International Journal of Information Security
https://doi.org/10.1007/s10207-024-00819-x

REGULAR CONTRIBUTION

Real-time system call-based ransomware detection

Christopher Jun Wen Chew[1] · Vimal Kumar[1] · Panos Patros[2] · Robi Malik[2]

Abstract
Ransomware, particularly crypto ransomware, has emerged as the go-to malware for threat actors aiming to compromise data on Android devices as well as in general. In this paper, we present a ransomware detection technique based on behaviours observed in the system calls performed by the malware. We first describe our repeatable and extensible methodology for extracting the system call log and patterns. We then identify and present some common high-level system call behavioural patterns exhibited by crypto ransomware, and evaluate these patterns. We further describe the implementation of a streaming implementation that utilises regular expressions for modelling malware behaviours and finite state machines for detecting crypto ransomware behaviours in real time. The success of our proof of concept evaluation allows us to envision our proposed technique applied as part of a self-protection system on Android phones against malware.

Keywords Crypto ransomware · System calls · Behaviour · Patterns · Android · Real-time

1 Introduction

Ransomware attacks, one of the more frequent types of attacks, pose a threat to both consumers and organisations. According to Sophos *The State of Ransomware 2023* report [61], 84% of the organisations surveyed in Singapore were affected by ransomware in 2022. Ransomware heavily impacts businesses monetarily. The report stated that the average ransom payment was $1,542,333 in 2023, almost doubled from the previous year of $812,380.

Due to the severity of ransomware threats, there has been extensive research in ransomware detection and mitigation [3, 51]. However, there is a notable gap in addressing crypto ransomware detection with a focus on overcoming resource constraints in mobile devices. This is particularly concerning because of the steady increase in the global usage of mobile phones, reaching 6.95 billion devices surveyed in 2020 and expected to grow to 7.49 billion by 2025 [63]. The prevalence of mobile phones can often be attributed to the conveniences they offer, such as communication applications, digital wallets, and entertainment applications. As our mobile devices continue to be closely intertwined in our daily lives, they not only store our personal data such as photographs, credit card information, and contacts but also sensitive business and organisational information through apps such as mail, cloud storage and business applications. Consequently, mobile phones have evolved into a high value and portable data storage, making them valuable targets to ransomware and malware attacks. The combination of the current state of ransomware [19], the trend of mobile device usage and the value of data on mobile devices will see mobile devices inevitably become one of the main targets of ransomware attacks. The mobile device market is dominated by two operating systems, Android and iOS. Of the two, Android holds about 70% of the mobile operating system market share [64]. In addition to the official Google Play Store, Android has various third-party app stores available to download apps, in contrast to iOS. The third-party app stores are much less strictly controlled than the official app store making users more susceptible to downloading potentially malicious applications and therefore being introduced to a larger threat surface. Hence, there is a need for more Android-based anti-ransomware solutions.

✉ Christopher Jun Wen Chew
cc246@students.waikato.ac.nz

Vimal Kumar
vimal.kumar@waikato.ac.nz

Robi Malik
robi@waikato.ac.nz

[1] Department of Computer Science, University of Waikato, Hamilton, New Zealand

[2] Department of Software Engineering, University of Waikato, Hamilton, New Zealand

Published online: 02 March 2024

Techniques with varying levels of sophistication and robustness, such as the ones based on machine learning [22, 26], and those based on behaviour [33, 56] have recently been used for identification of malware. Additionally, there have been approaches looking at system calls for dynamically analysing malware. System calls-based approaches 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. Kernel-level analysis offers more depth and resilience; however, the approaches can often lead to a complex design, thus leading to an over-fitted solution, which may not provide any significant benefits in the detection model.

Due to the dynamically changing and ever-evolving threat landscape, malware has become more sophisticated and cunning. Hence, to counteract this rapidly changing landscape, malware detection systems have trended towards real-time malware analysis [1, 46] and self-protection systems [32, 57], which provide benefits, such as early to immediate detection and consistent active monitoring. In this work, we apply the advantages of real-time malware analysis and the aforementioned benefits of system call-level analysis to dynamically identify behavioural patterns of crypto ransomware. By leveraging these two techniques, we aim to address the following research objectives (RO):

- *RO1: Identify system call-level behavioural patterns for crypto ransomware:* while there have been recent works on pattern detection on system call logs [33, 40], none has focused on patterns produced by specific malware types. We aim to discover a set of common behavioural patterns for crypto ransomware, such as file encryption and tampering with user files through the use of system call logs.
- *RO2: Evaluate the effectiveness of the behavioural patterns:* we also evaluate the feasibility and efficacy of these patterns in detecting crypto ransomware behaviours from different families, to discover the shared common behaviour among crypto ransomware.
- *RO3: Generate and make available, a dataset of system call logs of malware activity:* we believe behaviour detection using system calls can be a useful technique for malware detection and analysis, therefore we have made our dataset available for researchers to utilise in malware research.
- *RO4: Implement and evaluate a real-time streaming implementation for crypto ransomware detection:* this research objective aims to introduce a streaming implementation for detecting crypto ransomware in real-time, through the utilisation of token Finite State machines (FSMs). In addition, we evaluate the efficacy and feasibility of this new proposed approach for detecting crypto ransomware.

This paper is an extended version of our previous work ESCAPADE [13] in which we addressed Research Objectives 1 to 3. Additionally, in this paper we address research objective 4, where we apply the methodology of ESCAPADE in a streaming implementation for detecting crypto ransomware. This work also discusses real-time detection as opposed to offline detection as discussed in [13]. The paper is organised as follows. Section 2 details the background and relevant research in our work with Sects. 2.3 and 2.3.4 being extended to include relevant work in our new proposed approach. Section 2.4 describes the methodology utilised in our previous work to acquire our behavioural patterns. Section 3 presents the new proposed approach, which describes the design and architecture of our approach for real-time crypto ransomware detection. In Sect. 4, we detail our extended evaluation of the behavioural patterns and the proposed streaming approach along with some potential threats to validity with our work. In Sect. 4.5 we address the limitations in our work and propose some future improvements. Finally, in Sect. 5, we give an overview of the initial research objectives and how they were achieved.

2 Background and related work

In this section, we detail the evolution and improvements of Android security and its current state; followed by an overview of different types of ransomware, and conclude with the different types of malware analysis techniques used throughout the years, and how our proposed approach can contribute to the existing area.

2.1 Android security history

Since the introduction of Android—a mobile operating system—in 2008, there have been many updates and improvements to its security. In 2012, Bouncer was released in an effort to deter the upsurge of Android malware in the preceding year [47]. Bouncer targeted pre-existing applications as well as new applications. The approach that Bouncer took was sandboxing [41], where applications were executed, and scanned for malware in an isolated environment on a cloud infrastructure, which was devoid of any access to the users' real data.

However, researchers quickly detected the vulnerabilities of Bouncer. Oliva Hou from Trend Micro [30] 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.

Bouncer was, therefore, not a sustainable security mechanism. A few years later in May 2017, a more robust approach known as Play Protect was introduced. 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 ideas: 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.

2.2 Ransomware

Ransomware, a type of malware that holds the users' data for ransom—often requesting monetary payment—generally consist of two types: locker ransomware and crypto ransomware [48]. 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 [7]. Another technique used is altering users' lock screens, thus preventing access to their devices [4, 34].

Crypto ransomware are more destructive where the user's files are encrypted to prevent them from accessing any of their data [4, 35]. Similar to locker-type ransomware, a ransom note is often displayed after the encryption phase has been completed. Typically, for crypto 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 speed up the encryption process, and efficiently determine the important user files to encrypt (i.e. the files most important to a user) [21]. For the encryption process, the data of the identified files are read, and written to a new encrypted file with an unknown file extension. The original file is then removed or overwritten [12].

