Discrete Event Dynamic Systems (2024) 34:539–571
https://doi.org/10.1007/s10626-024-00406-1

Automatic detection of Android crypto ransomware using supervisor reduction

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

Received: 27 September 2023 / Accepted: 14 October 2024 / Published online: 12 November 2024

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

Keywords Ransomware · Malware detection · System calls · Finite-state automaton · Android

1 Introduction

Ransomware currently poses one of the most serious cyber threats to both consumers and organisations. The global damages from ransomware attacks are expected to exceed $30 billion by 2023 (Poireault 2022). With the increasing frequency and sophistication of ransomware attacks, there is an increasing demand for tools that recognise malicious software automatically in order to protect computers and devices.

Many systems have adopted artificial intelligence or machine learning techniques, which offer automatic and often robust classification of malware. Applications can be analysed *statically* without executing them, for example by inspecting permissions, Application Programming Interfaces (APIs), and bytecode (Li et al. 2017; Bakour et al. 2018), or *dynamically*

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

Robi Malik
robi@waikato.ac.nz

Vimal Kumar
vimal.kumar@waikato.ac.nz

Panos Patros
panos@raygun.com

[1] University of Waikato, Hamilton, New Zealand

[2] Raygun Performance Monitoring, Wellington, New Zealand

by observing the application's behaviour while it is running (Bhandari et al. 2018; Tam et al. 2015). Dynamic methods typically trace and analyse the system calls or other API calls generated by an application in an attempt to identify suspicious behaviour and enforce security policies. Some methods also combine static and dynamic analysis (Gharib and Ghorbani 2017).

The rapidly changing landscape of malicious software and ransomware poses many challenges for the development of anti-malware systems. Advances in static signature-based methods have been countered by malware developers using code obfuscation techniques to avoid detection. This has led to the development of dynamic approaches, for example using system calls, which are largely resilient to simple code obfuscation (Guerra-Manzanares et al. 2022). While system calls offer valuable information about an application's behaviour, the large quantity of information presented by call logs creates a challenging task to design an automated approach for malware detection. Furthermore, future threat actors may find avenues to circumvent such approaches (Anderson et al. 2017; Fang et al. 2019), which means that more diverse solutions will be required.

The main contribution of this paper is an alternative approach to detect crypto ransomware on Android devices, based on *supervisor reduction* (Vaz and Wonham 1986; Su and Wonham 2004). Supervisor reduction is a finite-state machine (FSM) simplification technique used in supervisory control of discrete event systems (Ramadge and Wonham 1989), which can also be used to generalise observed patterns of malicious and benign behaviour. The proposed method is a dynamic approach based on system call traces. System call traces are captured by running known malicious and benign applications. The captured traces are filtered and tokenised, before FSM algorithms including supervisor reduction are used to construct a simplified classification model FSM. The classification model can be implemented to produce a fast and lightweight monitoring system to detect malicious behaviour of running applications. A prototype implementation has been developed to construct classification models, and the paper presents the results of a series of cross validation experiments, which demonstrate that the proposed monitoring system recognises malicious behaviour with reasonable accuracy.

The paper is organised as follows. Section 2 starts with a brief introduction to ransomware and an overview of the methods to detect it from the recent literature, and Sect. 3 provides the necessary background on FSM minimisation and supervisor reduction algorithms. Next, Sect. 4 describes the proposed methodology of capturing system calls and converting them to token traces and finite-state machines. Afterwards, Sect. 5 presents the results of the cross validation experiments to evaluate the approach, including a comparison to related work. Finally, Sect. 6 adds concluding remarks. Further details can be found in the PhD thesis of the first author (Chew 2023).

2 Related work

This section gives a brief introduction to ransomware and its behaviour, followed by an overview of the state-of-the-art malware analysis methodologies found in the recent literature.

2.1 Ransomware

Ransomware is a notorious type of malware that aims to hold the users' device or data at ransom, often for monetary gain. Ransomware uses a variety of methods to gain access to computers and devices, ranging from exploit kits and email phishing links to being incorporated with other types of malware, such as trojans and botnets (Al-rimy et al. 2018; Kumar and Ramlie 2021; Hull et al. 2019; Aurangzeb et al. 2017).

There are two main categories of ransomware: locker-type and crypto (Humayun et al. 2021; Kharaz et al. 2016). *Locker-type* ransomware use techniques that prevent users from accessing most of their system. This is generally achieved by displaying a persistent screen that cannot be closed or terminated unless the ransom is paid (Wyke and Ajjan 2015; Shinde et al. 2016). Locker-type ransomware are considered less dangerous as they are easier for users to circumvent without paying the ransom.

This paper focuses on the second type of ransomware, *crypto* or *encryption-type* ransomware, which are more prevalent and more harmful. Crypto ransomware generally search directories for the users' personal documents and encrypt them. To accelerate the encryption process, the ransomware often target specific file types, such as `.jpg`, `.png`, or `.mp4` files, which are deemed most valuable to the users. After encrypting the files and rendering them inaccessible, the users are prompted to pay the ransom in order to be able to decrypt their files (Ko et al. 2019; Gonzalez and Hayajneh 2017).

2.2 Static analysis

Static analysis is one of the two primary categories of malware analysis, which inspects the files of a software package. This can by done by analysing application metadata and manifest files or parsing the program source code or bytecode (Li et al. 2017; Bakour et al. 2018).

Static analysis methods typically extract *features* from the application files to form a *signature*, which is then used in combination with machine learning algorithms to classify applications as malicious or benign. An example is DroidDet (Zhu et al. 2018), which uses permissions, invocation of system events and their frequency, as well as access to sensitive APIs and URLs as features. RansomProber (Chen et al. 2018) uses the entropy of files, foreground process, and layout information to detect Android ransomware in real-time, whereas RansomDroid (Sharma et al. 2021) reverse engineers applications to acquire static features, and performs forensic analysis and unsupervised machine learning to detect Android ransomware. R-PackDroid (Maiorca et al. 2017), extracts invoke-type instructions within the application's bytecode. Scalas et al. (2019) extend this approach further to observe a more refined set of system API-related information and improve the detectability of malware and benign applications. Similarly, DroidAPIMiner (Aafer et al. 2013) mines the bytecode for specific API calls and parameters. MaMaDroid (Onwuzurike et al. 2019) constructs a probabilistic Markov chain model of the application's call graph from the bytecode, and uses this model as features for machine learning to detect Android malware. DroidNative (Alam et al. 2017) goes even further and examines the native code rather than the bytecode, in an effort to capture malware developed with programming languages that do not use bytecode.

One of the main challenges facing static analysis is the use of obfuscation techniques by malware programmers, such as junk code insertion, encrypting source code, and embedding of malicious payload in native code (Bakour et al. 2018; Islam and Altas 2012), which can only be alleviated to some extent by working at the native code level (Alam et al. 2017). To avoid these issues, the method proposed in this paper does not use static analysis but dynamic analysis.

2.3 Dynamic analysis

The second main category of malware analysis techniques is *dynamic analysis*, which monitors and observes the behaviour of applications while they are running, for example by tracking file system access, network activity, or register or memory contents. This can be achieved by logging or intercepting *system calls* or other API calls, either using debugging

tools such as `strace` (Levin 2020) or by implementing logging interfaces (Tam et al. 2015; Sekar et al. 2000) or instrumenting the application code (Beaucamps et al. 2010; Xu et al. 2012). Dynamic methods are largely resilient to common static obfuscation techniques (Hou et al. 2016; Guerra-Manzanares et al. 2022), because they are based on the behaviour of an application rather than its source code.

While system calls offer valuable information about the behaviour of a running application, typical logs contain an overwhelming amount of detail that makes it difficult to extract relevant information. Therefore it is common to filter the captured system call trace using a *white-list* (Isohara et al. 2011; Chew et al. 2024) to restrict it to a subset of system calls deemed relevant. Amer and El-Sappagh (2022) further abstract API or system calls by only recording the cluster containing them. Isohara et al. (2011) additionally analyse process trees to restrict the trace to the system calls made by processes related to an application in question.

DNA-Droid (Gharib and Ghorbani 2017) identifies and removes repeated subsequences to reduce the length of captured traces further. Similarly, SWORD (Bhandari et al. 2018) removes repetitions by converting system call traces into a probabilistic Markov chain model and extracting short traces based on statistical analysis. CopperDroid (Tam et al. 2015) includes a variety of techniques to extract high-level behaviour from the sequence of system calls observed in dynamic analysis. While largely unnecessary for crypto ransomware, which can be classified based on file-related system calls that are easily recognisable with simple logging, these techniques will likely be crucial when applying the method proposed in this paper to other types of malware.

Some dynamic methods are designed as extensions of static analysis (Gharib and Ghorbani 2017; Amer and El-Sappagh 2022). In these cases, the application is executed in an emulated environment to capture a system call log, which is then used to produce additional features to help with classification. Ferrante et al. (2018) also incorporate static and dynamic elements. The static component monitors the frequency of op-code sequences, while the dynamic component obtains behavioural information, such as memory and CPU utilisation. Machine learning algorithms and a sliding window-based mechanism are used for classification in the dynamic component. Other dynamic methods check for specific patterns in the system calls of a running application in order to recognise suspicious behaviour (Sekar et al. 2000; Chew et al. 2024) or enforce security policies (Xu et al. 2012). In a related effort to improve privacy, TaintDroid (Enck et al. 2014) labels sensitive data, monitors its use, and detects attempts to transmit it over the network.

