literature as referenced in Section 3 of Chapter 2. The results are summarised in Table 7.7.

DNA-Droid (Gharib and Ghorbani, 2017) employs a real-time hybrid approach with static and dynamic components, which is evaluated using cross validation based on dataset of 1928 ransomware samples and 2500 benign samples. Their most accurate result is obtained with the static component and using *model B* with DNN classifier, reporting a true positive rate of 0.981 and false positive rate of 0.005.

R-PackDroid (Maiorca et al., 2017) is another static method. The best reported result is based on cross validation with 440 ransomware samples from Hel-Droid (Andronio et al., 2015). The reported number of 415 of these samples classified as ransomware or malware corresponds to a true positive rate of 0.943.

ESCAPADE (Chew et al., 2020) is based on the same dataset as this work.

Table 7.7: Accuracy of methods to detect Android malware.

Method		Accuracy				
Name	Approach	TPR	FNR	TNR	FPR	F1
Methods to detect Android ransomware:						
DNA-Droid (Gharib and Ghorbani, 2017)	hybrid	0.981	0.019	0.995	0.005	—
R-PackDroid (Maiorca et al., 2017)	static	0.943	0.057	—	—	—
ESCAPADE (Chew et al., 2020)	dynamic	1.000	0.000	0.986	0.014	0.984
Amer and El-Sappagh (2022)	dynamic	0.989	0.011	0.989	0.011	0.980
Methods to detect other types of Android malware:						
DroidAPIMiner (Aafer et al., 2013)	static	0.978	0.022	—	—	—
DroidNative (Alam et al., 2017)	static	0.936	0.064	0.973	0.027	0.844
DroidDet (Zhu et al., 2018)	static	0.933	0.067	—	—	—
SWORD (Bhandari et al., 2018)	dynamic	0.958	0.042	0.926	0.074	0.943
MaMaDroid (Onwuzurike et al., 2019)	static	0.92	0.08	—	—	0.92
Supervisor reduction (this work)	dynamic	0.956	0.044	0.966	0.034	0.938

Their method is based on patters designed manually to capture all malicious traces in the dataset, resulting in a true positive rate of 1. The benign dataset is used to evaluate the efficacy of the patterns, with malicious behavioural patterns also detected in 1.4% of benign applications. These results are not based on cross validation.

Amer and El-Sappagh (2022) use several datasets to train their approach and evaluate it using 10-fold cross validation. One of the experiments uses the ESCAPADE dataset (Chew et al., 2020) and reports both true positive and true negative rates as 0.989. This result is shown in Table 7.7 as the data is the most similar to our work.

The above methods are specifically designed to detect Android ransomware. The second group of entries in Table 7.7 covers a wider group of methods for different types of malware. While possibly less relevant, the detection and evaluation techniques employed in these methods are similar and comparable to ransomware detection.

DroidAPIMiner (Aafer et al., 2013) is a static method that includes four machine learning algorithms, which is evaluated using split validation on a dataset consisting of 3987 malware samples and approximately 16000 benign samples. The most accurate model uses the KNN algorithm with a reported true positive rate of 0.978.

DroidNative (Alam et al., 2017) is another static method and evaluated using split validation on a dataset consisting of 358 malware samples and 3732 benign samples. They report a true positive rate of 0.936 and a false positive rate of 0.027.

Similarly, DroidDet (Zhu et al., 2018) employs a 10-fold cross validation approach to evaluate the methodology using a dataset of 1065 malicious applications and 1065 benign applications. The most accurate model has a true positive rate of 0.933.

SWORD (Bhandari et al., 2018) is a dynamic approach, evaluated using 10-fold cross validation on a dataset consisting of 1000 malicious and 1000 benign applications. The most accurate model achieves a true positive rate of 0.958 and a false positive rate of 0.074.

MaMaDroid's (Onwuzurike et al., 2019) dataset contains 35493 malicious samples and 8447 benign samples, but only parts of the data are used in any single experiment. For the most relevant comparison, Table 7.7 shows results of 10-fold cross validation based on the latest dataset collected in 2016, which contains 2974

malicious samples and 2568 benign samples. Based on this data, the most accurate model produces a true positive rate of 0.92 and an F1-score of 0.92.

The last line in Table 7.7 represents the best classification model obtained with supervisor reduction as mentioned in Section 7.2.3, which is obtained using three iterations of loop detection and a threshold of eight, and lexicographic state ordering without projection. Overall, the approach based on supervisor reduction achieves accuracy scores comparable to those presented in the literature.

7.2.7 Threats to Validity

The main concern about the experiments described in this section is the limited size of the dataset, which consists of only 205 malicious applications from six ransomware families. The sparseness of the dataset raises issues in some experiments as mentioned in Section 7.2.4, and generally makes it difficult to determine how the method scales as the amount of data increases. As this work focuses on a specific type of malware, namely encryption-type ransomware for Android devices, we were unable to obtain more samples. While new ransomware will arise overtime, the dataset in this work is believed to be close to the current extent of Android encryption-type ransomware.

Another possible issue is the randomly generated user interaction through Android Monkey when capturing traces. This random user interaction is unlikely to be representative of typical behaviour patterns of smartphone use. Additionally, traces are captured in on offline environment without Internet access, which makes it impossible to capture typical use cases of benign communication or social media applications. While the presented experiments demonstrate the feasibility to

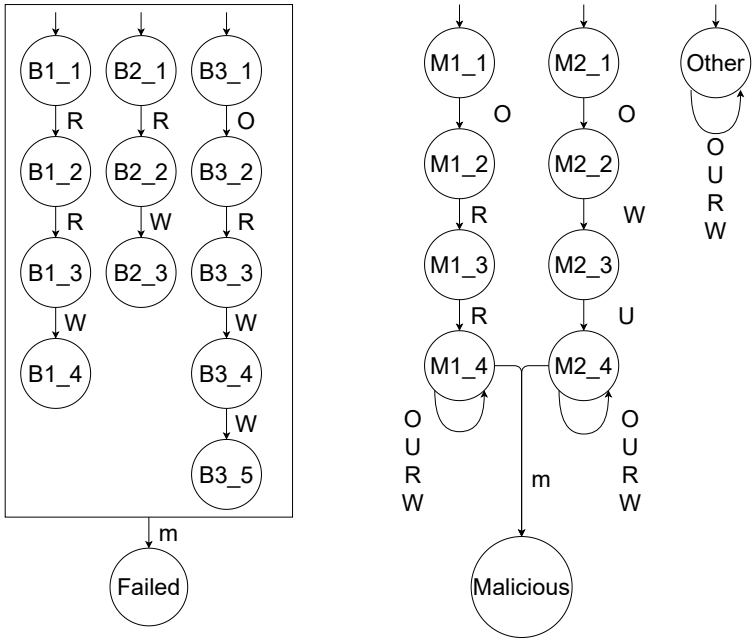

Figure 7.8: Nondeterminsitic FSM model with other state

distinguish applications based on system call traces, different ways of capturing behaviour will have to be explored for deployment on real-user smartphones.

7.2.8 Discussion

A nondeterministic FSM was necessary to generate the supervisor models. Initially, the first model consisted of an additional state defined as `other` with a self-looping transition containing all observed events (i.e., tokens) in the traces. The inclusion of the `other` state ensures that the generated deterministic models and supervisors are complete with all events defined in every state. The result of this created supervisors with the same results regardless of the algorithm or options used. An example of the NFSA with `other` state can be seen in Figure 7.8.

The second nondeterministic FSM model removed the `other` state, which gen-

erates incomplete deterministic and supervisor models, thus every event is no longer existent in every state. The incomplete models can still be utilised as classifiers if the classification process considers blocked states and states that were projected out. This also created more variation in the generated supervisors, thus the classification results are often more diverse. The removal of the other state is more beneficial for classification as the generated deterministic model produces fewer states, thus often providing faster supervisor reduction times and more accurate models. Hence, this chapter utilises the second nondeterminsitic model.

One of the more challenging objective encountered throughout this work was the verification of FSM models. The implementation of reduction methods and conversion from system call traces into Waters/Supremica format should strictly adhere to the rules specified in Section 2.5.2. The generated models were verified to ensure its correctness and prevent inaccuracies in the results. The classification process adopted a 10-fold cross validation approach. As such, each of the classification models generated were verified against the original 9 sets, which were used to generate the models (i.e., training set). A model is correctly generated if it adheres to rule 2.4 defined in Section 2.5.2 of Chapter 2.

7.3 Chapter Summary

This chapter proposes and evaluates a method for automatically detecting encryption-type ransomware at a system call level using a supervisor reduction algorithm. This is achieved by constructing an FSM model to classify benign and malicious applications based on sequences of system calls, which are represented as simplified token traces. Experimental results show that the proposed approach is effective in

detecting encryption-type ransomware with a promising avenue for detecting unknown variants. The classification models can be implemented to monitor real-user devices with minimal processor and memory requirements.

Chapter 8

Identification of Linux Encryption-type Ransomware

Ransomware will continue to be a widespread issue, that affects all operating systems. Hence, there is a demand for more flexible solutions with capabilities to detect ransomware for multiple operating systems. A research contribution of this thesis is to explore the generalisability of applying behavioural patterns and FSMs methodologies that have been devised for Android to detect encryption-type ransomware on a different operating system.

This chapter observes behavioural patterns in Linux encryption-type ransomware by applying methodologies described in Chapter 5 and Chapter 7. Android is based on the Linux Operating System (OS). Hence, one would think that the behavioural patterns should apply to Linux. We investigate if that indeed is the case by exploring the applicability of the previously proposed methodologies for ransomware detection on the Linux OS and further identify the differences between Android and Linux encryption-type ransomware.