In recent years, the trend of ransomware attacks have shifted, with crypto ransomware being the more common attacks as compared to locker-type ransomware [5, 45, 53]. One of the more recent works by Bansal [15], further highlighted this trend, by reviewing the most common variants of ransomware attacks, with Cryptolocker and WannaCry showing the highest percentage of attacks, both of which were crypto ransomware.

The aforementioned issue was exemplified by the low mitigation rates for crypto ransomware. For example, in a 2021 State of Ransomware report by Sophos [60], only 34% of cross-sectors and 39% of retail sectors (from surveyed retail IT managers) were successful in preventing ransomware attacks from encrypting their data. Such values implied a higher demand for preventative measures, specifically targeting crypto ransomware; reinforcing our decision to develop an approach, which focused on crypto ransomware.

2.3 Static and dynamic analysis

In static analysis, a malware analyst or computer program, observes the code of the given application and tries to determine if it is malicious or benign, and gains insight on its functionality without the necessity of executing the application. Static analysis, however, has limited effectiveness when more sophisticated malware utilises advanced techniques, such as binary/code/control flow obfuscation, and polymorphic coding [17, 20, 49] to avoid detection.

In dynamic analysis, rather than observing the code, malicious applications are directly executed in an isolated environment and observed over time for malicious behaviour. As such, dynamic analysis is more resilient to obfuscation techniques, which is a common limitation of static analysis [28, 29]. Generally, obfuscation in dynamic analysis is not an issue as dynamic analysis only observes the behaviour of the application at run-time. However, obfuscation techniques to circumvent existing dynamic analysis approaches have also been explored in past literature. A dynamic analysis obfuscation technique of particular relevance is system call obfuscation. Srivastava et al. [62] proposed a system call obfuscation technique by simulating 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`. This is difficult to discern from benign applications due to the marshalling process.

Over the years, researchers have developed unique and robust techniques through the use of static and dynamic analysis aimed at detecting malware or intrusion detection. The following subsections highlight the core areas surrounding our work and how we differentiate our approach from other related work in this area.

2.3.1 Signature and code analysis

One of the more traditional types of static analysis focuses on developing signatures or observing source code. AndroSimilar [18] and DroidMoss [70] adopted the idea of fuzzy hashing which compared similarities between the signatures generated. This produced a percentage of similarity with 100% being an exact match. This approach aimed to counteract the issue of code obfuscation and application

repackaging. However, AndroSimilar [18] produced high false negative rates (28%) when detecting unknown malware and considerably higher false negatives for the various methods of code obfuscation which consisted of method renaming (45%), junk method insertion (44%), goto obfuscation (43%), and string encryption (24%). DroidMoss's false negative rates were lower (10.7%). All the tested applications however, came from third-party app stores, whereas AndroSimiliar focused on both official Play store and third-party app stores.

Compton et al. [14] have also attempted to mitigate the issues of obfuscation through the use of code2vec [2], a method used for extracting information from an Abstract Syntax Tree (AST) derived from a piece of source code, to train models on obfuscated Java source code to reduce code2vec's reliance of variables names. In their evaluation, they utilised 7 datasets to determine if obfuscated variables names provided an improved model for identifying code semantics. One of those datasets was based on Android malware APKs, which showed minimal improvements with the newly trained models as malware are known to utilise sophisticated obfuscated techniques to avoid detection.

Traditional code and signature analysis techniques are known to be effective against known malware. There are, however, evident limitations of these techniques as mentioned in Sect. 2.3. The aforementioned works emphasise some of the techniques adopted by researchers to counteract the limitations. However, one of the core limitations of utilising signature and code analysis stems from the inability to detect newer and unknown variants of malware, which becomes a major issue as the malware landscape continues to evolve.

Many have attempted to observe Android malware or ransomware, such as Maiorca et al. [44] discusses an Android ransomware approach, which observed Android application's bytecode to determine if an application was a ransomware. This work was further extended by incorporating system API-related information to improve the efficacy of the proposed approach [55]. MaMaDroid [50] utilises machine learning and generates Markov Chain models on the application's call graph from bytecode to detect Android malware. Whereas, Amer and El-Sappagh [6] incorporates deep learning and the abstraction of API or system calls to detect Android malware and ransomware. Their proposed approach further extends to detect unknown malware.

The work proposed in this paper primarily employs dynamic analysis. Our methodology adopted aspects of signature detection, such as the comparison of behavioural patterns, while also being resilient to the aforementioned limitation as we capture and detect the high-level behavioural patterns in real-time. While dynamic analysis has demonstrated increased resilience against common obfuscation techniques, it still remains susceptible to obfuscation. As mentioned previously system call obfuscation has been shown to be possible by the Illusion attack described in [62], but the implementation of such an attack is demonstrably complex compared to static signature and code obfuscation, which can often be more easily achieved [52, 69]. As a consequence, to the best of our knowledge, system call obfuscation, while possible, has not yet been observed in the wild in malware. Furthermore, traditional static signature and code analysis are now understood to be insufficient for detecting newer and more sophisticated malware variants [23]. State-of-the-art tools have transitioned more towards the use of a combination of static and dynamic analysis, such as Android's official anti-malware system, Play Protect, which statically analyses the application upon installation as well as observing the application's behaviour using machine learning algorithms.

2.3.2 Taint analysis

Taint analysis, is a method of observing data flow and tainting sensitive data paths that could potentially be used maliciously. One of the earlier works of taint analysis was TaintDroid [16], which utilised variable-level tracking of native methods within the Dalvik VM interpreter, which contained taint markings in a taint map. These taint markings were propagated through the Android Inter-Process Communication Binder, based on the defined data flow rules on how the application used the tainted data, to the untrusted application's taint map. If the untrusted application made a library call deemed as a taint sink (e.g. *network send*), then the application was marked as malicious.

In contrast, under our method of detection, we observed high-level behavioural patterns at a system call-level with each pattern classified in different levels of severity. This allowed for more precise details regarding an application's behaviour and more flexibility.

Similarly, FlowDroid [9] also adopts the idea of taint analysis. They proposed a static analysis approach, which utilised flow-sensitive taint analysis through the use of Control-Flow Graphs (CFGs), that modelled the life-cycle of Android and call-back methods. FlowDroid's approach offers a unique and precise detection rate, however, due to the fully static approach it shares similar limitations to other static analysis approaches. For example, FlowDroid was only able to capture reflective calls if the arguments were defined as string constants, which was not always the case, as noted in their limitations. Conversely, we adopted a dynamic approach by observing the behaviour of crypto ransomware in real-time, which alleviated the aforementioned limitation.

2.3.3 System call analysis

System calls are often been used for kernel-level malware analysis. Works in [33, 40, 66] apply system call analysis on mobile operating systems, such as Android. This approach is useful because system calls are able to determine the precise operations that occurred during the execution of an application or program, which can help identify malicious activities or behaviours. We further contribute to this area by capturing high-level behavioural patterns exhibited by crypto ransomware.

One drawback, however, with system call monitoring is the large quantity of information generated. Due to background processes—such as `clock_gettime()` that periodically record the system clock time—occurring in parallel with the core operations, the information generated from monitoring an application, is large.

Isohara et al. [33] addressed this issue by filtering out unnecessary system calls. They achieved this by grouping system calls into specific categories and filtered processes unrelated to the application through the use of a process tree. For their detection phase, Isohara et al. [33] created 16 different patterns represented as regular expressions. These regular expressions utilised assistant keywords, which relate to specific strings such as, file paths or commands such as `su`.