The dynamic analysis approaches closest to this paper monitor the sequence of system calls generated by an application and use this information to build a *finite-state machine (FSM)* model of malicious behaviour. One of the earlier works adopting an FSM-based approach is by Beaucamps et al. (2010), who construct an FSM model from traces of library calls and match it against a database of malicious behavioural patterns. Similarly, ESCAPADE (Chew 2023; Chew et al. 2024) matches the sequence of system calls from a running application against patterns encoded as multi-layer FSMs. Both methods rely on patterns constructed manually by observing the behaviour of malware. For a more automatic approach, Sekar et al. (2000) develop an FSM model by tracking the locations from where system calls are made within an application, and use it in combination with machine learning. SWORD (Bhandari et al. 2018) uses probabilistic analysis to build a Markov chain model from system call traces, also in combination with machine learning.

This paper follows a similar approach as the aforementioned research, in that it constructs an FSM model from observed system call traces. An important difference is that the FSM is constructed automatically from a collection of known malicious and benign traces, and minimised using FSM algorithms rather than machine learning. The result is a classification

model in the form of an FSM, which can be used directly to monitor a running application while checking for the typical behaviour of crypto ransomware. It is important to propose new solutions to detect malicious software, because malware authors will continue to develop techniques to avoid existing anti-malware systems.

3 Background

This section briefly explains the problem of supervisor reduction and the algorithms to solve it in the tool Waters/Supremica (Åkesson et al. 2006), which the results and experiments in this paper are based on.

3.1 Languages and finite-state machines

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

Definition 1 A (nondeterministic) *finite-state machine (FSM)* is a tuple $A = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ where Σ is a set of *events*, Q is a finite set of *states*, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *transition relation*, and $Q^\circ \subseteq Q$ is the set of *initial states*.

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

$$\mathcal{L}(A) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} \,\} \,. \tag{1}$$

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

3.2 FSM minimisation with "don't care" states

Paull and Unger (1959) describe an FSM minimisation problem where only a part of the language of an FSM is preserved. This minimisation problem can be described using two subsets of an FSM's state set Q, the set of *positive* states $Q^+ \subseteq Q$ and the set of *negative* states $Q^- \subseteq Q$. The sets of traces leading to these states are

$$\mathcal{L}^+(A) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} Q^+ \,\} \,; \tag{2}$$

$$\mathcal{L}^-(A) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} Q^- \,\} \,. \tag{3}$$

In supervisory control, Q^+ may represent a set of states where a particular control action is to be enacted, while the same action is to be prevented at states in Q^-. When classifying

system call traces in this paper, positive traces in $\mathcal{L}^+(A)$ are known *malicious* traces, while traces in $\mathcal{L}^-(A)$ are *benign*.

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

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

$$\mathcal{L}^+(A) \subseteq \mathcal{L}^+(A') \quad \text{and} \quad \mathcal{L}^-(A) \subseteq \mathcal{L}^-(A') \quad \text{and} \quad \mathcal{L}^+(A') \cap \mathcal{L}^-(A') = \emptyset\,. \tag{4}$$

Every trace classified as positive or negative by the original FSM A must be classified in the same way by the reduced FSM A'. This reduction problem is different from standard FSM minimisation problems (Hopcroft et al. 2001) because of the "don't care" traces, which the reduced FSM may classify as positive or negative in ways that allow for a smaller number of states.

Example 1 Consider the FSM A in Fig. 1, which has one positive state $Q^+ = \{0\}$ and one negative state $Q^- = \{3\}$. Its positive and negative languages are $\mathcal{L}^+(A) = \{\alpha\gamma, \beta\alpha\}$ and $\mathcal{L}^-(A) = \{\varepsilon\}$. There also are two "don't care" traces $\alpha, \beta \in \mathcal{L}(A)$. The FSM A' with only two instead of four states achieves the same classification of the positive and negative traces as $\mathcal{L}^+(A') = \{\,\beta^n\alpha\gamma^m \mid n, m \geq 0\,\} \supseteq \mathcal{L}^+(A)$ and $\mathcal{L}^-(A') = \{\,\beta^n \mid n \geq 0\,\} \supseteq \mathcal{L}^-(A)$ while classifying the original "don't care" trace α as positive and β as negative. It is worth noting that A and A' are not language equivalent, $\mathcal{L}(A) \neq \mathcal{L}(A')$, as $\beta\beta \in \mathcal{L}(A')$ while $\beta\beta \notin \mathcal{L}(A)$.

3.3 Supervisor reduction

Supervisor reduction (Vaz and Wonham 1986) is an FSM simplification method used for supervisory control of discrete event systems, which is closely related to the minimisation problem described above. In the context of supervisory control (Ramadge and Wonham 1989), there are two components, namely a *plant* or a system to be controlled and a *supervisor* or controlling agent. Both can be represented as FSMs interacting in lock-step synchronisation in the style of Hoare (1985).

Definition 2 The *synchronous composition* of two FSMs $A_1 = \langle \Sigma, Q_1, \rightarrow_1, Q_1^\circ \rangle$ and $A_2 = \langle \Sigma, Q_2, \rightarrow_2, Q_2^\circ \rangle$ using the same event set Σ is $A_1 \| A_2 = \langle \Sigma_1 \cup \Sigma_2,\ Q_1 \times Q_2,\ Q_1^\circ \times Q_2^\circ,\ \rightarrow \rangle$ where $(x_1, x_2) \xrightarrow{\sigma} (y_1, y_2)$ if and only if $x_1 \xrightarrow{\sigma}_1 y_1$ and $x_2 \xrightarrow{\sigma}_2 y_2$.

In synchronous composition, an event can only be executed if it is enabled by all synchronised components at the same time. In supervisory control, a plant FSM G is synchronised with a supervisor FSM S that restricts the plant behaviour by disabling certain events. The *supervisor reduction problem* can be stated as that of finding a smaller supervisor that results in the same controlled behaviour. That is, given FSMs G and S, supervisor reduction seeks to find another FSM S' with fewer states than S such that $\mathcal{L}(G \parallel S) = \mathcal{L}(G \parallel S')$.

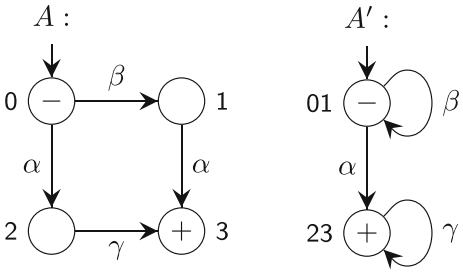


Fig. 1 FSM minimisation with "don't care" states

The above minimisation problem with "don't care" states can be expressed as a supervisor reduction problem as follows. Given a deterministic FSM $A = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ with disjoint positive and negative state sets $Q^+, Q^- \subseteq Q$, a trap state $\perp \notin Q$ is added, and a new event $\mu \notin \Sigma$ is added to represent positive classification. Then the plant and specification are constructed as follows:

$$G = \langle \Sigma \cup \{\mu\},\ Q \cup \{\perp\},\ \rightarrow \cup \{\, (x, \mu, \perp) \mid x \in Q^+ \cup Q^- \,\},\ Q^\circ \rangle\ ; \tag{5}$$

$$S = \langle \Sigma \cup \{\mu\},\ Q \cup \{\perp\},\ \rightarrow \cup \{\, (x, \mu, \perp) \mid x \in Q^+ \,\},\ Q^\circ \rangle\ . \tag{6}$$

The event μ is enabled in all positive and negative states by the plant and only in positive states by the supervisor. Thus the supervisor signals positive classification by enabling μ and negative classification by disabling μ. The enablement and disablement of μ by S is related to the positive and negative languages of A as follows.

$$\mu \text{ is enabled after } s \iff s\mu \in \mathcal{L}(S) \iff s \in \mathcal{L}^+(A)\ ; \tag{7}$$

$$\mu \text{ is disabled after } s \iff s\mu \in \mathcal{L}(G) \setminus \mathcal{L}(S) \iff s \in \mathcal{L}^-(A)\ . \tag{8}$$

When μ is not enabled in the plant, then μ cannot occur in synchronous composition independently of whether the supervisor enables or disables μ; such states are "don't care" states. If supervisor reduction finds another supervisor S' such $\mathcal{L}(G \parallel S) = \mathcal{L}(G \parallel S')$, then S' also solves the above minimisation problem by making a positive classification whenever μ is enabled. More precisely, the conditions Eq. 4 are achieved using an FSM A' obtained from S' where states with μ enabled are declared as positive, other states are declared as negative, and then the event μ is removed.

Example 2 To minimise the FSM A in Fig. 1 using supervisor reduction, the plant G and supervisor S constructed based on Eqs. 5–6 are shown in Fig. 2. The supervisor S is required to disable the event μ in the negative state 0 and enable it in the positive state 3. Supervisor reduction may simplify S by producing S', which achieves exactly the same enablement and

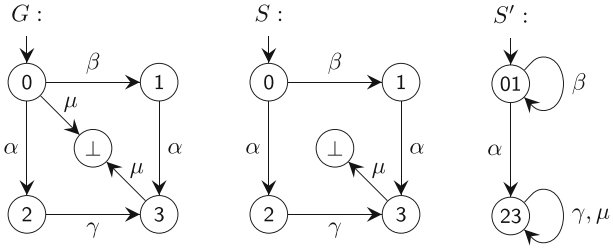


Fig. 2 FSM minimisation via supervisor reduction

disablement of μ as $\mathcal{L}(G \parallel S') = \mathcal{L}(G \parallel S) = \mathcal{L}(S)$. By observing that μ is disabled in state 01 and enabled in state 23 of S', marking these states as negative and positive, and removing the transition $23 \xrightarrow{\mu} 23$, the FSM A' in Fig. 1 is obtained.