8.1 Methodology

This section presents a set of findings for Linux encryption-type ransomware, which differs from the observed behavioural patterns of Android encryption-type ransomware.

8.1.1 Linux Behavioural Observations

Linux is a general purpose operating system generally used for development. Due to this, applications are often more specialised compared to other operating systems. The following two sections explore in more detail the unique traits of benign and malicious Linux applications and the differences in behavioural patterns compared to the Android Applications dataset.

8.1.1.1 Benign Linux Applications

As mentioned in Section 4.2 of Chapter 4, the Linux benign dataset consists of carefully selected applications used in software development. As a result, the samples in the benign dataset are more distinct and contain a broader range of application types and behaviours.

Contributing to the specialised environment of the Linux OS, is the command-line interface, which offers diverse tools. For example, `netcat` is a tool used to monitor Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) connections. Whereas, `diff` can perform comparisons between file and directories. Some of these tools can also perform similar actions to encryption-type ransomware, such as `mv`, which can be used to rename files or directories.

Many applications on Linux rely on passing arguments through the command-

line interface. Command-line arguments are used by applications to perform additional actions. For example, using `ls` on the command-line interface will list the contents of a directory, adding the command-line argument `-S` will list the directory contents sorted by file size in descending order. Similarly, adding the command-line argument `-t` will sort the contents of a directory by the creation time.

The diversity of tools offered by the Linux OS enables the creation of a benign dataset that covers a broader range of behaviours, which was not feasible with the Android Applications dataset due to the randomly simulated interactions from Android Monkey.

8.1.1.2 Malicious Linux Applications

The specialised environment of Linux also incentivises malware developers to engineer their ransomware to target unique and critical files for specific groups and organisations, such as the storage file of virtual machines for ESXi servers (Devkar, 2022). This behaviour differs from the Android samples as most traditional encryption-type ransomware target the average user and common file types, which are considered important to the user, such as `.jpg`, `.txt`, and `.png`. Related to the specific file types, it has been observed that some Linux ransomware samples target a specific directory specified by the user. As most Linux tools or software can be executed through the command-line, malware authors can design their ransomware to encrypt a specific directory to be entered as command-line arguments.

As observed in Android samples, the primary behavioural pattern of an encryption-type ransomware is to perform an unlink on files after the encryption process. For the Linux samples, the behavioural pattern is broader as some samples encrypt

and rename the original file instead of unlinking. Some Linux ransomware samples utilise a more covert approach by encrypting the original file and overwriting its contents without renaming or unlinking. This observation suggests that obtaining more diverse encryption-type ransomware samples can provide a more complete overview of encryption-type ransomware behaviour.

Due to the specific target of Linux ransomware, there are some samples, which can avoid detection if a specific argument, file, or environment was not found or provided. For example, the Hive ransomware searches for a key file to export (Matire and Ragusa, 2022). If the ransomware is unsuccessful in finding the file, then the program would exit without executing the malicious component. Techniques such as searching for specific files or checking for a specific environment (e.g., simulated environments) are common for Android ransomware samples and ransomware in general. However, Linux ransomware utilise techniques, which are distinct to the type of group that the ransomware is targeting (e.g., a ransomware can check for the specific location of an ESXi storage location or identify running background VM processes).

Further observation of the Linux system call logs indicates that there are some minor differences in system call usage compared to Android. Based on the observed Android samples, the system calls produced often contain the suffix *at*, which is used to consider the relative path specified by the file description (Kerrisk, 2022). For example, `unlinkat` and `openat`. In contrast, most Linux samples utilise system call variants without the *at* suffix, such as `unlink` and `open`.

8.2 Evaluation

This section details the evaluation methods and results for detecting Linux encryption-type ransomware. For consistency, we utilised a similar evaluation as described in Chapter 5 and Chapter 7 to show the viability of the two proposed Android methodologies for detecting Linux encryption-type ransomware.

8.2.1 System Call based Behavioural Pattern Alterations

The implementation employed in this work follows a similar methodology as Chapter 5 and Chapter 7. It is important to note that no major alterations were made as the primary objective of this work is to identify the generalisability of the proposed Android methodologies for detecting Linux encryption-type ransomware. However, in Section 8.1.1, it was noted that the behavioural patterns of Linux encryption-type ransomware are slightly different. Hence, minor alterations to the methodology were implemented to accommodate the differing behaviours exhibited by Linux encryption-type ransomware.

One of the main differences between Linux and Android encryption-type ransomware is associated with the type of system calls used. It has been observed that the Android operating system often produces system call logs that utilised the suffix *at*. For example, `openat()`, `unlinkat()`, and `renameat()`. Thus, these specific variations of system calls were white-listed in the filtering process. However, this was an issue for Linux system calls logs, as it was observed that most system calls do not utilise the suffix *at*. Due to the strict filtering process using regular expressions it was not possible to capture the malicious behaviours of Linux encryption-type ransomware, as the regular variants of system calls were not

considered, thus resulting in empty token traces. To alleviate this issue, a minor alteration was made to the tokenisation process to consider system calls with or without the `at` suffix using a conditional regular expression.

The methodology of Chapter 5 relies on predefined and carefully crafted behavioural patterns. One of the additional alterations was to observe the inclusion of a malicious rename pattern to capture the overwriting of user files exhibited by some Linux ransomware samples as mentioned in Section 8.1.1. This pattern follows a similar definition to other previously defined malicious patterns, in which it only observes file renames within the user directory. The following pattern combinations were used to define the new malicious rename pattern, using the same notation as Table 5.1 in Chapter 5:

$$OP(rename) > UD > MF > A \tag{8.1}$$

One of the objectives of this work is to assess the generalisability of the behavioural patterns discovered in Chapter 5. To achieve this, an experiment was conducted to identify malicious behavioural patterns in Linux encryption-type ransomware. The Linux dataset is evaluated multiple times, with each iteration containing a minor alteration to the patterns. Some samples in the Linux dataset produced different system call log formats due to the collection process using `strace`. Hence, all system call logs were formatted to ensure that it is in a consistent formatting for pre-processing. Unlike the evaluation method described in Chapter 5, which observes common malicious behaviours between different Android encryption-type ransomware families, this evaluation observes the detection rates. A malicious sample is classified as true positive if a malicious pattern was discovered in the

Table 8.1: Comparison of detection rates between Linux dataset and Android benign dataset.

Pattern Alteration	TPR	FNR	TNR	FPR	F1
None	0.123	0.877	0.982	0.018	0.212
Additional system call variants	0.616	0.384	0.835	0.165	0.616
Malicious rename pattern	0.836	0.164	0.812	0.188	0.735
Android	1	0	0.986	0.014	0.984

traces, whereas a false negative occurs when no malicious patterns are discovered in the traces. The types of pattern alteration and detection rates of each iteration are shown in Table 8.1.

As shown in Table 8.1, the patterns discovered in Chapter 5 with no pattern alterations detected 9 out of 73 malicious samples, leading to a low true positive rate of 0.123. This can be attributed to the strict regular expressions, which were manually crafted to specifically capture Android encryption-type ransomware. Due to the strictly defined regular expression, the malicious patterns try to identify different system call variants or an alternative form of encryption, as mentioned in Section 8.1.1. This was confirmed after further inspection of the logs files.

By altering the white-listed system calls to include the system call variants without the suffix `at`, the true positive rate increases to 0.616, which is a significant increase compared to the previous case. The increased detection rate is an indication that Linux malware utilises different variants of system calls, but share similar malicious behaviour to Android system calls.

As noted in Section 8.1.1, some Linux encryption-type ransomware overwrite and rename the original file. This behaviour is different from the observed Android samples, which writes the contents to a new encrypted file and removes the original files. The third row of Table 8.1 includes a new malicious rename pattern, which observes renames that occur within the user directory. The inclusion of this pattern results in a true positive rate of 0.836, which is a noticeable increase from the previous two iterations. The increase in detection rates is an indication that Linux encryption-type ransomware exhibits more diverse encryption behaviours compared to the observed Android samples.

In addition to the true positive rates, the results in Table 8.1 also present the true negative rates and false positive rates of the benign Linux dataset in comparison to the benign Android Applications dataset. The true negative rate without any alterations is 0.982, which is comparable to the Android benign true negative rate of 0.986. As further pattern alternations were made, the true negative rate also decreased. The most notable decrease observed was the inclusion of additional system call variants, which lowered the true negative rate to 0.835. This has stark similarities to the results of the malicious Linux dataset in Section 8.2.1, which significantly increased the true positive rate with the inclusion of the additional system call variants. This implies that the benign dataset shares similar trends in patterns compared to the malicious Linux dataset, particularly in the system call variants utilised.

The final pattern alteration further decreased the true negative rate of the benign Linux dataset to 0.812, which is noticeably lower compared to the Android benign dataset of 0.986. This is unsurprising as the Linux benign dataset was carefully designed to contain more diverse samples, which exhibit similar behaviours to

encryption-type ransomware, including samples mimicking the renaming of files. This was not the case with the Android Applications dataset as many of the benign applications did not access user files. This can likely be attributed to the randomly simulated events from Android Monkey, which made it infeasible to replicate all possible events in an application.

Further, inspecting the pattern files of the misclassified benign Linux samples showed that the most prominent malicious pattern to occur was *read user file*, which occurred in 23 samples, followed by 11 occurrences of *unlinking user file*, 10 *rename user files*, 3 *file created with unknown extensions*, and 3 *writing to unknown file extensions*. The predominant occurrence of *read user file* pattern was expected as the pattern closely resembles the behaviours exhibited in benign samples. The same malicious pattern was also observed in the misclassified Android benign applications.

The *unlink user files* and *rename user files* patterns share similar number of occurrence in benign samples. This is a result of the Linux benign samples exhibiting a broader range of file operation behaviours, such as using a text editor to unlink or rename a file.