The work of Isohara provides a good insight into pattern detection in system call logs using regular expressions. Our proposed approach improves on this notion by introducing a formalised methodology, which converts relevant system calls into tokens and utilises behavioural patterns, represented as a 2-layer token FSMs, for real-time detection of crypto ransomware.

SCSDroid [40] is a thread-grained behavioural pattern detection method on the system call-level leveraging the Longest Common Subsequence (LCS) algorithm to extract potentially malicious patterns from system calls. The Bayes theorem is then utilised with these patterns to determine if an application was a Maliciously Repackaged Application (MRA) or a benign application.

The proposed approach of SCSDroid gives a good perspective of the feasibility of pattern detection used in malware detection. However, as noted in their conclusion, one of the limitations is its inability to detect unknown families that have not been acquired (i.e. trained). In comparison, in our approach, we utilise behavioural patterns represented as Finite State Machines (FSMs) to match common behaviour and behavioural sequences based on a range of ransomware families in a stream of system calls. This allows us to capture a broad range of behavioural patterns in real-time as opposed to family-specific patterns.

One of the more prominent works of system call analysis was CopperDroid [66], which utilised value-based data flow analysis on system call sequences and IPC unmarshalling to reconstruct the high-level behaviour of Android malware. In contrast, our approach showed a higher-level of explainability for malware behaviour through our two layered FSMs, which captured both individual behaviours and behaviours occurring in specific sequences, thus enabling us to observe a more general overview of ransomware behaviour and understand why an application would be marked as malicious.

2.3.4 Real-time malware analysis

A work that focuses on utilising real-time malware detection is DNADroid [22], which adopted a hybrid approach by utilising static and dynamic modules to detect Android ransomware. For DNADroid's 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).

For DNADroid's dynamic module, they utilised a sandbox environment to capture the API call sequences, which were pre-processed by removing common API calls sequences utilised in both benign and malicious applications. After preprocessing, DNADroid utilised Multiple Sequence Alignment (MSA) for aligning multiple extracted strands of API call sequences to acquire the common malicious DNA 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 and 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 is utilised to extract the common DNA subsequences using MSA. These extracted DNA subsequences are compared against other previously extracted DNA 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-min intervals until a malicious sequence match is detected.

DNADroid provides a detection system through the utilisation of Machine Learning and Sequence Alignment techniques. On the contrary, our proposed prototype produces similar effective results for detection rates through the utilisation of 2-layer token FSMs without the reliance on ML models and a sandbox environment.

Semantic aWare andrOid malwaRe Detector (SWORD) [11] creates sequential System Call Graphs using Markov Chains to acquire the typical paths exhibited by malware. Once the typical paths are obtained, statistical analysis is applied using Average Logarithmic Branching Factor (ALBF) to acquire numerical representations of the typical

paths. After applying statistical analysis, supervised machine learning (Random Forest) is used on the training dataset to classify applications as malware or benign.

The methodology proposed by SWORD applied techniques, such as Machine Learning and information theory, to create a runtime detection system. However, one of the main limitations mentioned by SWORD is the cumulative overhead produced by different components, with an average of 13,802.57 s to process all components, and 2401.82 s as the fastest completion time. With our proposed architecture, the processing times are significantly less through the lightweight design of utilising regular expressions and token conversion.

Sun et al. [65] adopts a similar approach, utilising systems calls for real-time malware detection. The first process is initialisation, which generated resources 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.

For their Static Application Analyse process [65], the application's permissions and APIs were extracted with the decompiler tool known as ApkTool [68]. These were utilised in a preparation phase, where all applications statistics were acquired, such as number of malware applications using permissions, and number of benign applications not using permissions.

The Malware Application Identification process, utilised naive Bayes to identify if an application was benign or malicious by sending the application to a server; this extracted the log file and acquired static information (permissions and APIs). Once extracted, the probabilities were calculated using chi-square to determine if an application's requested permissions were related to the application's behaviour. The calculated probabilities were used to determine if an application was benign or malicious. Additionally, their system traversed through the created behavioural graph to identify and reconstruct malicious file operation, network operation, and IPC call behaviours.

As stated in their discussion, one of the limitations of the standard dynamic analysis approaches is the potentially extensive analysis time. This issue is more evident in larger applications with multiple traversable branches. By comparison, our proposed prototype, which observes the application in real-time as it executes, alleviates the lengthy analysis time required.

S^2A^2DE [42] proposed 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 expanded and improved on a preexisting IDS known as SyscallAnomaly, which generated profiles of system calls based on the arguments [37] to identify the normal behaviour of a program.

The methodology of S^2A^2DE clustered 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, S^2A^2DE utilises Markov Chains to observe sequences of clustered system calls enabling them to identify and characterise the program's behaviour.

S^2A^2DE applied the aforementioned methodology in a prototype implementation to show the feasibility of the proposed approach as an Intrusion Detection System (IDS). This prototype was further improved on in later works, which focused on reducing the false positive rates [43]. However, the clustering of system calls generates noticeable performance issues with 700MB of memory usage on the worst-case scenario. Additionally, the clustering and detection times are slower compared to our proposed approach with a worst-case scenario of 12 s for clustering and 12.9 s for detection as noted in their evaluation. Figure 1 summarises the notable features and dataset used in each related work discussed within this section with the addition of our proposed approach (Table 1).

2.4 Behavioural pattern methodology

Figure 1 provides an overview of our behavioural pattern collection. The sandbox environment component is our run-time environment where applications are examined; this environment is described in more detail in Sect. 2.5. The first phase is the *Observation phase* where applications are observed for their behaviour during runtime. After which, we manually derived behavioural patterns using regular expressions based on the benign and malicious behaviours observed during that phase. Section 2.6 provides the detail process of acquiring these patterns. These patterns are then converted into our token representation for pattern matching.

The tokens are used in our second phase, labelled as *Evaluation*. This phase starts with the extraction of the raw system calls logs collected from our sandbox environment, which is then applied with 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 in the final dataset, which contains the formatted system call logs and detected patterns.

The following subsections extensively describe our methodology of collecting and formatting system call logs for detection of malware in more detail. The methodology proposed enables researchers to utilise a streamlined and reproducible approach to safely extract system call logs for effective pattern-based malware detection.

Table 1 Table summarising related work key features and datasets used

Name	Approach	Notable features	Dataset	True positive (%)	F1 score (%)
AndroSimilar [18]	Static	Efficient processing	5993 Google Play samples 3309 Malicious samples 5139 third-party app store samples	80.7	41.7
DroidMoss [70]	Static	Efficient processing	1200 third-party apps (200 from each app store)	89.3	–
Compton et al. [14]	Static	Efficient processing	24,000 Malicious samples	44.9	–
TaintDroid [16]	Dynamic	Behavioural Analysis Real-time Detection	1100 Android Market samples	–	–
FlowDroid [9]	Static	Efficient processing	SecuriBench micro	96.7	90
Isohara et al. [33]	Dynamic	System call sequences	230 applications	–	–
SCSDroid [40]	Dynamic	System call sequences Behavioural analysis Efficient processing	49 Malicious samples 100 Benign samples	96	94.1
CopperDroid [66]	Dynamic	System call sequences Behavioural analysis	Android malware genome dataset Contagio mobile dataset McAfee dataset Total 2900 samples	73	–
DNADroid [22]	Static/Dynamic	Behavioural analysis Real-time detection Crypto ransomware	1928 Malicious samples 2500 Benign samples	98.1	92.1
SWORD [11]	Dynamic	System Call Sequence	1000 Malicious samples 1000 Benign samples	95.8	89.2
Sun et al. [65]	Dynamic	Behavioural analysis	122 Malicious samples 166 Benign samples	85.2	85.9
S^2A^2DE [42]	Dynamic	System call sequences Behavioural analysis Real-time detection Finite state machines	IDEVAL dataset	100	–
This work	Dynamic	System call sequences Behavioural analysis Real-time detection Efficient processing Finite state machines Crypto ransomware	213 Malicious samples 502 Benign samples	100	99.2