In this construction, μ is the only event ever to be disabled by the supervisor, as all other events remain enabled in all states where they are possible according to the plant. Such supervisor reduction problems with only one event to be disabled can be transformed into an equivalent FSM minimisation problem with positive and negative states. Positive states are those where the controlled event must be enabled, and negative states are those where it must be disabled. General supervisor reduction problems with several controlled events can be handled using *supervisor localisation*, where separate supervisors are constructed, each controlling only one event (Cai and Wonham 2016). The problem representation with positive and negative states is more concise and beneficial for algorithms as it enables them to work with only one FSM. For simplicity, this paper will use positive and negative states whenever possible in the following.

3.4 Supervisor reduction algorithm

The usual approach to supervisor reduction is to *merge* states, grouping them into a single state in the reduced supervisor. Positive states can be merged with other positive states or with "don't care" states, and likewise negative states can be merged with other negative states or with "don't care" states; it is also possible to merge two "don't care" states. However, it is not possible to merge a positive state with a negative state, because this leads to violation of the requirement $\mathcal{L}^+(A') \cap \mathcal{L}^-(A') = \emptyset$ in Eq. 4, meaning that the reduced supervisor cannot separate the positive and negative traces accurately.

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

Example 3 Consider states 0 and 1 of FSM A in Fig. 1. There is no immediate problem to merge these states as the negative state 0 can be merged with the "don't care" state 1. Yet, both states have outgoing transitions with event α. If 0 and 1 are merged, then their successors 2 and 3 must also be merged. This is possible here, as 2 is a "don't care" state that can be merged with the positive state 3. As neither 0 and 1 nor 2 and 3 have other common successors, they can be merged. The result is the FSM A' in Fig. 1.

In general, supervisor reduction algorithms seek to establish a *cover* of the state space, i.e., a set $\mathcal{C} = \{C_1, \ldots, C_k\}$ of subsets $C_i \subseteq Q$ of the state set Q that covers the entire state set, i.e., $C_1 \cup \cdots \cup C_k = Q$. The subsets C_i are called *cells*, and each cell becomes a state of the reduced supervisor. For the cover to be feasible for supervisor reduction, none of its cells can contain both a positive and a negative state, and the cover must be *dynamically consistent* (Su and Wonham 2004), i.e., it must respect the transitions as explained above.

This paper uses an algorithm described by Su and Wonham (2004) to compute such a cover and reduce an FSM. This algorithm repeatedly selects two states and attempts to merge them. To do so, it explores all successor pairs to determine whether a positive state needs to be merged with a negative state, collecting additional pairs that need merging in the process. Any

overlapping cells encountered are merged into a single cell, so that the computed cover has no overlapping cells and becomes a *partition*. If the two states selected are found to be mergible, they are merged, otherwise the pair is skipped. In this way, the algorithm systematically tries to merge all state pairs.

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

The supervisor reduction algorithm can be combined with a pre-processing step of *projection* (Malik 2020). The idea is that many supervisory control and classification problems contain more events than necessary to distinguish positive and negative traces. It can be determined with a relatively quick *verifier algorithm* whether or not an event or a set of events can be removed from an FSM while still allowing for the distinction of all positive and negative traces. Using this repeatedly, the method of Malik (2020) searches for a set of events that removes as many transitions as possible. Once an event set is found, the events are removed, the resulting nondeterministic FSM is determinised using subset construction and minimised (Hopcroft et al. 2001), and the result of this simplification is passed to supervisor reduction with the algorithm of Su and Wonham (2004). The subset construction step increases the worst-case time complexity of supervisor reduction to become exponential, which is avoided by Malik (2020) using a threshold: if subset construction generates too many states, it is aborted and the FSM is passed to supervisor reduction without projection. Then the worst-case time complexity of supervisor reduction with projection remains $O(|\Sigma||Q|^4)$.

4 Methodology

Figure 3 gives an overview of the proposed method to detect crypto ransomware at a system call level. In the first step, system call logs are acquired by *running applications* from a dataset of crypto ransomware. In the second step, the system call logs are *filtered and tokenised*, replacing the system calls by a small set of tokens suitable for an FSM model. Next, *loop detection* compresses the token traces to regular expressions by identifying repeated behaviour. Then the *FSM generation* step combines the regular expressions for all traces in a nondeterministic FSM, which is then converted to a minimal deterministic FSM using *subset construction and minimisation*. Finally, *supervisor reduction* minimises the FSM further and produces a classification model, which can then be used to monitor running applications and check for malicious behaviour.

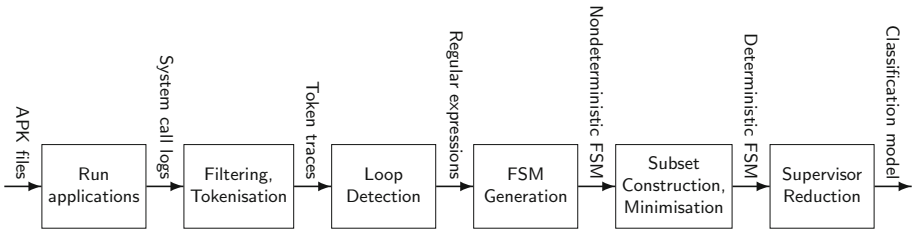


Fig. 3 Steps of classification model generation

The following subsections explain these steps in further detail. First, Subsection 4.1 describes how malicious and benign Android applications are obtained, then Subsection 4.2 describes the process of running these applications to obtain system call logs, Subsection 4.3 describes filtering and tokenisation, Subsection 4.4 describes loop detection, and Subsection 4.5 describes the process of converting regular expressions to FSMs and their minimisation, which covers the three remaining steps. Afterwards, Subsection 4.6 examines the computational complexity of the whole procedure.

4.1 Dataset acquisition

The work in this paper is based on a collection of Android applications from ESCAPADE (Chew et al. 2024), which originally contained 500 malicious ransomware applications from ten different families. However, after further analysis it was discovered that several ransomware applications did not encrypt the users' files due to unsuccessful installation or simply being unable to run in the emulation environment. As a result, the ESCAPADE dataset only contains 213 malicious applications.

The malicious applications were acquired from Koodous (2022), which contains a repository of packages often used for malware analysis. Their hashes or package names were validated on VirusTotal (Sood 2017) to identify and classify the ransomware by family. As a result, the malicious applications are associated with six distinct families as shown in Fig. 4. These are families of known malware that first appeared during 2014–2020, and all were still circulating in 2020. The applications in each family can be traced to the same malware core, albeit with slight variations and different packaging (e.g., different application type, or name). Except for Wipelocker, the malicious applications exhibit typical ransomware behaviour of encrypting and replacing files. Wipelocker only deletes the user's files with no evidence of encryption occurring (Chew et al. 2024). It is nevertheless included in the dataset because the deletion of files is still malicious and the behaviour is similar to that of crypto ransomware.

To counterbalance the malicious applications, 502 benign applications were obtained from APKPure (n.d.), which include a variety of types, such as entertainment, utility, and productivity applications. Two of the benign applications are cache cleaning applications, which are included because their behaviour resembles that of crypto ransomware, specifically the systematic deletion of files.

4.2 System call log collection

Having obtained a set of Android applications, the next step is to execute them and observe the generated system calls. Using Android Studio (Google 2021), an emulated environment that simulates a smartphone running Android 7.0 (API level 24) is created. The file system of the emulated environment is set up with 105 trap files of types `.jpg`, `.png`, `.mp3`, `.mp4`, `.txt`, `.cfg`, and `.xml`, which are commonly encountered on real user smartphones (Du et al. 2021).

For automating the trace collection process, Android Debug Bridge (Google 2020a) is used to send commands to the emulator. Each application is installed from its Android Package Kit (APK) file, under elevated privileges, and the application is run for three minutes. Additionally, Android Monkey (Google 2020b) is used to simulate user interaction. As each application is running, all executed system calls are captured using `strace` (Levin 2020). The process observed is the `Zygote` process, which is the main parent process on Android. This ensures that system calls from all running processes are captured and makes it possible to

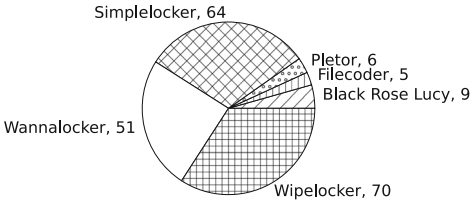


Fig. 4 Distribution of ransomware based on family

detect malware that uses multiple processes to avoid behaviour-based detection (Bidoki et al. 2017). The raw system call data from `strace` is saved to a log file for further processing.

After executing a malicious application, it is furthermore checked whether any of the trap files have been modified or deleted, in order to make sure that the captured trace indeed contains malicious behaviour. As a result of this check, eight samples of Black Rose Lucy were found not to modify any trap file. They were not considered as malicious and removed from the dataset, reducing the 213 malicious applications of Chew et al. (2024) to 205 malicious applications as shown in Fig. 4. These applications are confirmed to modify or delete trap files.

4.3 System call filtering and tokenisation

Applications typically generate a large number of system calls, not all of which are helpful to classify the behaviour as malicious or benign. An example is the system function `clock_gettime()`, which is frequently used to read the system clock time by both malicious and benign applications. Such irrelevant system calls increase the volume of the system call data significantly and make it difficult to identify malicious behaviour. To combat this issue, a white-listing approach (Isohara et al. 2011; Chew et al. 2024) is used to filter out some system calls.