The last two malicious pattern occurrences were *files created with unknown extensions* and *writing to unknown file extensions*, which were present in 3 benign samples. The benign sample was related to a password manager software, which attempted to create and write to temporary files and special password files using a custom file extension. Distinct and specific software, such as password managers and cache-cleaning applications, are expected to be falsely detected in the absence of any other secondary detection measures. This is often unavoidable due to similarities in high-level behavioural patterns with encryption-type ransomware.

Further, inspecting the false positive rates of the Linux dataset, there is a noticeable increase compared to the Android dataset. This increase is likely attributed to the more diverse behaviours in the benign applications. As previously mentioned, some benign applications in the Linux dataset were specifically crafted to closely resemble behaviours to encryption-type ransomware, such as file delete, and file write. This was done to highlight the efficacy of the aforementioned approach on a more challenging dataset.

This section explored the detection rates of the Linux benign dataset in comparison with the Android Applications dataset. The results without any pattern alteration showed low true positive rates of 0.123. This is a result of the carefully crafted regular expressions, which were specifically designed to detect Android samples. Although effective for Android samples, the patterns in Chapter 5 were an indication of overfitting. By broadening the white-listing process and the set of malicious behaviours, it was possible to achieve a more reasonable true positive rate of up to 0.836. In stark comparison, the true negative rates achieved a true negative rate of 0.982 without any pattern alteration. Further alterations decreased the true negative rate to 0.812, which is likely a result of the more diverse benign Linux applications, which exhibited behaviours similar to that of encryption-type ransomware.

8.2.2 Supervisor Reduction Evaluation

This evaluation presents the results of applying FSM algorithms and supervisor reduction to classify Linux encryption-type ransomware. The evaluation methodology applies a similar approach and metrics as Chapter 7.

Table 8.2: Classification models obtained with 3 iterations and a threshold of 4.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
RBFS	Lexicographic	Off	0.753	0.247	0.982	0.018	0.840	77	794.6 s
RBFS	Diagonal1	Off	0.740	0.260	0.982	0.018	0.831	77	817.0 s
RBFS	Diagonal2	Off	0.329	0.671	1.000	0.000	0.495	12281	923.0 s

Preliminary results in Table 8.2 show the generated classification models for the Linux dataset using three iterations and a threshold of four with a timeout of 1 hour. Based on our previous observations, a timeout of 1 hour was a sufficient indicator to determine if a classification can be generated. These values were selected to identify the upper-bound of generating classification models. The results in the table show fewer generated models and lower accuracy than the Android Applications dataset, implying that the data in the Linux dataset contains more challenging patterns for classification.

As previously mentioned in Section 8.1.1 the benign dataset has been distinctively generated to exhibit a more diverse set of behaviours compared to the Android Applications dataset. This diversity generates a broader range of behavioural patterns, some of which resemble similar file operation behaviours to malicious samples, resulting in a more challenging task for minimisation, thus producing less accurate classification models overall. This could also explain the success of the Android results in Chapter 7 as the Android benign samples were randomly simulated using Android Monkey. Consequently, it was unlikely to observe all behavioural paths that were representative of a typical user. The random

simulation of user interaction was noted as a possible threat to validity of the previous evaluation in Chapter 7 as some benign applications could have exhibited file operation behaviours, which were not explored, resulting in an easier model for classification.

The number of tokens in the Linux samples also contribute to the complexity of the models. The average token length of the malicious Linux dataset is 5101 and the benign dataset is 2885 tokens, which is noticeably longer than the average token length of 489 and 599 for the malicious and benign Android Applications dataset respectively. This observation suggests that malicious Linux samples exhibit more file operation sequences compared to Android malicious samples, and the longer trace lengths presents a more difficult task generating a classification model.

In Section 7.2.7 of Chapter 7, one of the main threats to validity of the Android Applications dataset was the sparseness of the data, which resulted in issues for some experiments. The Linux dataset is affected even more by this issue as it contains fewer samples compared to the Android Applications dataset. Furthermore, the traces present a challenging task for minimisation due to the more diverse benign dataset, which also contributes to lower detection rates.

The preliminary results of this evaluation explored the application of FSM algorithms and supervisor reduction to a Linux encryption-type ransomware. Several challenges were identified, which prevented the Linux classification models from generating more accurate classification models when compared to the Android Applications dataset. However, overall the results show acceptable detection rates with an F1 score up to 0.84, which is better than the highest behavioural pattern F1 score in the previous section.

8.2.3 Filtered Sample Results

The previous section presented a set of preliminary results without modifications to the dataset. Further inspection of the Linux dataset showed that 8 samples encrypted one or two specific files. These files often contain the file extension `.vmdk`, which are storage disk files for virtual machines. This creates a challenging task for supervisor reduction to distinguish between benign and malicious applications as the distinctive patterns that differentiate between malicious and benign file operations are no longer as evident. This section explores the results of removing these 8 malicious samples, leading to a smaller dataset of 65 malicious samples. It is important to note that the removal of these 8 samples can have a negative implication as `.vmdk` files are often important files that can contain valuable information. As such, these specific variants of encryption-type ransomware that target a specific type of file can pose a potential limitation.

In addition to the removal of the 8 samples, further observations of the token traces showed that one of the benign applications generated an empty token trace file as the Set IV did not consider `open()` with the `O_RDWR` flag. To alleviate this issue, a new token set was created in the following experiments, which also considers other uses of `open()` that have not been observed, including `O_RDWR`. This new token set is shown in Table 8.3.

As indicated in Table 8.4 the highest F1 score obtained was 0.847, which was achieved with a pair ordering of RBFS and Diagonal1 state ordering with projection off. In comparison to Table 8.2 the highest F1 score was 0.84 using the same parameters. The minor increase in F1 score suggests that the removal of malicious samples with single file encryption has a positive effect on the accuracy

Table 8.3: New token set.

Operation	New Token Set
Open directory	OD
Open file for reading	ORF
Open file for writing (create)	OCWF
Open file for writing (append)	OAWF
Open file for writing (other + RDWR)	OXF
Rename file	RN
Unlink/delete file	U

of the classification models, however, as previously mentioned, there are other challenges, which contribute to the lower accuracy of the classification models. Overall, these results indicate a minor increase in the detection rates. Hence, the following experiments utilises this malicious dataset of 65 Linux samples.

8.2.4 Evaluation of Supervisor Reduction Options

This section applies the same evaluation methodology as in Section 7.2.2 of Chapter 7. We observe the effects of applying different supervisor reduction and loop detection options to the filtered Linux dataset using a ten-fold cross validation approach. The new token set used in Table 8.3 was utilised instead of Set IV. DFS pair ordering was omitted in the results as some sets were unable to generate a classification model. After initial observations, two iterations of loop detection and a threshold value of 8 produced the most effective models based on the F1

Table 8.4: Classification models obtained with three iterations and threshold of four with removal of 8 samples.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
RBFS	Lexicographic	Off	0.738	0.262	0.982	0.018	0.828	247	794.6 s
RBFS	Diagonal1	Off	0.769	0.231	0.982	0.018	0.847	354	834.6 s
RBFS	Diagonal2	Off	0.277	0.723	0.994	0.006	0.429	12246	932.5 s

Table 8.5: Classification models obtained with 2 iterations and a threshold of 8.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
RBFS	Lexicographic	Off	0.846	0.154	0.982	0.018	0.894	45	772.8 s
RBFS	Lexicographic	On	0.785	0.215	0.965	0.035	0.836	163	872.8 s
RBFS	Diagonal1	Off	0.831	0.169	0.988	0.012	0.893	53	807.3 s
RBFS	Diagonal1	On	0.800	0.200	0.953	0.047	0.832	119	774.3 s
RBFS	Diagonal2	Off	0.323	0.677	1.000	0.000	0.488	15928	872.3 s
RBFS	Diagonal2	On	0.677	0.323	0.953	0.047	0.752	3047	804.4 s

score. The results of this evaluation are shown in Table 8.5.

The results indicate that classification models generated with Lexicographic and Diagonal1 state ordering without projection produced the highest F1 score of 0.894 with a true positive rate of 0.846 and a true negative rate of 0.982. In relation to the state sizes, the data indicates that lower state sizes result in more accurate true positive rates, thus leading to a more accurate overall, similar to the results in Chapter 7.

The classification models generated with projection on can still achieve acceptable detection rates with Lexicographic and Diagonal1. Whereas, Diagonal2 models consistently produce low F1 scores, similar to the Android Applications dataset. Setting aside the Diagonal2 models, the true positive rates of Lexicographic and Diagonal1 are noticeably lower compared to the models without projection.

Further examination of the projection models indicated that all ten rounds of cross validation removed OAWF, nine rounds removed U, two rounds removed ORF, one round removed O, and one round removed OD. The removal of O, ORF, and OD, which corresponds to open, open file for reading, and open directory respectively, appears to occur less frequently in some rounds, which is likely due to the random distribution of samples. Similarly, the removal of the token OAWF, indicating open file for appending, suggests the occurrence of this token is infrequent in the Linux sample, this is similar to the Android Applications dataset, which also observed the removal of OAWF in all ten rounds of cross validation. One of the more notable differences between removed tokens for Android and Linux samples is U, which is the token used to indicate file unlinking. The removal of this token implies that unlinking of files is an infrequent occurrence for Linux samples. This is consistent with Section 8.1.1, which indicated that some Linux samples overwrite the original files using a rename operation as opposed to unlink.

The classifications models produced using the Linux dataset are less effective compared to the Android Applications dataset, which could be a result of a more diverse and smaller dataset. Considering these challenges, the highest F1 score of 0.894 is still acceptable, achieving high true negative rates, with minor difficulties in identifying malicious encryption-type ransomware.