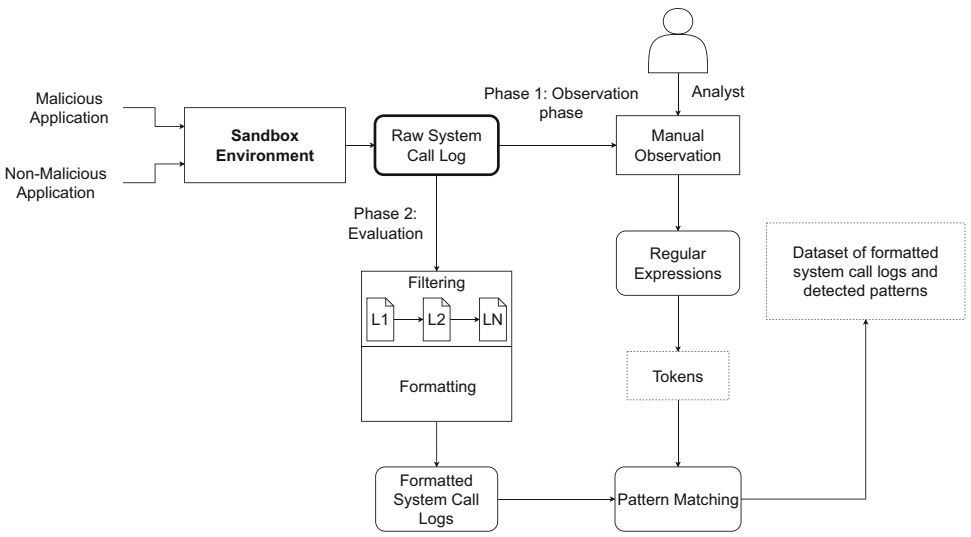


Fig. 1 Methodology process overview

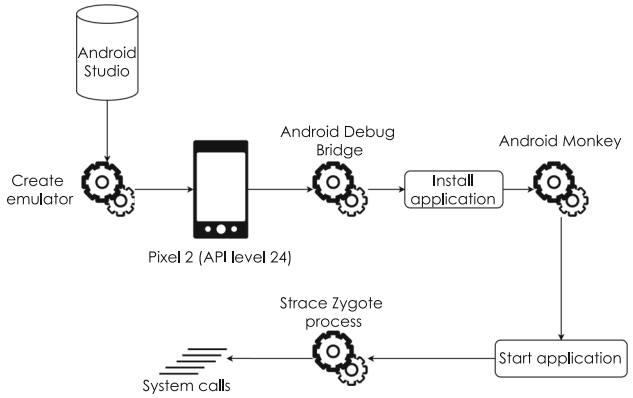


Fig. 2 Overview of system call log collection process

2.5 System call log collection

The first part of our approach is the collection of system call logs. To achieve this, we devised an automatic process of installing applications and tracing system call logs. The environment we used was a Google Pixel 2 emulator running API level 24, created on Android Studio. To automate the process of installing applications and starting applications, we used Android Debug Bridge [25] (ADB) and Android Monkey [27], a program used for generating events on an application. To acquire the system call logs, we ran `strace` [39], a command line tool originally utilised on Linux, to extract and capture the system calls from each application during runtime. The parent process (`Zygote`) was traced to ensure we capture all behaviours produced by the applications. Figure 2 provides an abstract overview of this process.

During the observation phase, we noticed that Android ransomware often prompts for admin privileges. Hence, we automatically accepted the requested permissions for each application. Additionally, to simulate a real-user experience, we used Android Monkey to insert events periodically during the application's runtime. This is described in more detail in Sect. 2.6.

2.6 Acquisition of behavioural patterns

To acquire a set of high-level common behavioural patterns for crypto ransomware, a pilot test was conducted by evaluating 10 crypto ransomware samples from five families obtained from CICAndMal2017 [38] and Koodous [36]. 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. All samples were evaluated from each of these families to acquire our common high-level behaviours. The common high-level behavioural patterns were derived from manual observation of the system call logs. Furthermore, 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 Sect. 4. 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.

2.7 Pattern acquisition and classification

Our method of acquiring the patterns was based on our 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. Our aim is to observe common high-level behavioural patterns specifically focusing on crypto 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 [54] 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 crypto ransomware. 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. Patterns in the *General* category are common benign behaviours that exist in malicious and benign applications with low indication of malicious behaviour.

Note: Suspicious and *General* patterns are not used in our evaluations. These patterns were primarily identified and created to aid future detection systems that utilise common high-level behaviour. Furthermore, crypto 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.

2.7.1 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 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 crypto 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) [58]. The main system call sequences observed, began with `renameat`, followed by an `fstat`, which always occurred before an `unlinkat` operation.

The next malicious pattern from our observations was unlinking of users' files. This behaviour is normally exhibited by crypto ransomware after the file encryption process has occurred [24, 31]. From our analysis, we found consistent occurrences of this pattern in both benign and ransomware samples during our observation phase. However, in the benign samples, the unlinked files were application specific (i.e. within the application's directory) and were unrelated to the user specific directories. There are, however, specific benign applications, such as cache-cleaning application, which can unlink files within the user directories and cause potential false positives. This issue is further discussed in Sect. 4.3.3. The sequence for this pattern began with an `unlinkat` system call followed by the location of the user directory, and the type of file removed.

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 behaviour for crypto 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 main sequence of tokens for this pattern started with an `openat` system call followed by the user directory token, then searched for any files created not matching a regular file extension type.

It is worth noting that it is entirely possible that apps such as of games, etc. produce temporary file types with arbitrary extensions, leading to potential false positives. However, it would be difficult to ascertain the extent of this as the numbers would be dependent on the apps chosen to perform the analysis. Furthermore, fairly and accurately evaluating the use of temporary file occurrence would be challenging given our current methodology of automating the applications with randomly simulated interactions.

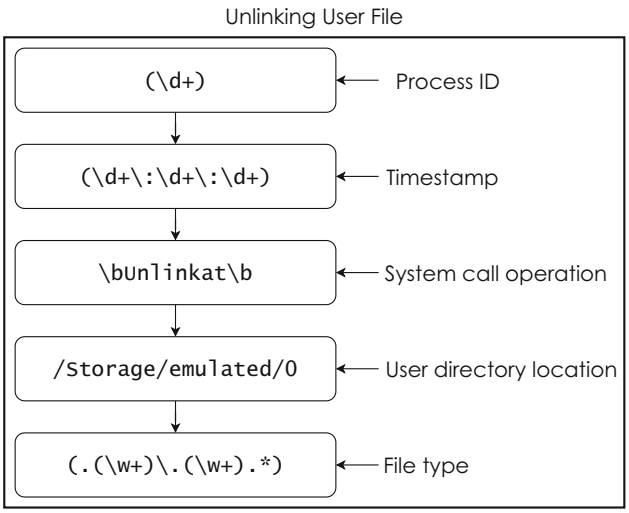


Fig. 3 Abstract view of representing 'Unlinking of user files' malicious pattern using regular expressions

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 a crypto 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 continuously 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 sequence of this pattern begins with an `openat` system call followed by the location of the user directory then three `read` operations.

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 crypto ransomware in our observation phase. The sequence of tokens for this pattern starts with an `openat` system call with the user directory specified, followed by a file created with an unknown file extension and a `write` operation. Figure 3 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.