Furthermore, after filtering system calls based on the white-list, the raw data generated by `strace` still contains a lot of detail about each system call including timestamps, parameters, and return values, which needs to be abstracted from to build a useful FSM classification model. Therefore, the system calls that survive filtering are replaced by *tokens* selected from a small set, again similar to the approaches of Chew et al. (2024) and Isohara et al. (2011).

Table 1 shows six different ways considered in this paper to convert system calls to tokens. The defining behaviour of crypto ransomware is the encryption process (Chew et al. 2024; Kok et al. 2019; Lemmou et al. 2021), and therefore the selected tokens are related to the specific system calls used when the ransomware searches for and encrypts files. The most relevant system call is `openat()`, which is needed to open directories in order to search them, to

Table 1 Six possibilities to tokenise system calls

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

open files to read their contents, and to create files to write encrypted data. Additionally, the `renameat()` system call is represented by a token, because certain samples of Wannalocker exhibit behaviour where the encrypted file is renamed (Chew et al. 2024). Lastly, crypto ransomware often removes the original file after the encryption process or as part of the extortion process, and therefore the `unlinkat()` system call is represented by another token.

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

The six token sets in Table 1 are defined to examine the effects of different levels of abstraction in the token traces. Set I is the coarsest set, which represents all `openat()` system calls with the same token O and makes no distinction based on the flags, while Set II distinguishes between opening directories and opening files for reading or writing, without separating the different modes of writing. Some token sets filter out certain system calls, which is shown by a dash in the table to indicate that no token token is generated. Most of the evaluation presented in the following is based on Set IV, which is designed based on the understanding of ransomware behaviour in the hope of preserving a reasonable level of detail while keeping the token traces small. The other tokenisations are evaluated in Sect. 5.4 below.

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

4.4 Loop detection

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

A sliding window technique is used to combine sequences of repeated tokens into loops. The process begins by taking a token trace and a threshold $k \geq 1$ as input. The threshold signifies the number of repeated tokens required before a token sequence is replaced by a loop. For example, with threshold 2, a sequence OO is replaced by O^{2+}, which indicates two or more occurrences of the token O. In general $E^{k+} = E^k E^*$ for $k \geq 1$ and regular expressions E, and at threshold k any sequence α^j with $j \geq k$ and $\alpha \in \Sigma$ is replaced by α^{k+}.

Example 4 Assuming $\Sigma = \{\mathsf{O}, \mathsf{R}\}$, the token trace OORRR is replaced by $\mathsf{O}^{2+}\mathsf{R}^{2+}$ at threshold 2 and by OOR^{3+} at threshold 3.

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

Example 5 Given a threshold value of 2, the token trace OOOROOROOOOR is replaced by $\mathsf{O}^{2+}\mathsf{RO}^{2+}\mathsf{RO}^{2+}\mathsf{R}$ in the first iteration, and this is replaced by $(\mathsf{O}^{2+}\mathsf{R})^{2+}$ in the second iteration.

Both the threshold and the number of iterations are adjustable parameters of the loop detection process. Section 5.3 contains the results of an experiment to examine how changing these parameters affects the accuracy of ransomware detection.

4.5 Classification model generation

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

$$\mathcal{L}^+(A) = M\Sigma^*; \tag{9}$$

$$\mathcal{L}^-(A) = \mathrm{Pre}(B) \setminus M\Sigma^*. \tag{10}$$

Every malicious trace and every possible continuation of a malicious trace is classified as positive (Eq. 9). This represents the idea that, once malicious behaviour in M has occurred, damage has been done and cannot be reverted by further action. Furthermore, all prefixes of benign traces are classified as negative, so traces that can be continued to an observed benign trace are not considered as malicious (Eq. 10). To ensure the requirement from Sect. 3.2 that the positive and negative languages are disjoint, traces that occur both in the malicious and benign sets are classified only as positive by removing the continuations of malicious traces from the negative language (Eq. 10). That is, ambiguous traces encountered as both malicious and benign lead to false positives rather than false negatives. This situation did not arise during the experiments carried out for this paper.

To construct the FSM A, first the regular expressions representing the reduced token traces according to Sect. 4.4 are converted to nondeterministic FSMs using a standard algorithm (Hopcroft et al. 2001). These FSMs are then combined in a single nondeterministic FSM with several initial states that includes the positive and negative languages (Eqs. 9 and 10). This nondeterministic FSM is converted to a minimal deterministic FSM with the same positive and negative languages using subset construction and Hopcroft's minimisation algorithm (Hopcroft et al. 2001). If the deterministic FSM from subset construction contains any states that are marked as both positive and negative, then the negative marking is removed from these states to ensure Eq. 10.

Finally, the minimal deterministic FSM is converted to a supervisor reduction problem as explained in Sect. 3.3, and then a supervisor reduction algorithm as described in Sect. 3.4 is used to obtain the *classification model*. As explained in Sect. 3.3, the supervisor reduction problem includes an additional event μ to label states with positive classification.

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

Example 6 Consider the token set $\Sigma = \{\mathsf{OD},\ \mathsf{OCWF},\ \mathsf{ORF},\ \mathsf{U}\}$ and the benign and malicious languages

$$B = \{\mathsf{ORF\ ORF\ OCWF},\ \mathsf{ORF\ OCWF},\ \mathsf{OD\ ORF\ ORF\ ORF}\}\ ; \tag{11}$$

$$M = \{\mathsf{OD\ ORF\ OCWF},\ \mathsf{OD\ U\ U}\}\ . \tag{12}$$

Figure 5 shows the nondeterministic FSM A constructed from these traces (without loop detection). The three benign traces appear on the left with all their states negative. Thus all prefixes of benign traces lead to negative states. The two malicious traces on the right have positive end states, and the selfloops labelled Σ indicate transitions with all events in Σ to represent that continuations of malicious traces remain malicious.

The nondeterministic FSM A is next replaced by a deterministic FSM $\det(A)$ using the subset construction algorithm (Hopcroft et al. 2001). This results in a so-called *observer* FSM whose states represent sets of states where the original FSM A may be. For example, the initial state of $\det(A)$ combines the five initial states of A, and its successor on event ORF represents the second states of the first two benign traces as the observation of ORF means that the system may follow either of these traces. The combined states are marked as negative or positive if at least one of their constituent states has this marking. Without loop detection, subset construction merely combines the traces into a tree structure as it appears in Fig. 5.

The FSM resulting from subset construction may contain redundancy. The three end states of the benign traces all have negative markings and no outgoing transitions, and likewise the two positive states at the end of the malicious traces represent the same behaviour. Such states are combined into a single state using Hopcroft's minimisation algorithm (Hopcroft et al. 2001). The result $\mathrm{mindet}(A)$ is the unique FSM with the fewest states that has the same positive and negative languages as A.

Finally, Fig. 5 shows a classification model A' resulting from the application of supervisor reduction to $\mathrm{mindet}(A)$, in this case obtained by merging the states grouped with the dashed lines in the diagram. The benign trace ORF ORF OCWF takes A' to a negative state, indicating classification as benign, while the malicious trace OD ORF OCWF takes A' to its positive state and is classified as malicious. The trace OD U, which takes $\mathrm{mindet}(A)$ to a "don't care" state, takes A' to its positive state and is classified as malicious. The trace U U is not accepted by A' from the first step onward and is considered as benign.

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

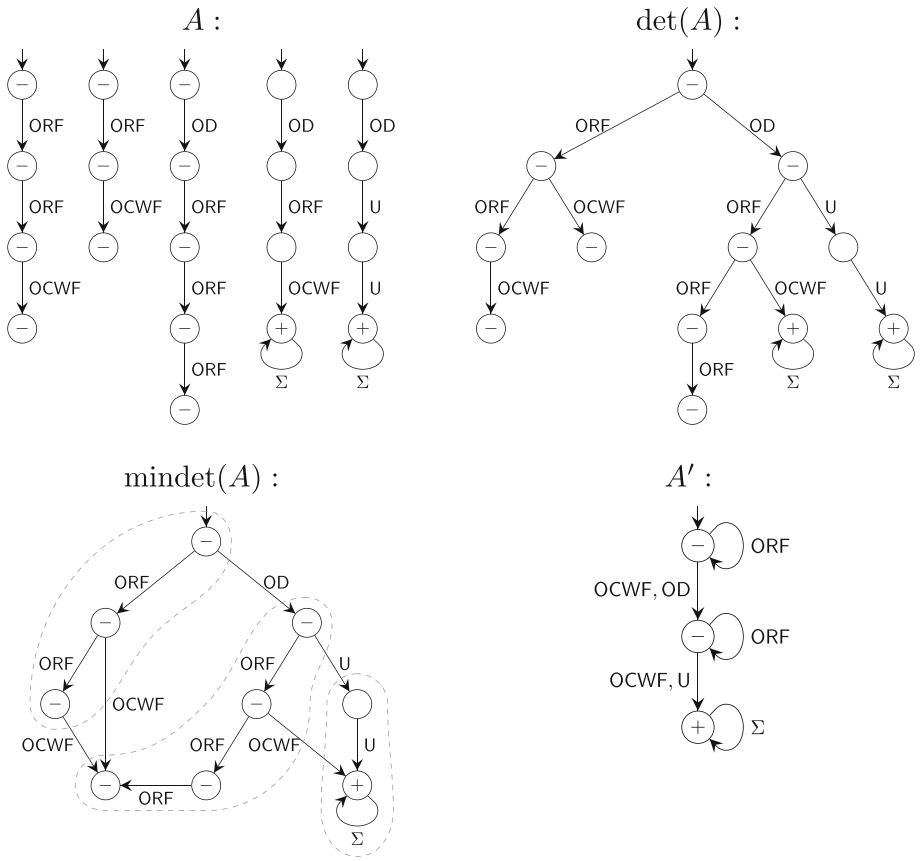


Fig. 5 Constructing a classification model. The nondeterministic FSM A is constructed from the malicious and benign traces, converted to a deterministic FSM $\det(A)$, reduced to the smallest language-equivalent deterministic FSM $\text{mindet}(A)$, and further reduced by supervisor reduction to obtain the classification model A'

4.6 Computational complexity

Considering the steps of classification model generation in Fig. 3, the steps with the highest computational complexity are *Subset Construction and Minimisation* and *Supervisor Reduction*.