8.2.5 Evaluation of Loop Detection

Similar to Chapter 7, this section observes the effectiveness of loop detection using ten-fold cross validation. As previously mentioned, Set IV from the Android Applications dataset was unable to be used due the empty trace that was produced by a benign application. Although, an empty trace will have no impact on the overall results of the benign set due to the effects of supervisor reduction, this section introduces a new token to the set, which also considers other `open()` system calls, including the `O_RDWR` flag. As shown in the previous section, 3 iterations and a threshold value of 4 were unable to produce the models with projection on. Hence, this evaluation only considers 1 and 2 iterations with a maximum threshold value of 12. Additionally, 2 iterations with a threshold value of 1 were unable to generate models. Therefore, it will be omitted from this evaluation.

Figure 8.1 shows the results of the false positive and false negative rates in relation to the average state numbers of the classification models. The results of this evaluation can also be compared to Table 8.6, which shows the results of the classifications model obtained without loop detection. Furthermore, as shown in Table 8.6, it was not feasible to generate classification models without the use of projection, which was likely attributed to the larger initial state sizes.

The results of Figure 8.1 suggest that the classification models can correctly identify most malicious and benign applications. The highest false positive rate was 0.047 whereas the majority of data points are within 0.01 to 0.03. Conversely, the false negative rates are significantly higher up to 0.31 with the majority of data points falling between 0.18 to 0.27. The high false positive rates are unsurprising due to the challenges previously mentioned.

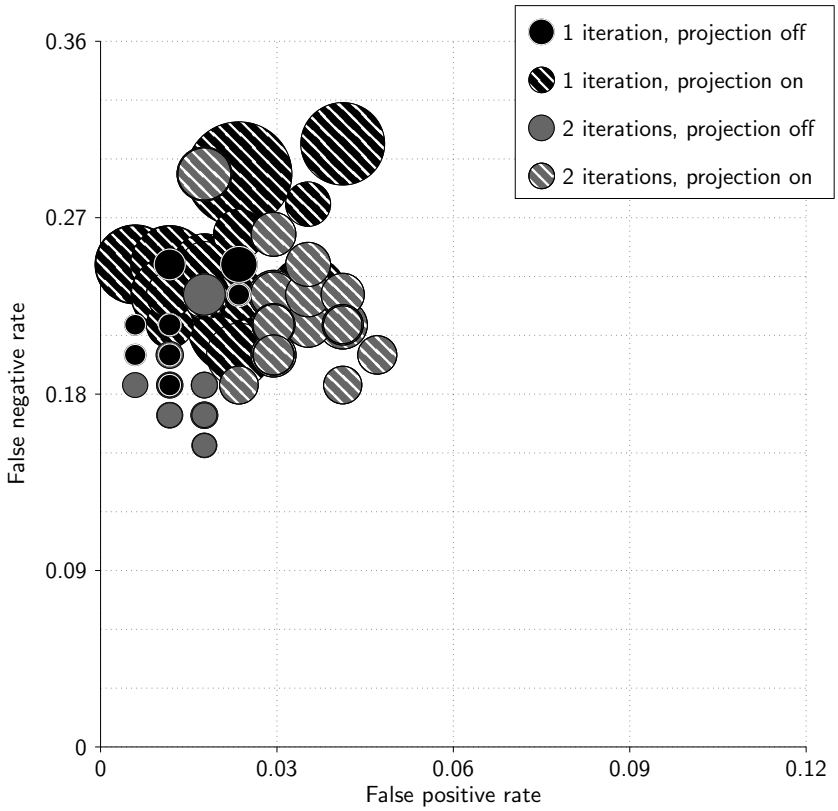

Figure 8.1: Classification model accuracy and size depending on loop detection parameters and projection for the Linux dataset.

Additionally, the models obtained without projection generally produced better classification rates compared to the models with projection, which share stark similarities to the Android Applications dataset. Furthermore, models with projection often produced higher false positives. The trend is more pronounced when observing the iterations for models with projection enabled, which indicates that higher iterations generally produce models with higher false positive rates.

A comparison between Figure 8.1 and Table 8.4 shows that applying loop detection provides several benefits. One of the more noticeable benefits is the

Table 8.6: Classification models obtained without loop detection.

Supervisor Reduction Options			Accuracy					Result	
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
RBFS	Lexicographic	On	0.692	0.308	1.000	0.000	0.818	628	1766.8 s
RBFS	Diagonal1	On	0.692	0.308	0.982	0.018	0.796	628	1873.7 s
RBFS	Diagonal2	On	0.369	0.631	0.976	0.024	0.516	18787	2148.4 s

number of classification models generated. Without loop detection, it was not possible to generate classification models with projection disabled due to the initial large state sizes. By reducing the number of states with loop detection, it was feasible to generate models with projection enabled and disabled. Additionally, the models obtained with loop detection are generally more accurate, with the most effective models achieving an F1 score of up to 0.894. Whereas, models without loop detection can only produce an F1 score up to 0.818. Furthermore, the models have fewer states compared to models without loop detection. The classification model with the fewest states with loop detection has 35 states, whereas, the model with the fewest states without loop detection has 628 states. Conversely, the classification models with the most states have 871 and 18,787 states with loop detection and without loop detection respectively. This implies that loop detection often results in classification models with fewer states, whilst also maintaining a higher accuracy compared to the models without loop detection. The smaller number of states often also results in faster supervisor reduction times.

Overall, this evaluation demonstrates the efficacy of loop detection on the Linux dataset. Similar to the results of the Android Applications dataset, models with

Table 8.7: Android and Linux open system call variants count.

(a) Android open token count

Token	Token Occurrence (%)
RF	50.64%
RDWR	37.56%
CWF	5.51%
O_D	4.76%
Other Write	1.42%
AWF	0.11%

(b) Linux open token count

Token	Token Occurrence (%)
RF	58.15%
RDWR	12.60%
CWF	6.13%
O_D	23.09%
Other Write	0.02%
AWF	0.01%

loop detection are more beneficial than models without loop detection.

8.2.6 Token Set Changes

In Section 8.2.3 it was noted that one of the benign applications produced an empty trace file as Set IV did not observe `open()` system calls with the flag `O_RDWR`. Hence, a new tokenisation was introduced, which also explored other `open()` system calls for writing, including the flag `O_RDWR`. Initially, `O_RDWR` flag was omitted from the token sets in the Android Applications dataset as it was not feasible to generate models for classification. This differed for the Linux dataset and by further exploring the open system call token variants in Table 8.7 we found significant difference between the number of occurrences for `open()` system call variants.

One of the differences between Android and Linux is the open directory variant, which is represented using the token OD. The occurrence of this token is more frequent in Linux system calls compared to Android token traces, which is a

result of applications exhibiting more diverse behaviours, particularly for benign applications. A more important difference between the Android and Linux open variants is the ORW token, which is significantly more prevalent in the Android Applications dataset compared to the Linux dataset. The infrequent occurrence of open system calls with the `O_RDWR` flag in the Linux dataset is likely attributed to the inclusion of more command line tools, which exhibit more traditional file access behaviours, such as opening a file for reading or opening a file for writing. This also explains why it was feasible to generate classification models with the inclusion of open system call variants with the `O_RDWR` whereas the Android Applications dataset produced long traces, which were difficult to process.

Table 8.8: Nine possibilities to tokenise Linux system calls.

	Tokens								
Operation	Set I	Set II	Set III	Set IV	Set V	Set VI	Set VII	Set VIII	Set IX
Open directory	O	OD	OD	OD	OD	OD	OD	OD	OD
Open file for reading	O	ORF	ORF	ORF	OXF	ORF	ORF	ORF	ORF
Open file for writing (create)	O	OWF	OCWF	OCWF	OCWF	OCWF	OCWF	OCWF	OCWF
Open file for writing (append)	O	OWF	OAWF	OAWF	OXF	OXWF	OAWF	OXWF	OAWF
Open file for writing (other)	O	OWF	OXF	—	OXF	OXWF	OXF	OXWF	—
Open File for writing (RDWR)	O	OWF	OXF	—	OXF	OXWF	ORW	ORW	ORW
Rename file	RN	RN	RN	RN	RN	RN	RN	RN	RN
Unlink/delete file	U	U	U	U	U	U	U	U	U

Based on the aforementioned observations, new tokenisations were created. The new tokenisations are presented in Table 8.8 and Table 8.9. These tokenisations differ and expand the Android tokensations to further explore the effects of including `open()` system call variants with the `O_RDWR` flag.

Table 8.9: Altered ESCAPADE tokenisation.

	Tokens	
Operation	User directory	Other directory
Open directory	$\mathsf{OD}_{\mathsf{udir}}$	$\mathsf{OD}_{\mathsf{other}}$
Open file for writing (create)	$\mathsf{OCWF}_{\mathsf{udir}}$	$\mathsf{OF}_{\mathsf{other}}$
Open file for reading or writing (excl. create, incl. RDWR)	$\mathsf{OXF}_{\mathsf{udir}}$	$\mathsf{OF}_{\mathsf{other}}$
Rename file	$\mathsf{RN}_{\mathsf{udir}}$	—
Unlink/delete file	$\mathsf{U}_{\mathsf{udir}}$	$\mathsf{U}_{\mathsf{other}}$

Set I and Set II follow a similar tokenisation process to the Android Applications dataset. Set I tokenises all observed `open()` system call variants with the token O. Whereas, Set II distinguishes between *Open file for reading* and *Open file for writing*. The difference between the tokenisation for the Linux dataset is the inclusion of the additional `open()` system call variant with `O_RDWR`. Set III is the token set used for the previous evaluations, which shares a similar tokenisation to Set IV. The main difference is the inclusion of the additional OXF token, which observes other open file for writing variants, including `O_RDWR`. Token Set IV is the original token set used in the Android Applications dataset. Although this tokenisation does produce an empty benign trace, the trace can still be considered for classification as it would not affect the result of the models due to the effects of supervisor reduction. The fifth and sixth token sets utilise a similar tokenisation to the Android Applications dataset with the inclusion of the new open *file for writing* variant, which considers the flag `O_RDWR`. Token Set V does not distinguish between different open file operations, except *open directory* and *open file for*

writing (create). Conversely, token Set VI does not distinguish between different open file for writing operations except *open file for writing (create)*. Set VII distinguishes between all open file variants, including the new variant with the flag `O_RDWR`. Whereas, Set VIII merges the tokens for *open file for writing (append)* and *(other)*. Due to the low occurrences of other *open file for writing* variants in Table 8.7, Token Set IX explores a tokenisation, which excludes the observation of other variants and includes the new open for writing variant with `O_RDWR`.