2.7.2 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 be a non-malicious app connecting to the outside internet. We therefore, classified as suspicious but not malicious. 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 crypto 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 sequence consists of an `openat` system call and a directory name, then a sequence of `getdents64` (system call for getting directory entries), ending with a `close`.

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 sequence of tokens for this pattern comprised an `openat` system call, then any obfuscated file name with no file extension, followed by a `pwrite64` operation with the contents matching any URL address.

The last suspicious pattern was the acquisition of network information via `getaddrinfo`. From our observations, the majority of ransomware applications 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`. This pattern began by matching a `socket` system call followed by the subsequent sequence of system calls: `setsockopt`, `connect`, `fnctl64`, `fstat64`, and concluding with a match for a URL address.

2.7.3 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 or write operation. The last pattern *Generic File Unlink* matches any *unlinkat* system call. During our observation phase, benign applications normally unlinked files, such as *.flock*, *.xml*, *.bak*, or *.db-wal*, which were files unrelated to the user. Hence, *Generic File Unlink* focuses on these specific file extensions.

3 Implementation with streaming system calls

The previous section described an offline methodology for detecting crypto ransomware utilising 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 4 provide an abstract overview of our proposed approach, with the following subsections further elaborating on each module.

3.1 Process token module

To stream the system call data (i.e. capture the system call data in real-time), we used 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 crypto 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()` that periodically record the system clock time and `gettimeofday()`, which 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 Sect. 2.4, to the streaming process. We improved this process by white-listing a smaller subset of systems calls used for crypto ransomware

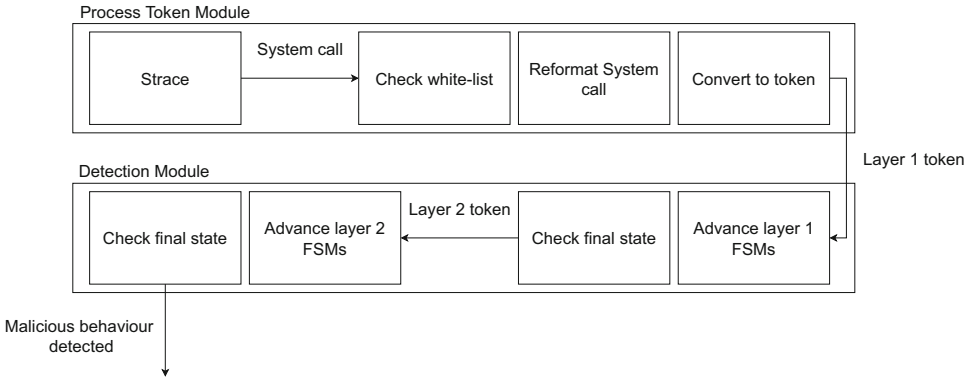


Fig. 4 Block diagram for streaming approach

(e.g. open, write, read). Thus, providing a further reduction in the processing and detection time.

It should be noted that in this work only a small subset of system calls is observed as it enables us to utilise them more efficiently in Finite State Machines (FSM). Through the analysis of system calls, we have observed that the incorporation of additional system calls such as `fstat` and network related calls does not significantly contribute to detection of crypto ransomware at a system call-level. Our emphasis is on a core set of system calls that have shown to be sufficient for identifying malicious behaviour based on initial observations described in Sect. 2.6.

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 2), derived from regular expressions, that matched each system call based on the operation and system call arguments.

Table 2 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

3.2 Detection module

The Detection Module utilises the behavioural patterns previously discussed in Sect. 2.7. 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 crypto 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 Sect. 2.7. 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.

Crypto 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 crypto ransomware. The second layer of FSMs represents the sequential occurrence of behaviours observed in crypto 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

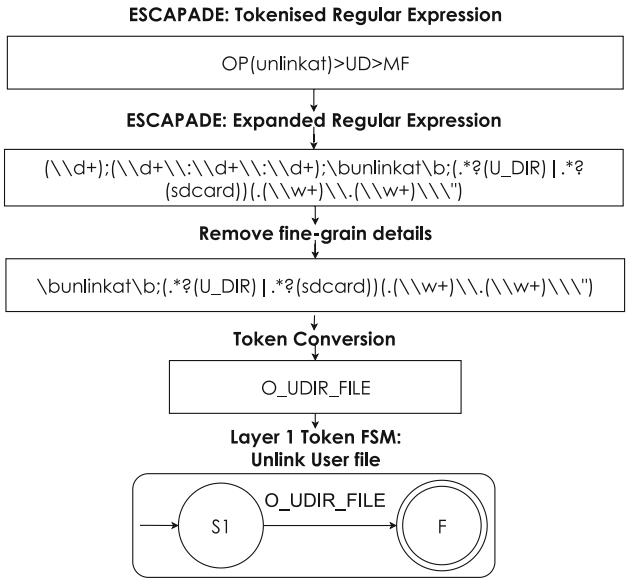


Fig. 5 Transformation of behavioural patterns to layer 1 token FSM

layer 2 FSM is the layer 1 FSM behavioural pattern name (e.g. `Unlink user file`, `General unlink`).

3.2.1 Creation of layer 1 FSMs

Layer 1 FSMs are based on previously discovered crypto ransomware behavioural patterns. However, as mentioned in Sect. 3.2 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 Sect. 3.1. Through this process of generalisation and compaction, we acquired tokenised FSMs. Figure 5 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.

3.2.2 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, crypto ransomware exhibited distinct sequences of behaviours. To acquire the specific sequences of behaviours, we randomly selected six sample 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 crypto ransomware as shown in Table 3. 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 behaviour, which unlinks user files). If one of these sequences is discovered in the 2nd layer of FSMs, the application is considered malicious. Figure 6 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.

4 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. Our process of acquiring the ransomware dataset, the methods used to evaluate our approaches, and the results of our experimentation, which consisted of detected malicious patterns, false positives 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 2048 MB internal storage, 512 MB SDCard storage, and 1536 MB of RAM.

4.1 Dataset acquisition

To acquire the dataset of crypto ransomware samples, we retrieved the hash or package name publicised from established anti-virus vendors, such as Avast [10] and ESET [67], and relevant search tags, such as family name from Koodous [36]; then we manually verified each mali-

Table 3 Sequence of common behaviours exhibited by crypto 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

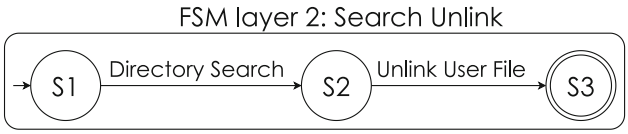


Fig. 6 Layer 2 FSM example for search Unlink

cious application against VirusTotal [59] before downloading the APK from Koodous [36]. As our focus was crypto Android ransomware, it was difficult to acquire a large sample size due to the distinctive category. Nonetheless, we managed to acquire 500 distinct samples. Out of that set, 213 applications exhibited crypto ransomware behaviours. Applications that did not encrypt our files were manually re-evaluated to examine the potential cause of failure. From the re-evaluation, we discovered 18 samples required manual interaction to enable the encryption component. These 18 samples are inclusive of the 213 samples.

From our observations via manual re-evaluation, we noticed several factors that caused the failure of encryption. Some of the samples required a connection to a C&C server that was no longer active. Additionally, some of the applications crashed upon start-up, thus, preventing the malicious code from executing. Furthermore, there were applications that failed to install on the emulator due to issues, such as a missing manifest file.

As part of our contribution, we produced a dataset of system call logs collected from our evaluation of 213 crypto ransomware.[1] We hope this will enable others working on system call-based pattern detection to evaluate their own approaches, or expand and develop new behavioural patterns from their own observations.