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

The bound on the number of states increases with the number of loop detection iterations to a limit of $O(N^k)$ where N is the length of the longest trace and k is the number of traces, which is exponential in the number of traces. However, this limit can only be reached with a high number of loop detection iterations. Exponential blow-up at this stage was not observed

in our experiments with up to four loop detection iterations. The exponential worst-case can be avoided by monitoring the number of states generated during subset construction and restarting with a lower number of loop detection iterations when the number of states exceeds a threshold. Then the time complexity remains linear in the size of the input, the threshold, and the number of loop detection iterations.

The next time consuming step is *Supervisor Reduction*, which has a worst-case time complexity of $O(|\Sigma||Q|^4)$ as explained in Sect. 4.6. The worst-case for the number of states is $|Q| = N^k$, which means exponential time complexity in the number k of traces. If the number of states can be kept linear in the size $n = kN$ of the input, then the complexity of the supervisor reduction step is polynomial in the size of the input, $O(n^4)$. As supervisor reduction is the step with the highest complexity in Fig. 3, this becomes the theoretical worst-case time complexity for the entire classification model generation process.

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

5 Evaluation

This section evaluates the performance of the classification models obtained using the method described in Sect. 4 in terms of their size and accuracy. First, Subsection 5.1 describes the evaluation methodology, and the following Subsections 5.2–5.4 present experiments to determine how different parameters of the classification model generation process affect the results. Afterwards, Subsection 5.5 presents an experiment that attempts to detect malware from an unknown family, and Subsection 5.6 compares the proposed approach to related work.

5.1 Cross validation approach

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

After classifying the testing data for all ten rounds of cross validation, the results are accumulated to determine the numbers of true positives, false positives, true negatives, and false negatives. A *true positive* occurs if the classification model correctly identifies a trace from a malicious application as malicious. Otherwise, if a malicious trace is incorrectly classified as benign, it is called a *false negative*. Similarly, a benign trace is called a *true negative* if it is correctly classified as benign and a *false positive* if it is incorrectly classified as malicious. After the numbers of true positives etc. are determined, it is possible to calculate

their *rates*:

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \qquad \text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{13}$$

$$\text{TNR} = \frac{\text{True Negatives}}{\text{False Positives} + \text{True Negatives}} \qquad \text{FNR} = \frac{\text{False Negatives}}{\text{True Positives} + \text{False Negatives}} \tag{14}$$

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

$$\text{F1} = \frac{2 \times \text{True Positives}}{2 \times \text{True Positives} + \text{False Positives} + \text{False Negatives}} \tag{15}$$

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

5.2 Evaluation of supervisor reduction options

In the first experiment, it is examined how different supervisor reduction settings affect the generated classification model. As mentioned in Sect. 3.4, the supervisor reduction algorithm (Su and Wonham 2004) is sensitive to the order in which it attempts to merge state pairs. Therefore the tool Waters/Supremica (Åkesson et al. 2006), which was used for all experiments, supports several options to control this algorithm.

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

Having ordered the states of the FSM being reduced, there are different possible orders in which to consider state pairs for merging. The simplest strategy is the *Lexicographic* pair ordering, where the first state in the state ordering is first paired with the remaining states in the state ordering. After that, the second state is paired with the states that appear after it in the state ordering, etc. Alternatively, the *Diagonal* pair ordering strategy starts by pairing the first and second states, then considers pairings of the third state with the states before it in the state ordering, before moving on to the fourth state, etc. There are two variants, *Diagonal1* and *Diagonal2* where the states each state is paired with are considered in ascending or descending order. The main difference is that lexicographic pair ordering starts pairing the

first state with all others, while diagonal pair orderings give preference to pairs of states that are both close to the start.

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

For the first experiment, the system call logs in the training data are tokenised based on Set IV in Table 1 and subjected to one iteration of loop detection with a threshold value of eight, and the resulting regular expressions are used to produce a nondeterministic and then a minimal deterministic FSM. Then classification models are generated by supervisor reduction with all combinations of the options mentioned above, and used to classify the traces in the testing data.

Table 2 shows the results of this evaluation. For each combination of options, the table shows the true positive and associated rates according to Eqs. 13 and 14 as well as the F1 score according to Eq. 15 from ten-fold cross validation. The table also shows the number of states of the generated classification model FSMs, averaged over ten rounds of cross validation, as an absolute number and as a percentage of the number of states before supervisor reduction. The last column gives the time taken by the supervisor reduction algorithm to compute them, again averaged over the ten rounds. The experiments were carried out on a PC running Ubuntu 18.04.6 LTS with a 3.8 GHz Intel Core i5-7600K CPU and 16 GiB of RAM.

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

Supervisor reduction substantially reduces the number of states of its input FSM, which has 128400 states. With RBFS state ordering, Lexicographic or Diagonal1 pair orderings, and with projection disabled, more than 99.9% of the states are removed. To put these numbers in

Table 2 Classification models obtained with different supervisor reduction options

Supervisor reduction options			Accuracy					Result		
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Red%	Time
DFS	Lexicographic	Off	0.444	0.556	0.984	0.016	0.599	2078	1.62%	72.6 s
DFS	Diagonal1	Off	0.585	0.415	0.954	0.046	0.690	742	0.58%	1493.0 s
DFS	Diagonal2	Off	0.620	0.380	0.930	0.070	0.692	581	0.45%	528.2 s
DFS	Lexicographic	On	0.649	0.351	0.972	0.028	0.756	1281	1.00%	32.7 s
DFS	Diagonal1	On	0.673	0.327	0.902	0.098	0.704	909	0.71%	175.0 s
DFS	Diagonal2	On	0.702	0.298	0.896	0.104	0.718	1192	0.93%	100.2 s
RBFS	Lexicographic	Off	0.951	0.049	0.948	0.052	0.915	91	0.07%	502.5 s
RBFS	Diagonal1	Off	0.951	0.049	0.946	0.054	0.913	91	0.07%	535.0 s
RBFS	Diagonal2	Off	0.312	0.688	0.998	0.002	0.474	13293	10.35%	665.3 s
RBFS	Lexicographic	On	0.917	0.083	0.938	0.062	0.887	107	0.08%	52.8 s
RBFS	Diagonal1	On	0.922	0.078	0.942	0.058	0.894	114	0.09%	54.8 s
RBFS	Diagonal2	On	0.620	0.380	0.994	0.006	0.758	7665	5.97%	84.9 s

context, Table 3 shows the state numbers and reduction rates of the earlier model generation steps. Loop detection reduces the number of states by about half compared to an FSM constructed without loop detection, and FSM determinisation and minimisation via subset construction removes slightly less than one quarter of states from the result. The last two lines in Table 3 show the state numbers of the worst and best performing supervisor reduction options according to Table 2. When measured as a percentage, supervisor reduction achieves better reduction than all other steps.

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

As explained in Sect. 3.4, projection searches for events that are irrelevant for the classification task and removes them from the FSM prior to supervisor reduction. Table 2 shows that projection always changes the generated classification model. By closer inspection of the classification models, it is found that the tokens ORF and OAWF of Set IV are removed in all ten rounds of cross validation, and RN is removed in nine of the ten rounds. The token ORF indicates opening a file for reading and regularly appears in both benign and malicious traces. The token OAWF, which indicates opening a file for appending, is used less frequently, and RN, which indicates renaming of a file, is only observed in some Wannalocker traces. The removal of these tokens by projection suggests that the corresponding system calls are not needed to distinguish malicious and benign traces, and ransomware can be detected by only considering the pattern of directory accesses, files opened for writing, and file deletions. This makes sense given that the malicious activity of the applications considered involves the creation of encrypted files and the deletion of user files, usually in combination with directory searches.

While the use of projection often reduces the runtime of supervisor reduction significantly, it reduces the number of states only once and never leads to better accuracy in this experiment. Particularly with events that are used only occasionally, although accurate classification may be possible without them, it may require a more complicated classification model. In this experiment, supervisor reduction seems to be able to find better classification models that use all events, although it takes longer to do so because the supervisor reduction algorithm has to process more states.

The supervisor reduction times in Table 2 are all below 25 min, with the better performing supervisor reduction options finishing in less than ten minutes. Notwithstanding the theoretical complexity analysis, the by far most time-consuming step when generating a classification

Table 3 State space reduction by different steps of model generation

Step	States before	States after	Red%
Loop detection	351979	167512	47.59%
Subset construction and minimisation	167512	128400	76.65%
Supervisor reduction (RBFS, Diagonal2, On)	128400	13293	10.35%
Supervisor reduction (RBFS, Lexicographic, Off)	128400	91	0.07%

model is running the applications. It takes 36 h to run the 707 applications for three minutes each, although the process can be parallelised. System call logs can also be saved and reused, so when a dataset is extended with new applications, only the new applications need to be run. Still, the supervisor reduction times are comparatively small, even if several classification models are constructed to choose a smallest one. The times taken for the other steps in Fig. 3 are negligible in comparison.