For consistency with other token sets, the tokenisation from Chapter 5 will be referred to as ESCAPADE tokenisation. Furthermore, the tokenisation has also been expanded to include `open()` system call variants with the flag `O_RDWR`. As shown in Table 8.9, the new variant is merged with the tokenisations of *open file for reading or writing*.

This section explored the occurrences of `open()` system call variants between Linux and Android traces, which showed that tokenising open system variants with flag `O_RDWR` was feasible due to the significantly lower occurrences in the Linux dataset. Based on the observations, a new set of tokenisations were introduced to extensively explore the effects of including the open system call variant. The following section examines the efficacy of the presented token sets by applying supervisor reduction options.

8.2.7 Evaluation of Different Tokenisations

Following the same evaluation as previously conducted in Chapter 7, this section describes the effects of using different tokenisation options based on the sets defined in Section 8.2.6. Similarly, the experiments conducted in this section use

the most effective options identified in the previous section, which are two loop detection iterations with a threshold value of 8 and the pair ordering of RBFS with Lexicographic or Diagonal1 state orderings.

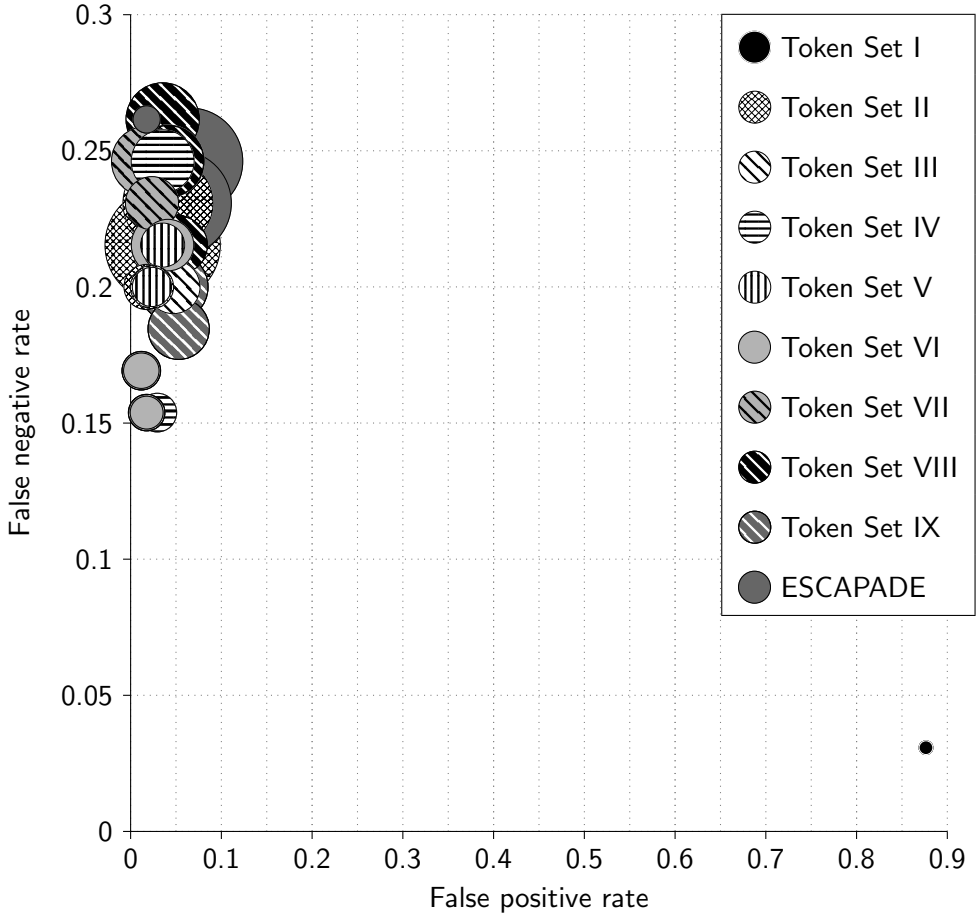

Figure 8.2: Classification model accuracy and size for different Linux tokenisations.

Following the same format as Section 7.2.4, Figure 8.2 shows the false positive and false negative rates of different tokenisations options in relation to the average state sizes. It should be noted that some models achieved the same false positive and false negative rates, which results in overlapping data points. Furthermore,

Table 8.10 shows a more detailed analysis of the classification models with the lowest false negative rates for each tokenisation option.

Table 8.10: Classification models with the lowest false negative rates for each Linux tokenisation.

	Supervisor Reduction Options			Accuracy					Result	
Tokenisation	States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Time
Token Set I	RBFS	Lexicographic	Off	0.969	0.031	0.124	0.876	0.455	8	0.7 s
Token Set II	RBFS	Lexicographic	Off	0.800	0.200	0.982	0.018	0.867	86	791.1 s
Token Set III	RBFS	Lexicographic	Off	0.846	0.154	0.982	0.018	0.894	45	813.2 s
Token Set IV	RBFS	Lexicographic	Off	0.846	0.154	0.971	0.029	0.880	62	681.9 s
Token Set V	RBFS	Diagonal1	Off	0.800	0.200	0.976	0.024	0.860	65	810.8 s
Token Set VI	RBFS	Lexicographic	Off	0.846	0.154	0.982	0.018	0.894	45	802.5 s
Token Set VII	RBFS	Diagonal1	On	0.785	0.215	0.953	0.047	0.823	159	810.1 s
Token Set VIII	RBFS	Diagonal1	On	0.785	0.215	0.953	0.047	0.823	159	806.2 s
Token Set IX	RBFS	Lexicographic	Off	0.846	0.154	0.982	0.018	0.894	55	808.6 s
ESCAPADE	RBFS	Diagonal1	On	0.769	0.231	0.947	0.053	0.806	462	656.1 s

One of the more striking outliers seen in Figure 8.2 is Set I, which produced a low false negative rate of 0.031 and a high false positive rate of 0.876. This can be explained by the abstracted `open()` file operation, which creates a challenging task for supervisor reduction as the patterns exhibited in benign and malicious token traces are no longer evident. Conversely, the Android Applications dataset showed that it was feasible to derive a classification model with an F1 score of 0.932. This difference in accuracy is indicative of the diversity in benign and malicious samples between the two datasets. The Android Applications dataset contained patterns,

which could still accurately classify benign and malicious samples even with the abstracted tokens, whereas the low F1 score in the Linux tokenisation provides a more realistic outcome due to the inclusion of more diverse samples and shows a more reliable dataset overall.

By introducing a more fine-grained tokenisation in Set II, the classifications models are more consistent with the results obtained in the previous experiments, achieving a false negative rate of 0.2 and a false positive rate of 0.018, resulting in an acceptable F1 score of 0.867. In comparison, Set II for the Android Applications dataset produced some of the most inaccurate models and the usage of a finer tokenisation set makes it more difficult to minimise the FSM, while results for the Linux dataset suggests that the models benefit from distinguishing `open()` read and write operations due to the higher accuracy achieved compared to the coarse tokenisation of Set I.

The most effective model for the ESCAPADE tokenisation achieved an F1 score of 0.806 with a false positive rate of 0.053 and a false negative rate of 0.231. In comparison, the ESCAPADE tokenisation from the Android Applications dataset achieved a false positive rate of 0.010 and a false negative rate of 0.002 resulting in an F1 score of 0.999 for the most effective classification model. The ESCAPADE tokenisation from the Android Applications dataset was flawed due to the potentially biased dataset, where only 49 benign applications exhibited user directory access as noted in Section 7.2.4 of Chapter 7, which explains the high F1 score. Although the Linux dataset achieved a noticeably lower F1 score, the outcome was expected given the results of Section 8.2.1, which suggested that ESCAPADE behavioural patterns were less effective in classifying encryption-type ransomware due to the more complex samples in the dataset.

Set VII and Set VIII achieved an F1 score of 0.823 with a false positive rate of 0.047 and a false negative rate of 0.215. Although more effective than the ESCAPADE tokenisation, the classification accuracy is still lower than other sets, such as Set III. The lower classification accuracy could be related to the more fine-grained tokenisation used in Set VII, which creates a more challenging task for minimisation using supervisor reduction. Furthermore, the tokenisation of Set VIII suggests that inclusion of `open()` with the `O_RDWR` flag and not distinguishing between open file for appending and other open file for writing can produce less accurate classification models. Conversely, tokenisations that distinguish between file read access and create file often produce good results as indicated in Set III, Set VI, and Set IX, which obtained the highest F1 scores of 0.894 with a false positive rate of 0.018 and a false negative rate of 0.154.

Overall, the experiment described in this section shows that the inclusion of `open` system calls with the `O_RDWR` flag, and distinguishing between file read access and create file generally produces the most effective classification models as indicated by the results of Set III, Set VI, and Set IX. The accuracy of the classification models obtained in this experiment are lower than the models obtained in the Android Applications dataset, however, the results obtained are indicative of a more realistic dataset.