Alongside our malicious dataset of crypto ransomware, we acquired 502 benign applications from APKPure [8] to evaluate the efficacy of our approach. Two of these samples were cache cleaning applications. These two special samples were included as these types of applications closely resembled the high-level behaviours of crypto ransomware, specifically the behaviour of removing user files. These two applications were tested separately with manual interaction to ensure we captured the cleaning process.

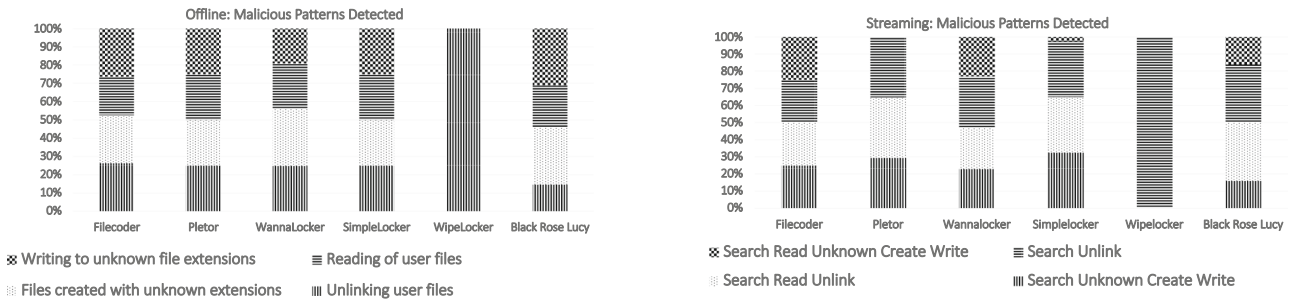
[1] As this dataset consists of active ransomware samples, access can be granted upon request through vimal.kumar@waikato.ac.nz.

4.2 Evaluation method

To evaluate the offline approach, we ran each application for two minutes using our automation script. This automation script installs and starts the applications and utilises Android Monkey [27] 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 crypto ransomware families. Figure 7a, shows the individual malicious patterns detected in the offline approach and Fig. 7b 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 crypto 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 crypto 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 crypto ransomware. The overall results of our evaluation in Fig. 7a, b, provide visible indication of shared common behaviour among crypto ransomware regardless of the family. The only exception is of WipeLocker, which demonstrates a singular behavioural pattern. WipeLocker is



(a) Offline: Malicious behaviour results

(b) Streaming: Sequence of malicious behaviour results

Fig. 7 (a) Offline: malicious behaviour results. (b) Streaming: sequence of malicious behaviour results

known to only remove user files, without encrypting them. Although there have been different classifications for WipeLocker [12], we chose to classify this specific family as a crypto 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 crypto ransomware. Additionally, we have shown the feasibility of our streaming approach for detecting malicious patterns by achieving similar successful results to our offline approach.

4.3 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.

4.3.1 Offline method

Out of the 500 benign applications (excluding the 2 cache cleaning apps), 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*.

- 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 is one of the drawbacks of capturing high-level behaviour. In 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 crypto 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 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 *photographs*, *documents*, and *downloads*.

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.

4.3.2 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 Sect. 3.2.2, 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 crypto ransomware (from different families). Each sample was observed for 5 min in an emulated environment with trap files stored within the Android directory. In this test, 5 out of 6 ransomware encrypted the files within the Android folder except for Wannalocker, which did not encrypt files within the Android folder. These results posed an issue as the exclusion of the `Android` folder limited the scope of our detection process.

To mitigate this issue without compromising on the detection rate, we observed the differences in behaviour between benign and crypto ransomware, specifically the behaviour of directory search. We noticed that with crypto 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.

Table 4 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	

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.

4.3.3 Cache-cleaning applications

As previously detailed in Sect. 2.7.1, specific benign applications, such as cache-cleaning applications could produce behaviours, which can potentially be deemed as malicious if the context is not know (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 Sect. 2.4, 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 4 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 5 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.

Table 5 Summary of benign evaluation with the streaming approach using aforementioned methods

Methodology	True negative	False positive	Sample size
Layer 1 evaluation	489	11	500
Layer 2 evaluation	479	21	500
Restricting user directory	495	5	500
Altered directory search	**498**	**2**	**500**
Incl. cache-cleaning application	498	4	502

Bold is to highlight the most significant result (in terms of highest accuracy) achieved in that specific evaluation.

This was due to the introduction of a layer 2 FSM, which observed sequences 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.

4.4 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 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 s. 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 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). It should be noted that the time to label an application as a ransomware is the average time defined in sequential matches. For example, it will take approximately 0.335 s to determine if a running application exhibited a malicious *Unlink User File* pattern, therefore labelling the application as a ransomware. As can be seen, 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 approach by measuring the number of system calls that can be processed per second (i.e. throughput). In order to do this, we observed 10 random benign samples for 120 s and measured the average CPU time (*usertime* + *systemtime*) 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. $Throughput = Numberofsystemcalls/CPUtime$). 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 s (i.e. $Application\ run - time\ throughput = Number\ of\ system\ calls/120\,s$) 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 s was $13,4020 \pm 96,078$, and the average CPU time for our streaming approach was $17.57s \pm 12.975s$. 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 120 s, 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.

4.5 Discussion

In this section we discuss some of our observations as well as experiences.

As established by now, we were observing the behaviour of crypto ransomware on Android operating system. In order to do this we needed to acquire and then execute the ransomware samples on our VM. The process of acquiring and validating these samples was very time-consuming as each downloaded sample had to be manually checked against VirusTotal [59] to ensure that the malware was of a crypto ransomware family. Crypto ransomware that executes on Android is a subset of all the crypto ransomware which limited the number of samples we could collect. Since we needed the ransomware to actually execute, this further limited the number of samples that we could use, because a large number of samples we collected did not execute. Of the 500 samples

Table 6 Average detection time for individual patterns in seconds

Pattern name	Offline (s)	Single Match (s)	Sequential Match (s)
Unlink user file	0.623 ± 0.0081	0.026 ± 0.0171	0.335 ± 0.0908
Unknown file ext	0.670 ± 0.0076	0.175 ± 0.3068	0.406 ± 0.1437
Read user file	$0.738\text{s} \pm 0.0079$	0.021 ± 0.0112	0.384 ± 0.0897
Write to unknown file ext	0.661 ± 0.0027	0.024 ± 0.0120	0.454 ± 0.0982

we collected, 213 exhibited crypto ransomware behaviour. The remaining 287 samples could not be utilised due to one of the following reasons,

- The application not executing due to missing manifest files
- The application not executing due to incompatible Android versions
- The applications not exhibiting crypto ransomware behaviour
- The application requiring a connection to C2 server

Static and code analysis techniques that only consider the executable file(s) and don't need to execute the ransomware do not generally face these issues. As a result of this limitation, we acknowledge that our models could potentially lead to the issue of an overfitted solution due to the low malicious sample size. However, the samples that we did collect covered the vast majority of crypto ransomware samples on Android devices; although limited, we believe this is close to the extent of the current Android crypto ransomware that we can obtain through publicly accessible and legal means.