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

5.3 Evaluation of loop detection

A series of experiments was carried out to evaluate the efficacy of the loop detection process described in Sect. 4.4. Loop detection has two parameters, the *number of iterations* which determines the length of repeated sequences and the potential nesting level of detected loops, and the *threshold* which determines the minimum number of repetitions before a sequence of tokens is replaced by a loop. In the experiment, the ten-fold cross validation process described above is repeated after performing 1–4 iterations of loop detection with threshold values in the range 2–12. All system call logs are tokenised using Set IV, and the supervisor reduction options are limited to the combinations found most effective in Sect. 5.2, namely reversed breadth-first search (RBFS) state ordering and Lexicographic or Diagonal1 pair ordering.

Figure 6 gives an overview of the accuracy and size of the classification models produced in this experiment. Each circle indicates the results for one particular combination of loop detection and supervisor reduction options. The position of the circle indicates the false positive and false negative rates measured by ten-fold cross validation, and the size of the circle represents the average number of states of the classification models in the ten rounds of cross validation. The results can also be compared to classification models computed with loop detection disabled, which are shown in Table 4 with the same details as in Table 2.

All the classification models in this experiment have F1 scores between 86.5% and 93.8%. Figure 6 shows that the majority of the models computed with loop detection are more accurate than those computed without loop detection, which have F1 scores between 88.9% and 89.6%. This suggests that loop detection can help to find more accurate classification models.

Figure 6 suggests that more accurate classification models have fewer states, but with the poorly performing supervisor reduction options removed, this trend is less pronounced than previously observed in Table 2. The numbers of states obtained with the different loop detection options range from 78 to 930. The smallest model with 78 states, obtained with one loop detection iteration and a threshold of five, has an F1 score of 88.8% and a false negative rate of 8.8%. The most accurate model with an F1 score of 93.8% and a false negative rate of 4.4% has 91 states; it is obtained with three or four iterations and a threshold of eight.

Furthermore, there is a difference in accuracy between models obtained with projection enabled and disabled, which was already observed in Table 2. The models computed with projection and with one and particularly two loop detection iterations are among the least accurate. On the other hand, models computed with two loop detection iterations and without projection form a group of fairly accurate models. A higher number of iterations leads to a more irregular spread, including the most accurate models as well as several large and poorly performing models both with projection enabled or disabled.

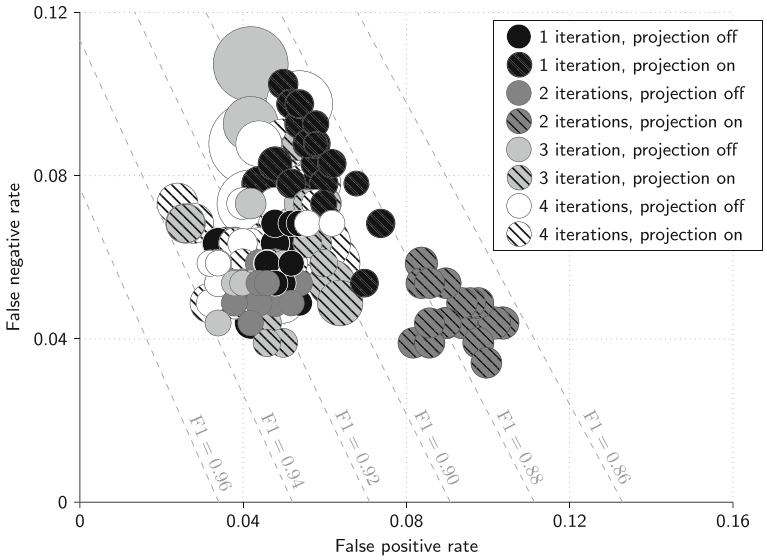


Fig. 6 Classification model accuracy and size depending on loop detection parameters and projection

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

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

Table 4 Classification models obtained without loop detection

Supervisor reduction options			Accuracy					Result		
States	Pairs	Projection	TPR	FNR	TNR	FPR	F1	States	Red%	Time
RBFS	Lexicographic	Off	0.912	0.088	0.946	0.054	0.893	176	0.063%	2407.4 s
RBFS	Diagonal1	Off	0.912	0.088	0.946	0.054	0.893	176	0.063%	2461.3 s
RBFS	Lexicographic	On	0.922	0.078	0.944	0.056	0.896	77	0.028%	74.1 s
RBFS	Diagonal1	On	0.917	0.083	0.940	0.060	0.889	75	0.027%	78.3 s

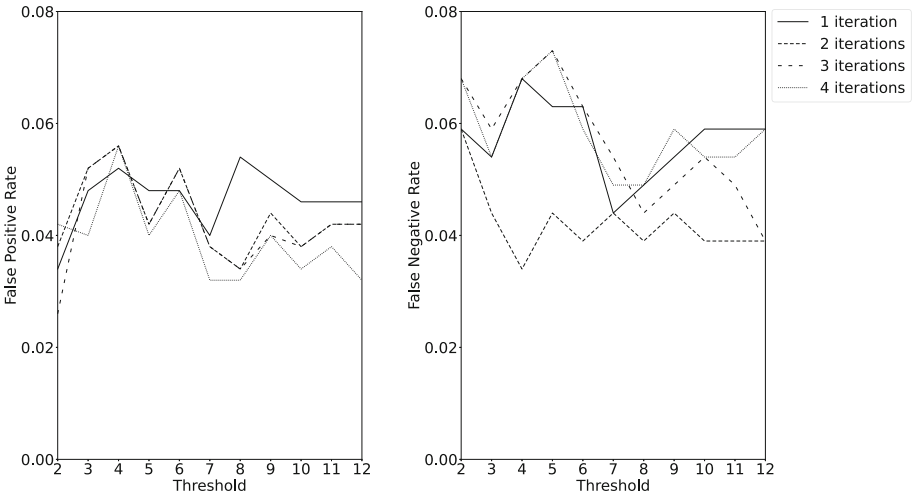


Fig. 7 Lowest false positive and false negative rates depending on loop detection parameters

Figure 8 shows the supervisor reduction times measured in this experiment. The graphs show average runtimes over four combinations of supervisor reduction options and over ten rounds of cross validation, depending on the two loop detection parameters. The diagram shows that the runtime increases steadily with increasing threshold towards the average of 1255.3 s without loop detection, when using one loop detection iteration. Yet, this increase is not observed with more iterations. The runtime of supervisor reduction is mainly determined by the size of the deterministic FSMs, whose number of states appears to remain limited when two or more loop detection iterations are used. A secondary factor affecting the runtime is the structure of the FSMs, which become more complex as each iteration produces longer and more deeply nested loops. In this experiment, two iterations seem to result in the fastest runtime.

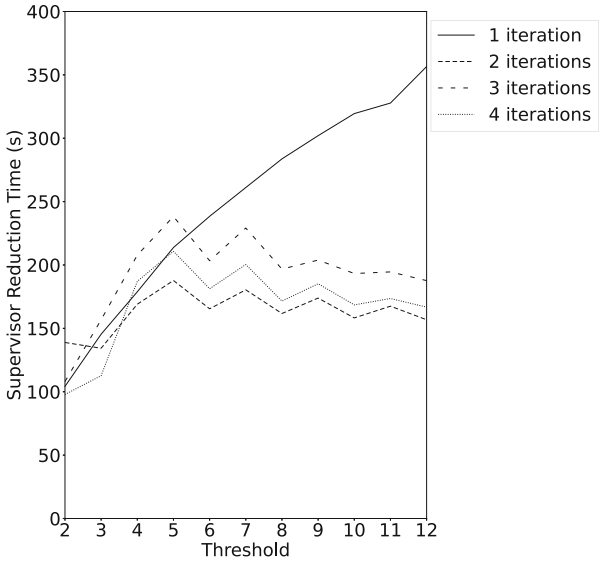


Fig. 8 Average supervisor reduction times depending on loop detection parameters

Loop detection broadens the sets of positive and negative traces in the training data. For each positive trace, a group of similar traces is identified that contain the same subsequences with more repetitions, and these similar traces are removed from the set of "don't care" traces and added to the set of positive traces; the set of negative traces is broadened likewise. This results in a more difficult supervisor reduction problem, which has the potential to provide a more accurate classification model provided that the broadening is accurate. This explains why most of the models computed with loop detection are more accurate than those without. Additionally, loop detection tends to produce smaller deterministic FSMs, which usually means faster supervisor reduction times.

5.4 Evaluation of different tokenisations

This section describes an experiment to determine how classification models are affected by different ways of filtering and tokenising system calls. This experiment again uses one iteration of loop detection with a threshold of eight, and the supervisor reduction options identified as most effective in Sect. 5.2, i.e., RBFS state ordering and Lexicographic or Diagonal1 pair ordering.

In addition to the six ways of tokenisation introduced in Table 1 in Sect. 4.3, a seventh tokenisation is considered that was originally proposed for ESCAPADE (Chew et al. 2024). This tokenisation, shown in Table 5, distinguishes system calls based on whether they access the *user directory* or another directory. The user directory on Android is a space for the user to store personal files such as photos, downloads, or multi-media, which are a primary target for crypto ransomware. The other directories predominantly contain system files, applications, and their configuration files. As shown in Table 5, the ESCAPADE tokenisation uses different granularities of tokens representing system calls in the user directory or in other directories, which was found effective for use with the hand-coded classification models of Chew et al. (2024).