8.2.8 Discussion

One of the core ideas adopted in this thesis was the tokenisation of system calls and their arguments. This tokenisation process was achieved by manually evaluating the collected systems call logs and constructing regular expressions that matched

specific system calls and arguments, which correlate to observed behavioural patterns. The manual creation of regular expressions is a more targeted approach in identifying specific patterns. However, as previously mentioned in Section 6.2.4 of Chapter 6, these regular expressions are often complex and time-consuming to define. The complexity of these regular expressions can also lead to an overfitted definition, which could result in undetected behaviours. Chapter 7 alleviated these issues by evaluating more tokenisation options with varying degrees of coarseness.

The issue of precise definition for regular expressions arose in this work. It was discovered that the majority of malicious patterns defined in Chapter 5 were unrecognisable in Linux samples. This was caused by minor variations in syntax, behaviour, or different usage of system calls, which share similarities, such as `open()` and `openat()`. This issue is not as prevalent if an automated approach is used to classify the samples, such as the use of supervisor reduction in Chapter 7. Manual construction of patterns using regular expressions might lead to undetected behaviours in the future due to overfitting, thus a more reliable and automated approach of defining patterns would be more advantageous.

8.3 Chapter Summary

This chapter expands on the work described in Chapter 5 and Chapter 7 by exploring the feasibility of applying the methodologies for detecting Linux encryption-type ransomware. Additionally, this chapter identifies the different behavioural patterns between Android and Linux encryption-type ransomware. Our observations and experimental results indicate that Linux encryption-type ransomware often exhibits different behavioural patterns compared to Android encryption-type

ransomware, and minor modifications to the patterns were required. Overall, the results indicate that the proposed methodologies are transferable and can be used to classify Linux encryption-type ransomware.

Chapter 9

Conclusions

This thesis highlighted the growing threat of encryption-type ransomware and presented effective alternative solutions to combat this ongoing issue. This chapter explores closing remarks and concludes with future improvements.

9.1 Closing Remarks

This thesis explored and analysed the behavioural patterns exhibited in encryption-type ransomware at a system call-level for resource constrained devices. Based on this analysis, 12 behavioural patterns were developed using regular expressions, which can be utilised to aid in the development of future anti-ransomware or pattern detection solutions. Using the behavioural patterns discovered, this thesis further presented a real-time detection method to effectively identify encryption-type ransomware at a system call-level using two-layered token FSMs. Additionally, this thesis explored an automated and alternative encryption-type ransomware detection approach using supervisor reduction algorithms. This approach deviated from

traditional ML approaches and presents a promising avenue to counteract the ever-changing malware landscape. The proposed approach can also be implemented in a lightweight monitoring system on a real device, which conforms to the computational and storage constraints of mobile devices. Finally, this thesis showed the transferability and generalisability of methodologies on a different operating system and present a promising path for further development of more effective and alternative behaviour-based malware detection solutions.

9.2 Future Work

The collection of system calls logs on Android heavily relies on the randomly simulated interaction of Android Monkey. As mentioned in Section 7.2.7 of Chapter 7, the random simulation of events is a limitation as it would not be feasible to capture all behaviours exhibited by an application. Hence, a potential avenue for future work is to develop a more robust and deterministic method of capturing behavioural patterns at a system call-level and reduce the uncertainty of behaviours captured.

The generation of classification models using supervisor reduction was a time-consuming process. This was due to the partially manual process of creating the deterministic models. A future improvement for this issue is to develop a fully automated approach to convert any given set of system call traces into a deterministic model. One of the potential solutions to achieve this is by developing a CLI tool, which first converts a set of system call traces into token traces and separates the converted traces into their respective validation-sets. Once the sets are separated, it would then be possible to generate a nondeterministic model, which

can be used in Waters/Supremica to generate a deterministic model. To acquire the deterministic model programmatically, modification of Waters/Supremica is required. This can be achieved by developing a LUA script to call the respective functions for subset construction.

In addition to the potential improvements and amendments of the work proposed in this thesis, there are other areas that the proposed solutions can be applied in. One of those areas is the observation of other malware types. Malware will continue to evolve by adopting more sophisticated techniques. Thus, there is demand for more adaptive anti-malware solutions. A future work is to introduce more behavioural patterns capable of detecting other types of malware, such as Backdoors, and Trojans. This will also alleviate the issue of the limited dataset samples as more types of malware can be observed. The final goal is to identify behavioural patterns for different types of malware and utilise the automated approach proposed in this thesis as part of a monitoring system that recognises malicious applications and stops them before they can cause damage.

An important factor to note is the timing of API and system calls. Works such as Hull et al. (2019) developed a ransomware deployment predictive model (Randep) to predict the deployment characteristics of ransomware. The model observed various variables such as propagation time. It was observed that malware often completed specific operations in different time intervals. In relation to the work proposed in this thesis, one potential approach to incorporate the findings of Hull et al. (2019) is to introduce a delay token mechanism. By implementing this mechanism, a token would be produced at regular intervals when no activity occurs or when the application is in an idle state. This strategy allows for the monitoring of delays that may occur within the application.

The methodology proposed in this thesis focuses on the detection of encryption-type ransomware based on tokenised system calls. Fundamentally, it is achievable as the classification models are only reliant on tokenised system call traces, which were derived from system call logs. A classification model can be generated given that an arbitrary log is tokenisable. This presents opportunities to apply the methodology in other areas of cybersecurity, such as the observation of network traffic to identify anomalous activity.

References

Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, volume 127 of *LNICST*, pages 86–103. Springer International Publishing, 2013. doi: 10.1007/978-3-319-04283-1_6.

Knut Åkesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica—an Integrated Environment for Verification, Synthesis and Simulation of Discrete Event Systems. In *8th International Workshop on Discrete Event Systems*, pages 384–385. IEEE, 2006. doi: 10.1109/WODES.2006.382401.

Muhammad Shabbir Abbasi, Harith Al-Sahaf, Masood Mansoori, and Ian Welch. Behavior-based ransomware classification: A particle swarm optimization wrapper-based approach for feature selection. *Applied Soft Computing*, 121:108744, 2022. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2022.108744. URL `https://www.sciencedirect.com/science/article/pii/S1568494622001867`.

Faitouri A Aboaoja, Anazida Zainal, Fuad A Ghaleb, Bander Ali Saleh Al-rimy, Taiseer Abdalla Elfadil Eisa, and Asma Abbas Hassan Elnour. Malware detec-

tion issues, challenges, and future directions: A survey. *Applied Sciences*, 12 (17):8482, 2022.

Bander Ali Saleh Al-rimy, Mohd Aizaini Maarof, and Syed Zainudeen Mohd Shaid. Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions. *Computers & Security*, 74:144–166, 2018a. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.01.001. URL https://www.sciencedirect.com/science/article/pii/S016740481830004X.

Bander Ali Saleh Al-rimy, Mohd Aizaini Maarof, and Syed Zainudeen Mohd Shaid. Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions. *Computers & Security*, 74:144–166, 2018b.

Shahid Alam, R Nigel Horspool, Issa Traore, and Ibrahim Sogukpinar. A framework for metamorphic malware analysis and real-time detection. *computers & security*, 48:212–233, 2015.

Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. DroidNative: Automating and optimizing detection of Android native code malware variants. *Computers & Security*, 65:230–246, 2017. ISSN 0167-4048. doi: 10.1016/j.cose.2016.11.011. URL https://www.sciencedirect.com/science/article/pii/S016740481630164X.

Samah Alsoghyer and Iman Almomani. Ransomware detection system for android applications. *Electronics*, 8(8):868, Aug 2019. ISSN 2079-9292. doi: 10.3390/electronics8080868. URL http://dx.doi.org/10.3390/electronics8080868.

Nisreen Alzahrani and Daniyal Alghazzawi. A Review on Android Ransomware Detection Using Deep Learning Techniques. In *Proceedings of the 11th Interna-*

tional Conference on Management of Digital EcoSystems, pages 330–335, New York, United States, 2019. Association for Computing Machinery.

Eslam Amer and Shaker El-Sappagh. Robust Deep Learning Early Alarm Prediction Model Based on Behavioural Smell for Android Malware. *Computers & Security*, 116:102670, 2022. ISSN 0167-4048. doi: 10.1016/j.cose.2022.102670.

Nicoló Andronio, Stefano Zanero, and Federico Maggi. HelDroid: Dissecting and Detecting Mobile Ransomware. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *RAID 2015: Research in Attacks, Intrusions, and Defenses*, volume 9404 of *LNCS*, pages 382–404. Springer International Publishing, 2015. doi: 10.1007/978-3-319-26362-5_18.

APKPure. APKPure, n.d. URL `https://apkpure.com/`.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.

Sana Aurangzeb, Muhammad Aleem, Muhammad Azhar Iqbal, and Muhammad Arshad Islam. Ransomware: A Survey and Trends. *Journal of Information Assurance & Security*, 12(2):48–58, 2017.

Avast Blog. Avast, n.d. URL `https://blog.avast.com/`.

Haseeb Awan. Mobile Security Threats Prediction for 2023. Efani, N.A., 2023. URL `https://www.efani.com/blog/mobile-threats-prediction-2023`.

Khaled Bakour, H. Murat Ünver, and Razan Ghanem. The Android malware static analysis: Techniques, limitations, and open challenges. In *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, pages 586–593. IEEE Computer Society, 2018. doi: 10.1109/UBMK.2018.8566573.

Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior Abstraction in Malware Analysis. In *International Conference on Runtime Verification*, volume 6418 of *LNCS*, pages 168–182. Springer International Publishing, 2010. doi: 10.1007/978-3-642-16612-9_14.