As mentioned in Sect. 2.2, we focus specifically on crypto ransomware as it is more prevalent and destructive compared to locker-type ransomware. The system call-level behaviour of locker type ransomware is different from crypto type. We therefore, do not believe it would be feasible to accurately detect locker-type ransomware using the current behavioural implementation without further significant adjustments. While the issue of the limited number of samples in the dataset can be addressed by observing more malware types, as this work focuses on crypto ransomware, the behavioural patterns were specifically designed to only capture crypto ransomware. Different malware types are likely to exhibit stark differences in behavioural patterns at a system call level. Hence, it would not be feasible to achieve a fair comparison in the classification process for discriminating malware and crypto ransomware as potential matches would be coincidental. This issue can be alleviated by further extensive evaluation to understand the underlying behavioural patterns for each malware type. 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. In the current state-of-the-art, the implementation of such a system is a challenging problem due to the requirement of root privileges, and structure of the Android system. However, in future, if the acquisition of system calls were more easily accessible, 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 crypto 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.

While our proposed approach is capable of achieving good detection rates, there are potential improvements that can be implemented to develop a more robust detection system. As previously mentioned in Sects. 2 and 2.3.1 the use of static analysis is also valuable and modern anti-malware systems use a hybrid approach. Our dynamic analysis-based approach can determine whether an application is malicious or benign, however, it has a small but nonzero detection time which

would mean a small amount of data would be encrypted even in the case of a successful detection. Therefore, while we believe that our approach is successful, a complete and practical anti-ransomware system will additionally include a static analysis-based approach to identify known ransomware. The inclusion of static analysis provides reliability. Hence, in future, an interesting avenue to explore is to employ the use of static analysis in our proposed method to develop a more robust and reliable detection approach.

5 Conclusion

In this work, we have described and evaluated a behaviour-based ransomware detection method. We first identified system call-level behavioural patterns for crypto ransomware. We presented our methodology for collecting and identifying behavioural patterns at a system call level. Using this methodology, we were able to discover 12 common high-level behavioural patterns at a system call level. We then evaluated the effectiveness of the behavioural patterns we had identified. This was achieved by evaluating them against a set of crypto ransomware to identify shared commonalities between different families using pattern matching. We have also made our dataset of formatted system calls publicly available. We then improved upon our initial approach to detect crypto ransomware in real-time using a 2-layer token-based finite state machine streaming approach. Finally, we analysed the performance of our approach to demonstrate that our ransomware detection system can run on an Android operating system with acceptable overhead.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Research Data Policy and Data Availability Statements The dataset used in this article is available on request by contacting vimal.kumar@waikato.ac.nz.

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article.

Ethical approval The authors declare that this article does not contain any studies involving human participants or animals.

References

1. Alam, S., Horspool, R., Traore, I., Sogukpinar, I.: A framework for metamorphic malware analysis and real-time detection. Comput. Secur. **48**(C), 212–233 (2015). https://doi.org/10.1016/j.cose.2014.10.011
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019)
3. Al-Rimy, B.A.S., Maarof, M.A., Shaid, S.Z.M.: Ransomware threat success factors, taxonomy, and countermeasures: a survey and research directions. Comput. Secur. **74**, 144–166 (2018). https://doi.org/10.1016/j.cose.2018.01.001. https://www.sciencedirect.com/science/article/pii/S016740481830004X
4. Al-rimy, B.A.S., Maarof, M.A., Shaid, S.Z.M.: Ransomware threat success factors, taxonomy, and countermeasures: a survey and research directions. Comput. Secur. **74**, 144–166 (2018)
5. Alzahrani, N., Alghazzawi, D.: A review on android ransomware detection using deep learning techniques. In: Proceedings of the 11th International Conference on Management of Digital EcoSystems, pp. 330–335. Association for Computing Machinery, New York (2019)
6. Amer, E., El-Sappagh, S.: Robust deep learning early alarm prediction model based on the behavioural smell for android malware. Comput. Secur. **116**, 102670 (2022). https://doi.org/10.1016/j.cose.2022.102670
7. Andronio, N., Zanero, S., Maggi, F.: Heldroid: dissecting and detecting mobile ransomware. In: Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2015, vol. 9404, pp. 382–404. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-319-26362-5_18
8. APKPure. Download APK on Android with Free Online APK Downloader - APKPure. https://apkpure.net/. Accessed 21 Feb 2024
9. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM Sigplan Notices, vol. 49, pp. 259–269. ACM, Association for Computing Machinery, Edinburgh (2014)
10. Avast Blog. https://blog.avast.com/. Accessed 21 Feb 2024
11. Bhandari, S., Panihar, R., Naval, S., Laxmi, V., Zemmari, A., Gaur, M.S.: Sword: semantic aware android malware detector. J. Inf. Secur. Appl. **42**, 46–56 (2018)
12. Chen, J., Wang, C., Zhao, Z., Chen, K., Du, R., Ahn, G.J.: Uncovering the face of Android ransomware: characterization and real-time detection. IEEE Trans. Inf. Forens. Secur. **13**(5), 1286–1300 (2017)
13. Chew, C.J.W., Kumar, V., Patros, P., Malik, R.: Escapade: encryption-type-ransomware: system call based pattern detection. In: Kutyłowski, M., Zhang, J., Chen, C. (eds.) Network and System Security, pp. 388–407. Springer, Cham (2020)
14. Compton, R., Frank, E., Patros, P., Koay, A.: Embedding java classes with code2vec: improvements from variable obfuscation. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, pp. 243–253. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3379597.3387445