Figure 9 shows the false positive and false negative rates in relation to the average state numbers of the classification models, which is represented by the size of the circles in the same format as in Fig. 6. Some tokenisations appear to have fewer than four classification models, which occurs when the same false positive and false negative rates are obtained using different supervisor reduction options. Additionally, Table 6 shows the classification models with the lowest false negative rates for each tokenisation.

Most of the tokenisations result in accuracy and state numbers comparable to those observed in Sect. 5.3, but there are outliers. The ESCAPADE tokenisation produces both the most and the least accurate classification models by large margins. With projection enabled, it produces a false positive rate of 0.2% and a false negative rate of 1.0%. The classification models have only 12 states on average and use only some of the tokens related to user

Table 5 ESCAPADE tokenisation

Operation	Tokens	
	User directory	Other directory
Open directory	OD_{udir}	OD_{other}
Open file for writing (create)	$OCWF_{udir}$	OF_{other}
Open file for reading or writing (excluding create)	OXF_{udir}	OF_{other}
Rename file	RN_{udir}	—
Unlink/delete file	U_{udir}	U_{other}

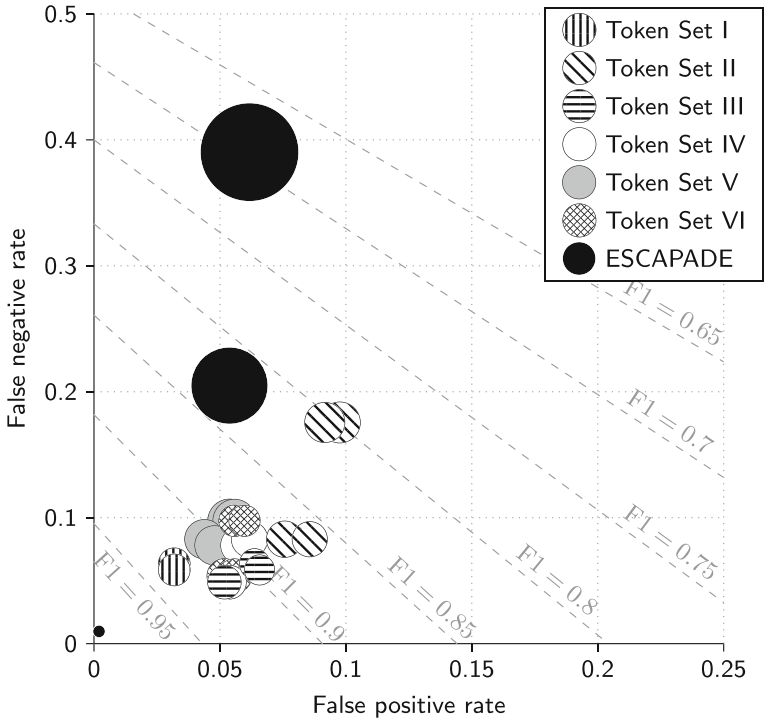


Fig. 9 Classification model accuracy and size for different tokenisations

directory access, while the other tokens (including all tokens related to non-user directories) have been projected out. These accurate and compact models are only found with projection enabled. Without projection, supervisor reduction fails to identify a reasonable pattern, which may be caused by the large number of tokens or the coarse granularity of the non-user directory tokens.

Unfortunately, the compact and accurate models produced by the ESCAPADE tokenisation may be caused by a bias in the dataset. The user directory is accessed by all 205 malicious applications, but only by 49 of the 502 benign applications, and this explains why projecting out the non-user directory tokens produces so small classification models. Yet, the bias may not only be a reflection of the fact that the user directory is the primary target of ransomware,

Table 6 Classification models with the lowest false negative rates for each tokenisation

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

which has also been observed to encrypt files in other directories. Benign applications may access files in the user directory if requested to do so, but such behaviour is unlikely to be exposed when user interaction is generated randomly by Android Monkey. A different way of capturing traces may be needed to produce more reliable classification models based on the ESCAPADE tokenisation.

Setting aside the ESCAPADE tokenisation, the lowest false negative rate of 4.9% in this experiment is shared by Token Set IV, which is the set used for all other experiments, and Set III. Set III differs from Set IV by the removal of tokens that have been removed by projection in Sect. 5.2, and this may explain the similarities. The best F1 score of 93.8% and the lowest false positive rate of 3.2% besides ESCAPADE is obtained by Set I. This may be a surprise, considering that Set I is the coarsest tokenisation that does not differentiate between the `openat()` system calls based on their flags. Yet, the small number of tokens makes it easy for supervisor reduction to minimise the FSM, while the low level of loop detection in this experiment seems to be enough to reduce misclassification. The slightly finer Set II does not achieve the same accuracy and produces some of the most inaccurate models. It seems that the increased number of tokens makes it more difficult to minimise the FSM, and the model does not benefit from the distinction between the write operations in the remaining tokenisations.

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

5.5 Unknown ransomware detection

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

The benign samples are randomly split into six sets of equal size, with five of the six sets used to build each classification model. As in Sect. 5.3, the traces are tokenised using Set IV and simplified using one iteration of loop detection with a threshold of eight, and the supervisor reduction options are those identified as most effective in Sect. 5.2, i.e., RBFS state ordering and Lexicographic or Diagonal1 pair ordering.

Figure 10 shows the false positive and false negative rates in relation to the average state numbers of the classification models, which is represented by the size of the circles in the same format as in Fig. 6. The diagram shows fairly low false positive rates: the highest false positive rate is 20.2% while the majority of the data points falls below 8%. These rates are comparable to those observed in Sect. 5.3, which is expected given the random split of the benign samples.

The false negative rates in this experiment are higher and deserve a more detailed analysis. Hence, Table 7 shows the classification models with the lowest false negative rates for each family. These results suggest that the proposed approach is capable of accurately classifying

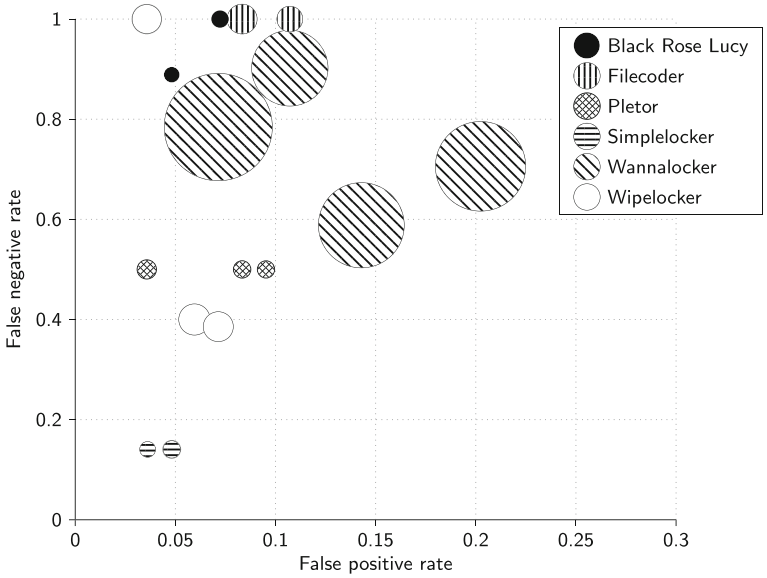


Fig. 10 Classification model accuracy and size for different ransomware families

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

5.6 Comparison to related work

This subsection compares the accuracy of the approach proposed in this paper to various static and dynamic methods from relevant literature as referenced in Sect. 2. The results

Table 7 Classification models with the lowest false negative rate in each family

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

Table 8 Accuracy of methods to detect Android malware

Method Name	Approach	Accuracy TPR	FNR	TNR	FPR	F1
Methods to detect Android ransomware:						
DNA-Droid (Gharib and Ghorbani 2017)	hybrid	0.981	0.019	0.995	0.005	—
R-PackDroid (Maiorca et al. 2017)	static	0.943	0.057	—	—	—
ESCAPADE (Chew et al. 2024)	dynamic	1.000	0.000	0.986	0.014	0.984
Amer and El-Sappagh (2022)	dynamic	0.989	0.011	0.989	0.011	0.980
Methods to detect other types of Android malware:						
DroidAPIMiner (Aafer et al. 2013)	static	0.978	0.022	—	—	—
DroidNative Alam et al. 2017	static	0.936	0.064	0.973	0.027	0.844
DroidDet (Zhu et al. 2018)	static	0.933	0.067	—	—	—
SWORD (Bhandari et al. 2018)	dynamic	0.958	0.042	0.926	0.074	0.943
MaMaDroid (Onwuzurike et al. 2019)	static	0.92	0.08	—	—	0.92
Supervisor reduction (this paper)	dynamic	0.956	0.044	0.966	0.034	0.938

are summarised in Table 8. Some accuracy values are missing in the table as they are not presented in the literature.

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

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

ESCAPADE (Chew et al. 2024) is based on the same dataset as this paper. Their method is based on patterns designed manually to capture all malicious traces in the dataset, resulting in a true positive rate of 1. The benign dataset is used to evaluate the efficacy of the patterns, with malicious behavioural patterns also detected in 1.4% of benign applications. These results are not based on cross validation.

Amer and El-Sappagh (2022) use several datasets to train their approach and evaluate it using 10-fold cross validation. One of the experiments uses the ESCAPADE dataset (Chew et al. 2024) and reports both true positive and true negative rates as 0.989. This result is shown in Table 8 as the data is the most similar to this paper.

The above methods are specifically designed to detect Android ransomware. The second group of entries in Table 8 covers a wider group of methods for different types of malware. While possibly less relevant, the detection and evaluation techniques employed in these methods are similar and comparable to ransomware detection.