Shweta Bhandari, Rekha Panihar, Smita Naval, Vijay Laxmi, Akka Zemmari, and Manoj Singh Gaur. SWORD: Semantic aWare andrOid malwaRe Detector. *Journal of Information Security and Applications*, 42:46–56, 2018. doi: 10.1016/j.jisa.2018.07.003.

Taniya Bhatia and Rishabh Kaushal. Malware Detection in Android based on Dynamic Analysis. In *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–6. IEEE, 2017.

Seyyed Mojtaba Bidoki, Saeed Jalili, and Asghar Tajoddin. PbMMD: A novel policy based multi-process malware detection. *Engineering Applications of Artificial Intelligence*, 60:57–70, 2017.

Miriam Burrell and RNZ. NZ spy agency assisting Waikato DHB after cyber attack/ransom demand. New Zealand Herald, May 18, 2021. URL `https://www.nzherald.co.nz/nz/nz-spy-agency-assisting-waikato-dhb-after-cyber-attackransom-demand/V2Q3ESGHZC3KPHUUQ7R7PNNRWU/?ref=readmore`.

Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. Uncovering the face of Android ransomware: Characterization and real-time detection. *IEEE Transactions on Information Forensics and Security*, 13 (5):1286–1300, 2017.

Xiao Chen, Wanli Chen, Kui Liu, Chunyang Chen, and Li Li. A comparative study of smartphone and smartwatch apps. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1484–1493, 2021.

Christopher Jun-Wen Chew, Vimal Kumar, Panos Patros, and Robi Malik. Escapade: Encryption-type-ransomware: System call based pattern detection. In *International Conference on Network and System Security*, pages 388–407. Springer, Springer International Publishing, 2020.

Christopher Jun-Wen Chew, Robi Malik, Vimal Kumar, and Panos Patros. Automatic Detection of Encryption-type Ransomware using Supervisor Reduction. *Under review*, 2022.

Kevin Collier. Baby died because of ransomware attack on hospital, suit says. NBCNews, October 1, 2021. URL `https://www.nbcnews.com/news/baby-died-due-ransomware-attack-hospital-suit-claims-rcna2465`.

Sudhir Devkar. Avoslocker – Modern Linux Ransomware Threats. VMware, February 25, 2022. URL `https://blogs.vmware.com/security/2022/02/avoslocker-modern-linux-ransomware-threats.html`.

Dissent. Consulate Health Care chain hit by Hive. DataBreaches, January 6, 2023. URL `https://www.databreaches.net/consulate-health-care-chain-hit-by-hive/`.

Shaoyong Du, Pengxiong Zhu, Jingyu Hua, Zhiyun Qian, Zhao Zhang, Xiaoyu Chen, and Sheng Zhong. An Empirical Analysis of Hazardous Uses of Android Shared Storage. *IEEE Transactions on Dependable and Secure Computing*, 18 (1):340–355, 2021. doi: 10.1109/TDSC.2018.2889486.

William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, 32(2), 2014. doi: 10.1145/2619091.

Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, page 87–95, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131496. doi: 10.1145/335169.335201. URL `https://doi.org/10.1145/335169.335201`.

Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2014.

Alberto Ferrante, Miroslaw Malek, Fabio Martinelli, Francesco Mercaldo, and Jelena Milosevic. Extinguishing Ransomware - A Hybrid Approach to Android Ransomware Detection. In *International symposium on foundations and practice of security*, pages 242–258. Springer, 2017.

Savan Gadhiya and Kaushal Bhavsar. Techniques for malware analysis. *Interna-*

tional Journal of Advanced Research in Computer Science and Software Engineering, 3(4), 2013.

Pranit Gaikwad, Dilip Motwani, and Vinayak Shinde. Survey on malware detection techniques. *International Journal of Modern Trends in Engineering and Research*, 21(7):1–25, 2015.

Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

Sergiu Gatlan. CD PROJEKT RED gaming studio hit by ransomware attack. BleepingComputer, February 9, 2021. URL `https://www.bleepingcomputer.com/news/security/cd-projekt-red-gaming-studio-hit-by-ransomware-attack`.

Alexandre Gazet. Comparative analysis of various ransomware virii. *Journal in computer virology*, 6(1):77–90, 2010.

Amirhossein Gharib and Ali Ghorbani. DNA-Droid: A real-time Android ransomware detection framework. In Zheng Yan, Refik Molva, Wojciech Mazurczyk, and Raimo Kantola, editors, *Network and System Security*, volume 10394 of *LNCS*, pages 184–198. Springer International Publishing, 2017. doi: 10.1007/978-3-319-64701-2_14.

GitHub. Github, n.d. URL `https://github.com/`.

Julia Glazova. New ransomware: a cross-platform future. Kaspersky, July 20, 2022. URL `https://www.kaspersky.com/blog/luna-blackbasta-ransomware/44900/`.

Google. Use Google Play Protect to help keep your apps safe and your data private. Google, 2019a. URL https://support.google.com/googleplay/answer/2812853?hl=en.

Google. Android Security & Privacy 2018 Year In Review. Google, 2019b. URL https://source.android.com/static/docs/security/overview/reports/Google_Android_Security_2018_Report_Final.pdf.

Google. Android Debug Bridge (adb). Google, 2020a. URL https://developer.android.com/studio/command-line/adb.

Google. UI/Application Exerciser Monkey. Google, 2020b. URL https://developer.android.com/studio/test/monkey.

Google. Platform Architecture. Google, 2021. URL https://developer.android.com/guide/platform.

Google. UI/Application Exerciser Monkey. Google, 2022a. URL https://developer.android.com/studio/test/monkey.

Google. Application Sandbox. Google, 2022b. URL https://source.android.com/security/app-sandbox.

Google. Privacy, Deception and Device Abuse. Google, n.d.a. URL https://play.google.com/about/privacy-security-deception/user-data/.

Google. Unwanted software policy. Google, n.d.b. URL https://www.google.com/about/unwanted-software-policy.html.

Naveen Goud. Black Rose Lucy Ransomware attack on Android Devices, n.d. URL https://www.cybersecurity-insiders.com/black-rose-lucy-ransomware-attack-on-android-devices/.

Alejandro Guerra-Manzanares, Marcin Luckner, and Hayretdin Bahsi. Android Malware Concept Drift using System Calls: Detection, Characterization and Challenges. *Expert Systems with Applications*, 206:117200, 2022. doi: 10.1016/j.eswa.2022.117200.

Help Net Security. The long-term psychological effects of ransomware attacks. Help Net Security, October 25, 2022. URL https://www.helpnetsecurity.com/2022/10/25/psychological-effects-ransomware/.

Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Boston, 2nd edition, 2001. ISBN 0201441241.

Oliva Hou. A Look at Google Bouncer [Blog post]. Trend Micro, July 20, 2012. URL https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/.

Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. Deep4MalDroid: A deep learning framework for Android malware detection based on Linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111, 2016. doi: 10.1109/WIW.2016.040.

Gavin Hull, Henna John, and Budi Arief. Ransomware deployment methods and analysis: views from a predictive model and human responses. *Crime Science*, 8(2), 2019. doi: 10.1186/s40163-019-0097-9.

Stefano Iannucci, Sherif Abdelwahed, Andrea Montemaggio, Melissa Hannis, Leslie Leonard, Jason S King, and John A Hamilton. A model-integrated approach to designing self-protecting systems. *IEEE Transactions on Software Engineering*, 46(12):1380–1392, 2018.

Intel. Pin. Intel Corporation, 2020. URL `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html`.

Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In *2011 Seventh International Conference on Computational Intelligence and Security*, pages 1011–1015. IEEE Computer Society, 2011a. doi: 10.1109/CIS.2011.226.

Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based behavior analysis for Android malware detection. In *2011 Seventh International Conference on Computational Intelligence and Security*, pages 1011–1015. IEEE, 2011b.

Kaggle. Kaggle, n.d. URL `https://www.kaggle.com/`.

Meet Kanwal and Sanjeev Thakur. An app based on static analysis for Android ransomware. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 813–818. IEEE, 2017.

Michael Kerrisk. intro(2) — Linux manual page. Linux Programmer's Manual,

August 18, 2021a. URL https://man7.org/linux/man-pages/man2/intro.2.html.

Michael Kerrisk. close(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021b. URL https://man7.org/linux/man-pages/man2/close.2.html.

Michael Kerrisk. exit(3) — Linux manual page. Linux Programmer's Manual, March 22, 2021c. URL https://man7.org/linux/man-pages/man3/exit.3.html.

Michael Kerrisk. getpid(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021d. URL https://man7.org/linux/man-pages/man2/getpid.2.html.

Michael Kerrisk. ioctl(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021e. URL https://man7.org/linux/man-pages/man2/ioctl.2.html.

Michael Kerrisk. mmap2(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021f. URL https://man7.org/linux/man-pages/man2/mmap2.2.html.

Michael Kerrisk. read(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021g. URL https://man7.org/linux/man-pages/man2/read.2.html.

Michael Kerrisk. syscalls(2) — Linux manual page. Linux Programmer's Man-

ual, August 18, 2021h. URL https://man7.org/linux/man-pages/man2/syscalls.2.html.

Michael Kerrisk. umask(2) — Linux manual page. Linux Programmer's Manual, March 22, 2021i. URL https://man7.org/linux/man-pages/man2/umask.2.html.

Michael Kerrisk. open(2) — Linux manual page. Linux Programmer's Manual, 2022. URL https://man7.org/linux/man-pages/man2/open.2.html.

S Kok, Azween Abdullah, NZ Jhanjhi, and Mahadevan Supramaniam. Ransomware, threat and detection techniques: A review. *Int. J. Computer Science and Network Security*, 19(2):136, 2019.