15. Bansal, U.: A review on ransomware attack. In: 2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC), pp. 221–226. IEEE Computer Society, Jalandhar (2021). https://doi.org/10.1109/ICSCCC51823.2021.9478148
16. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
17. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android security: a survey of issues, malware penetration, and defenses. IEEE Commun. Surv. Tutor. **17**(2), 998–1022 (2014)
18. Faruki, P., Laxmi, V., Bharmal, A., Gaur, M.S., Ganmoor, V.: AndroSimilar: robust signature for detecting variants of Android malware. J. Inf. Secur. Appl. **22**, 66–80 (2015)
19. Ferdous, J., Mahboubi, A.., Islam, Md.: A review of state-of-the-art malware attack trends and defense mechanisms. IEEE Access 11:121118-121141 (2023). https://doi.org/10.1109/ACCESS.2023.3328351
20. Gandotra, E., Bansal, D., Sofat, S.: Malware analysis and classification: a survey. J. Inf. Secur. **05**, 56–64 (2014). https://doi.org/10.4236/jis.2014.52006
21. Gazet, A.: Comparative analysis of various ransomware virii. J. Comput. Virol. **6**(1), 77–90 (2010)
22. Gharib, A., Ghorbani, A.: Dna-droid: a real-time android ransomware detection framework. In: Yan, Z., Molva, R., Mazurczyk, W., Kantola, R. (eds.) Network and System Security, pp. 184–198. Springer, Cham (2017)
23. Ghillani, D., Gillani, D.H.: A perspective study on malware detection and protection, a review. (2023). https://doi.org/10.22541/au.166308976.63086986/v1. https://www.authorea.com/users/506161/articles/585873-a-perspective-study-on-malware-detection-and-protection-a-review. Accessed 21 Feb 2024
24. Gonzalez, D., Hayajneh, T.: Detection and prevention of crypto-ransomware. In: 2017 IEEE 8th Annual Ubiquitous Computing. Electronics and Mobile Communication Conference (UEMCON), pp. 472–478. IEEE Computer Society, New York (2017)
25. Google: Android Debug Bridge (ADB) (2020). https://developer.android.com/studio/command-line/adb
26. Google: help protect against harmful apps with google play protect (2019). https://support.google.com/googleplay/answer/2812853?hl=en
27. Google: UI/application exerciser monkey (2020). https://developer.android.com/studio/test/monkey
28. Guerra-Manzanares, A., Luckner, M., Bahsi, H.: Android malware concept drift using system calls: detection, characterization and challenges. Expert Syst. Appl. **206**, 117200 (2022). https://doi.org/10.1016/j.eswa.2022.117200
29. Hou, S., Saas, A., Chen, L., Ye, Y.: 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), pp. 104–111 (2016). https://doi.org/10.1109/WIW.2016.040
30. Hou, O.: A Look at Google Bouncer (2012). https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/
31. Hull, G., John, H., Arief, B.: Ransomware deployment methods and analysis: views from a predictive model and human responses. Crime Sci. **8**(1), 1–22 (2019)
32. Iannucci, S., Abdelwahed, S., Montemaggio, A., Hannis, M., Leonard, L., King, J.S., Hamilton, J.A.: A model-integrated approach to designing self-protecting systems. IEEE Trans. Softw. Eng. **46**(12), 1380–1392 (2018)
33. Isohara, T., Takemori, K., Kubota, A.: Kernel-based behavior analysis for android malware detection. In: 2011 7th International Conference on Computational Intelligence and Security, pp. 1011–1015. IEEE Computer Society, Sanya (2011)
34. Kanwal, M., Thakur, S., Lashkari, R.: An app based on static analysis for android ransomware. In: 2017 8th International Conference on Computing. Communication and Networking Technologies (ICCCNT), pp. 1–6. IEEE Computer Society, Delhi (2017)
35. Kok, S., Abdullah, A., Jhanjhi, N., Supramaniam, M.: Ransomware, threat and detection techniques: a review. Int. J. Comput. Sci. Netw. Secur. **19**(2), 136 (2019)
36. Koodous: Malicious dataset (n.d.). https://koodous.com/
37. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) Computer Security—ESORICS 2003, pp. 326–343. Springer, Berlin, Heidelberg (2003)
38. Lashkari, A.H., Kadir, A.A., Taheri, L., Ghorbani, A.: Toward developing a systematic approach to generate benchmark android malware datasets and classification. In: 2018 International Carnahan Conference on Security Technology (ICCST), pp. 1–7. IEEE Computer Society, Montreal, Quebec, Canada (2018)
39. Levin, D.V.: Strace (2020). https://strace.io/
40. Lin, Y.D., Lai, Y.C., Chen, C.H., Tsai, H.C.: Identifying Android malicious repackaged applications by thread-grained system call sequences. Comput. Secur. **39**, 340–350 (2013)
41. Lockheimer, H.: Android and security [Blog post] (2012). https://googlemobile.blogspot.com/2012/02/android-and-security.html
42. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Trans. Dependable Secur. Comput. **7**(4), 381–395 (2008)
43. Maggi, F., Matteucci, M., Zanero, S.: Reducing false positives in anomaly detectors through fuzzy alert aggregation. Inf. Fus. **10**(4), 300–311 (2009)
44. Maiorca, D., Mercaldo, F., Giacinto, G., Visaggio, C.A., Martinelli, F.: R-PackDroid: API package-based characterization and detection of mobile ransomware. In: SAC '17: Proceedings of the Symposium on Applied Computing, pp. 1718–1723. Association for Computing Machinery (2017). https://doi.org/10.1145/3019612.3019793
45. McConnell, D.: The current state of ransomware in today's world and why the future is bleak (2017). https://www.cs.tufts.edu/comp/116/archive/fall2017/dmcconnell.pdf
46. Mehnaz, S., Mudgerikar, A., Bertino, E.: Rwguard: a real-time detection system against cryptographic ransomware. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) Research in Attacks, Intrusions, and Defenses, pp. 114–136. Springer, Cham (2018)
47. Micro, T.: Behind the Android menace: malicious apps—TrendLabs security intelligence blog [Blog Post] (2012). https://blog.trendmicro.com/trendlabs-security-intelligence/infographic-behind-the-android-menace-malicious-apps
48. Mohammad, A.H.: Ransomware evolution, growth and recommendation for detection. Mod. Appl. Sci. **14**(3), 68–74 (2020)
49. Moser, A., Krügel, C., Kirda, E.: Limits of static analysis for malware detection. In: 23d Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430. IEEE Computer Society, Miami Beach (2007)
50. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: detecting Android malware by building Markov chains of behavioral models (extended version). ACM Trans. Priv. Secur. (2019). https://doi.org/10.1145/3313391
51. Oz, H., Aris, A., Levi, A., Uluagac, A.S.: A survey on ransomware: evolution, taxonomy, and defense solutions. ACM Comput. Surv. (CSUR) **54**(11s), 1–37 (2022)
52. Pizzolotto, D., Fellin, R., Ceccato, M.: Oblive: seamless code obfuscation for java programs and android apps. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 629–633. IEEE (2019)

53. Richardson, R., North, M.M.: Ransomware: evolution, mitigation and prevention. Int. Manag. Rev. **13**(1), 10 (2017)
54. Robert Lipovský Lukáš Štefanko, G.B.: Labour party is latest victim of Blackbaud ransomware attack (2016). https://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf
55. Scalas, M., Maiorca, D., Mercaldo, F., Visaggio, C.A., Martinelli, F., Giacinto, G.: On the effectiveness of system api-related information for android ransomware detection. Comput. Secur. **86**, 168–182 (2019). https://doi.org/10.1016/j.cose.2019.06.004. https://www.sciencedirect.com/science/article/pii/S0167404819301178
56. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001, vol. 1, pp. 144–155. IEEE, Oakland (2001). https://doi.org/10.1109/SECPRI.2001.924295
57. Skandylas, C., Khakpour, N.: Design and implementation of self-protecting systems: a formal approach. Fut. Gen. Comput. Syst. **115**, 421–437 (2021)
58. Song, S., Kim, B., Lee, S.: The effective ransomware prevention technique using process monitoring on android platform. Mob. Inf. Syst. **2016**, 1–9 (2016). https://doi.org/10.1155/2016/2946735
59. Sood, G.: Virustotal: R client for the virustotal API. VirusTotal. R package version 0.2.1 (2017)
60. Sophos: the state of ransomware 2020 (2021). https://www.sophos.com/en-us/medialibrary/pdfs/whitepaper/sophos-state-of-ransomware-retail-2021-wp.pdf
61. Sophos: the State of Ransomware 2023 (2023). https://www.sophos.com/en-us/content/state-of-ransomware
62. Srivastava, A., Lanzi, A., Giffin, J., Balzarotti, D.: Operating system interface obfuscation and the revealing of hidden operations. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 214–233. Springer (2011)
63. Statista: Forecast number of mobile devices worldwide from 2020 to 2025 (in billions). Statista (2021). https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/
64. Statistica: global market share held by mobile operating systems from 2009 to 2023, by quarter (2023). https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
65. Sun, S., Fu, X., Ruan, H., Du, X., Luo, B., Guizani, M.: Real-time behavior analysis and identification for android application. IEEE Access **6**, 38041–38051 (2018)
66. Tam, K., Khan, S., Fattori, A., Cavallaro, L.: Copperdroid: automatic reconstruction of android malware behaviors. In: NDSS Symposium 2015, pp. 1–15. NDSS, San Diego (2015). https://doi.org/10.14722/ndss.2015.23145. Annual Network and Distributed System Security Symposium (NDSS) ; Conference date: 08–02–2015 Through 11–02–2015
67. WeLiveSecurity: WeLiveSecurity (2020). https://www.welivesecurity.com/
68. Wiśniewski, R.: Apktool (2021). https://ibotpeaches.github.io/Apktool/
69. Zhang, X., Breitinger, F., Luechinger, E., O'Shaughnessy, S.: Android application forensics: a survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forens. Sci. Int.: Digit. Invest. **39**, 301285 (2021). https://doi.org/10.1016/j.fsidi.2021.301285. https://www.sciencedirect.com/science/article/pii/S2666281721002031
70. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy, CODASPY '12, pp. 317–326. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2133601.2133640