Koodous. Koodous, n.d. URL https://koodous.com/.

Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the Detection of Anomalous System Call Arguments. In Einar Snekkenes and Dieter Gollmann, editors, *Computer Security – ESORICS 2003*, pages 326–343, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39650-5.

P. Ravi Kumar and Hj Rudy Erwan Bin Hj Ramlie. Anatomy of Ransomware: Attack Stages, Patterns and Handling Techniques. In Wida Susanty Haji Suhaili, Nor Zainah Siau, Saiful Omar, and Somnuk Phon-Amuaisuk, editors, *Computational Intelligence in Information Systems*, volume 1321 of *AISC*, pages 205–214. Springer International Publishing, 2021. doi: 10.1007/978-3-030-68133-3_20.

Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark Android mal-

ware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2018.

Yassine Lemmou, Jean-Louis Lanet, and El Mamoun Souidi. A behavioural in-depth analysis of ransomware infection. *IET Information Security*, 15(1):38–58, 2021. doi: 10.1049/ise2.12004.

leonv024 and HugoLB. Raasnet, 2019. URL `https://github.com/leonv024/RAASNet`.

Dmitry V. Levin. Strace. Strace, n.d. URL `https://strace.io/`.

Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88: 67–95, 2017. doi: https://doi.org/10.1016/j.infsof.2017.04.001.

Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying Android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

Robert Lipovský, Lukáš Štefanko, and Gabriel Braniša. The Rise of Android Ransomware. Eset, 2016. URL `https://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf`.

Hiroshi Lockheimer. Android and Security [Blog post]. Google, February 2, 2012. URL `https://googlemobile.blogspot.com/2012/02/android-and-security.html`.

Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2008.

Federico Maggi, Matteo Matteucci, and Stefano Zanero. Reducing false positives in anomaly detectors through fuzzy alert aggregation. *Information Fusion*, 10 (4):300–311, 2009.

Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. R-PackDroid: API package-based characterization and detection of mobile ransomware. In *SAC'17: Proceedings of the Symposium on Applied Computing*, pages 1718–1723. Association for Computing Machinery, 2017. doi: 10.1145/3019612.3019793.

Robi Malik. Supervisor Reduction by Hiding Events. 1st Virtual IFAC World Congress, IFAC-V 2020, July 2020.

Malware Bazaar. Malware Bazaar, n.d. URL `https://bazaar.abuse.ch/`.

Ohad Mana, Aviran Hazum, Bogdan Melnykov, and Liav Kuperman. Lucy's Back: Ransomware Goes Mobile. Check Point Research, April 28, 2020. URL `https://research.checkpoint.com/2020/lucys-back-ransomware-goes-mobile/`.

Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC, New York, USA, 2011.

Luigi Matire and Carmelo Ragusa. On the FootSteps of Hive Ransomware. Yoroi, July 26, 2022. URL `https://yoroi.company/research/on-the-footsteps-of-hive-ransomware/`.

David McConnell. The Current State of Ransomware in Today's World and Why the Future is Bleak, 2017. URL https://www.cs.tufts.edu/comp/116/archive/fall2017/dmcconnell.pdf.

Daniel McDermott. Cross-platform attacks could make ransomware even more deadly. Get Cyber Resilient, January 27, 2023. URL https://www.getcyberresilient.com/threat-insights/cross-platform-attacks-could-make-ransomware-even-more-deadly.

Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. RWGuard: A Real-Time Detection System Against Cryptographic Ransomware. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 114–136, Cham, 2018. Springer International Publishing. ISBN 978-3-030-00470-5.

Trend Micro. Behind the Android Menace: Malicious Apps. Trend Micro, n.d. URL https://blog.trendmicro.com/trendlabs-security-intelligence/infographic-behind-the-android-menace-malicious-apps.

Microsoft. Visual Studio Code. Microsoft, n.d. URL https://code.visualstudio.com/.

Maggie Miller. The mounting death toll of hospital cyberattacks. Politico, December 28, 2022. URL https://www.politico.com/news/2022/12/28/cyberattacks-u-s-hospitals-00075638.

Adel Hamdan Mohammad. Ransomware Evolution, Growth and Recommendation for Detection. *Modern Applied Science*, 14(3), 2020.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

Philip O'Kane, Sakir Sezer, and Domhnall Carlin. Evolution of ransomware. *IET Networks*, 7(5):321–327, 2018.

Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Transactions on Privacy and Security*, 22(2), 2019. doi: 10.1145/3313391.

Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—A state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

Bogdan Petrovan. "Dalvik is dead": Google replaces Dalvik with ART in AOSP master branch. Android Authority, June 19, 2014. URL `https://www.androidauthority.com/art-replaces-dalvik-aosp-394904/`.

Peter J. G. Ramadge and W. Murray Wonham. The Control of Discrete Event Systems. *Proc. IEEE*, 77(1):81–98, January 1989. doi: 10.1109/5.21072.

Ronny Richardson and Max M North. Ransomware: Evolution, mitigation and prevention. *International Management Review*, 13(1):10, 2017.

Michele Scalas, Davide Maiorca, Francesco Mercaldo, Corrado Aaron Visaggio, Fabio Martinelli, and Giorgio Giacinto. On the effectiveness of system api-related information for android ransomware detection. *Computers & Secu-*

rity, 86:168–182, 2019. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2019.06.004. URL https://www.sciencedirect.com/science/article/pii/S0167404819301178.

R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 144–155. IEEE Computer Society, 2000. doi: 10.1109/SECPRI.2001.924295.

Abraham. Silberschatz. *Operating system concepts*. Addison Wesley Longman, Reading, Mass, 5th ed. edition, 1998. ISBN 0201591138.

Charilaos Skandylas and Narges Khakpour. Design and Implementation of Self-Protecting systems: A Formal Approach. *Future Generation Computer Systems*, 115:421–437, 2021.

Sanggeun Song, Bongjoon Kim, and Sangjun Lee. The effective ransomware prevention technique using process monitoring on Android platform. *Mobile Information Systems*, 2016, 2016.

Gaurav Sood. *Virustotal: R Client for the virustotal API*, 2017. R package version 0.2.1.

Sophos. The State of Ransomware in Retail 2021. Sophos, 2021. URL https://www.sophos.com/en-us/medialibrary/pdfs/whitepaper/sophos-state-of-ransomware-retail-2021-wp.pdf.

Sophos. The State of Ransomware 2022. Sophos, 2022. URL https:

//assets.sophos.com/X24WTUEQ/at/4zpw59pnkpxxnhfhgj9bxgj9/sophos-state-of-ransomware-2022-wp.pdf.

Abhinav Srivastava, Andrea Lanzi, Jonathon Giffin, and Davide Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 214–233. Springer, 2011.

StatCounter. Mobile Operating System Market Share Worldwide. StatCounter, September 2022. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

Statista. Forecast number of mobile devices worldwide from 2020 to 2025 (in billions). Statista, 2021. URL https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/.

David RB Stockwell and A Townsend Peterson. Effects of sample size on accuracy of species distribution models. *Ecological modelling*, 148(1):1–13, 2002.

R. Su and W. Murray Wonham. Supervisor Reduction for Discrete-Event Systems. *Discrete Event Dyn. Syst.*, 14(1):31–53, January 2004. doi: 10.1023/B:DISC.0000005009.40749.b6.

Sixian Sun, Xiao Fu, Hao Ruan, Xiaojiang Du, Bin Luo, and Mohsen Guizani. Real-time behavior analysis and identification for Android application. *IEEE Access*, 6:38041–38051, 2018.

Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *NDSS*

Symposium 2015. Internet Society, February 2015. doi: 10.14722/ndss.2015.23145.

Sai Deep Tetali. Keeping 2 billion Android devices safe with machine learning. Google, May 24, 2018. URL `https://android-developers.googleblog.com/2018/05/keeping-2-billion-android-devices-safe.html`.

Fei Tong and Zheng Yan. A hybrid approach of mobile malware detection in Android. *Journal of Parallel and Distributed computing*, 103:22–31, 2017.

Bill Toulas. Medibank now says hackers accessed all its customers' personal data. Bleeping Computer, October 26, 2022. URL `https://www.bleepingcomputer.com/news/security/medibank-now-says-hackers-accessed-all-its-customers-personal-data/`.

Jovi Umawing. Software provider denied insurance payout after ransomware attack. MalwareBytes, January 5, 2023. URL `https://www.malwarebytes.com/blog/news/2023/01/software-provider-denied-insurance-payout-after-ransomware-attack`.

Dolly Uppal, Vishakha Mehra, and Vinod Verma. Basic survey on malware analysis, tools and techniques. *International Journal on Computational Sciences & Applications (IJCSA)*, 4(1):103, 2014.

Andrius Vabalas, Emma Gowen, Ellen Poliakoff, and Alexander J Casson. Machine learning algorithm validation with a limited sample size. *PloS one*, 14(11): e0224365, 2019.

A. F. Vaz and W. M. Wonham. On supervisor reduction in discrete-event systems. *Int. J. Control*, 44(2):475–491, 1986. doi: 10.1080/00207178608933613.

VirusShare. VirusShare, n.d. URL `https://virusshare.com/`.

WeLiveSecurity. WeLiveSecurity, n.d. URL `https://www.welivesecurity.com/`.

Ryszard Wiśniewski. Apktool. APKTool, 2010. URL `https://ibotpeaches.github.io/Apktool/`.

Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552. USENIX Association, 2012.

Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

Hui-Juan Zhu, Zhu-Hong You, Ze-Xuan Zhu, Wei-Lei Shi, Xing Chen, and Li Cheng. DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646, 2018. doi: 10.1016/j.neucom.2017.07.030.