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

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

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

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

MaMaDroid's (Onwuzurike et al. 2019) dataset contains 35493 malicious samples and 8447 benign samples, but only parts of the data are used in any single experiment. For the most relevant comparison, Table 8 shows results of 10-fold cross validation based on the latest dataset collected in 2016, which contains 2974 malicious samples and 2568 benign samples. Based on this data, the most accurate model produces a true positive rate of 0.92 and an F1-score of 0.92.

The last line in Table 8 represents the best classification model obtained with supervisor reduction as mentioned in Sect. 5.3, which is obtained using three iterations of loop detection and a threshold of eight, and lexicographic state ordering without projection. As the data shows, the method proposed here does not outperform all other methods—that is not to be expected from this first study of using supervisor reduction for ransomware detection, whereas machine learning techniques have been perfected by many researchers for years. The results show that supervisor reduction achieves accuracy scores comparable to those presented for other methods, and this suggests that supervisor reduction can be developed into an alternative new approach to solve this problem.

5.7 Threats to validity

A concern about the experiments described in this section is the limited size of the dataset, which consists of only 213 malicious applications from six ransomware families. The sparseness of the dataset raises issues in some experiments as mentioned in Sect. 5.4, and generally makes it difficult to determine how the method scales as the amount of data increases. This work focuses on a specific type of malware, namely crypto ransomware for Android devices, of which it was not possible to obtain more samples. While new ransomware will arise over time, the dataset in this paper is believed to be close to the current extent of Android crypto ransomware.

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

6 Conclusions

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

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

In future work, the authors would like to capture more traces of malicious and benign applications in order to run experiments with a larger dataset. The present work focuses on crypto ransomware for Android devices, and it would be interesting to apply the approach to other operating systems and other types of malware such as trojans, spyware, and locker-type ransomware. The final goal is to use the generated classification models as part of a monitoring system that recognises malicious applications and stops them before they can cause damage.

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

Data Availability As the samples used in this article contain malicious applications, the data is not publicly available. However, it can be accessed by request from the authors.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Aafer Y, Du W, Yin H (2013) DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Zia T, Zomaya A, Varadharajan V et al (eds) Security and privacy in communication networks, LNICST, vol 127. Springer International Publishing, pp 86–103. https://doi.org/10.1007/978-3-319-04283-1_6

Åkesson K, Fabian M, Flordal H et al (2006) Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In: 8th International workshop on discrete event systems. IEEE, pp 384–385. https://doi.org/10.1109/WODES.2006.382401

Alam S, Qu Z, Riley R et al (2017) DroidNative: automating and optimizing detection of Android native code malware variants. Computers & Security 65:230–246. https://doi.org/10.1016/j.cose.2016.11.011

Al-rimy BAS, Maarof MA, Shaid SZM (2018) Ransomware threat success factors, taxonomy, and countermeasures: a survey and research directions. Computers & Security 74:144–166. https://doi.org/10.1016/j.cose.2018.01.001

Amer E, El-Sappagh S (2022) Robust deep learning early alarm prediction model based on the behavioural smell for android malware. Computers & Security 116:102670. https://doi.org/10.1016/j.cose.2022.102670

Anderson HS, Kharkar A, Filar B et al (2017) Evading machine learning malware detection. Black Hat 2017:1–6

Andronio N, Zanero S, Maggi F (2015) HelDroid: dissecting and detecting mobile ransomware. In: Bos H, Monrose F, Blanc G (eds) RAID 2015: research in attacks, intrusions, and defenses, LNCS, vol 9404. Springer International Publishing, pp 382–404. https://doi.org/10.1007/978-3-319-26362-5_18

APKPure (n.d.) APKPure. https://apkpure.com/

Aurangzeb S, Aleem M, Iqbal MA et al (2017) Ransomware: a survey and trends. J Inf Assurance Secur 6(2):48–58

Bakour K, Ünver HM, Ghanem R (2018) The Android malware static analysis: techniques, limitations, and open challenges. In: 2018 3rd International conference on computer science and engineering (UBMK). IEEE Computer Society, pp 586–593. https://doi.org/10.1109/UBMK.2018.8566573

Beaucamps P, Gnaedig I, Marion JY (2010) Behavior abstraction in malware analysis. In: International conference on runtime verification, LNCS, vol 6418. Springer International Publishing, pp 168–182. https://doi.org/10.1007/978-3-642-16612-9_14

Bhandari S, Panihar R, Naval S et al (2018) SWORD: semantic aWare andrOid malwaRe Detector. J Inf Secur Appl 42:46–56. https://doi.org/10.1016/j.jisa.2018.07.003

Bidoki SM, Jalili S, Tajoddin A (2017) PbMMD: a novel policy based multi-process malware detection. Eng Appl Artif Intell 60:57–70. https://doi.org/10.1016/j.engappai.2016.12.008

Cai K, Wonham WM (2016) Supervisor localization: a top-down approach to distributed control of discrete-event systems, LNCS, vol 459. Springer

Chen J, Wang C, Zhao Z et al (2018) Uncovering the face of Android ransomware: characterization and real-time detection. IEEE Trans Inf Forensics Secur 13(5):1286–1300. https://doi.org/10.1109/TIFS.2017.2787905

Chew CJW (2023) Behaviour-based classification of encryption-type ransomware using system calls. PhD thesis, University of Waikato. https://hdl.handle.net/10289/15958

Chew CJW, Kumar V, Patros P et al (2024) Real-time system call-based ransomware detection. Int J Inf Secur. https://doi.org/10.1007/s10207-024-00819-x

Du S, Zhu P, Hua J et al (2021) An empirical analysis of hazardous uses of Android shared storage. IEEE Trans Dependable Secure Comput 18(1):340–355. https://doi.org/10.1109/TDSC.2018.2889486

Enck W, Gilbert P, Han S et al (2014) TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst 32(2). https://doi.org/10.1145/2619091

Fang Z, Wang J, Li B et al (2019) Evading anti-malware engines with deep reinforcement learning. IEEE Access 7:48867–48879. https://doi.org/10.1109/ACCESS.2019.2908033

Ferrante A, Malek M, Martinelli F et al (2018) Extinguishing ransomware – a hybrid approach to Android ransomware detection. In: Imine A, Fernandez JM, Marion JY et al (eds) Foundations and practice of security. Springer International Publishing, Cham, pp 242–258. https://doi.org/10.1007/978-3-319-75650-9_16

Gharib A, Ghorbani A (2017) DNA-Droid: a real-time Android ransomware detection framework. In: Yan Z, Molva R, Mazurczyk W et al (eds) Network and system security, LNCS, vol 10394. Springer International Publishing, pp 184–198. https://doi.org/10.1007/978-3-319-64701-2_14

Gonzalez D, Hayajneh T (2017) Detection and prevention of crypto-ansomware. In: 2017 IEEE 8th Annual ubiquitous computing, electronics and mobile communication conference (UEMCON). IEEE Computer Society, pp 472–478. https://doi.org/10.1109/UEMCON.2017.8249052

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

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

Google (2021) Meet Android Studio. https://developer.android.com/studio/intro

Guerra-Manzanares A, Luckner M, Bahsi H (2022) Android malware concept drift using system calls: detection, characterization and challenges. Expert Syst Appl 206:117200. https://doi.org/10.1016/j.eswa.2022.117200

Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall

Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation, 2nd edn. Addison-Wesley, Boston

Hou S, Saas A, Chen L et al (2016) 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. https://doi.org/10.1109/WIW.2016.040

Hull G, John H, Arief B (2019) Ransomware deployment methods and analysis: views from a predictive model and human responses. Crime Sci 8(2):22. https://doi.org/10.1186/s40163-019-0097-9

Humayun M, Jhanjhi N, Alsayat A et al (2021) Internet of things and ransomware: evolution, mitigation and prevention. Egyptian Informatics J 22(1):105–117. https://doi.org/10.1016/j.eij.2020.05.003

Islam R, Altas I (2012) A comparative study of malware family classification. In: International conference on information and communications security, LNCS, vol 7618. Springer International Publishing, pp 488–496. https://doi.org/10.1007/978-3-642-34129-8_48

Isohara T, Takemori K, Kubota A (2011) Kernel-based behavior analysis for Android malware detection. In: 2011 Seventh international conference on computational intelligence and security. IEEE Computer Society, pp 1011–1015. https://doi.org/10.1109/CIS.2011.226

Kharaz A, Arshad S, Mulliner C et al (2016) UNVEIL: a large-scale, automated approach to detecting ransomware. In: 25th USENIX security symposium (USENIX Security 16). USENIX Association, pp 757–772. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz

Ko JS, Jo JS, Kim DH et al (2019) Real time Android ransomware detection by analyzed Android applications. In: 2019 International Conference on Electronics, Information, and Communication (ICEIC). IEEE Computer Society. https://doi.org/10.23919/ELINFOCOM.2019.8706349

Kok S, Abdullah A, Jhanjhi N et al (2019) Ransomware, threat and detection techniques: a review. Int J Comput Sci Netw Secur 19(2):136–146. http://paper.ijcsns.org/07_book/201902/20190217.pdf

Koodous (2022) Collective intelligence against Android malware. https://koodous.com/

Kumar PR, Ramlie HREBH (2021) Anatomy of ransomware: attack stages, patterns and handling techniques. In: Suhaili WSH, Siau NZ, Omar S et al (eds) Computational intelligence in information systems, AISC, vol 1321. Springer International Publishing, pp 205–214. https://doi.org/10.1007/978-3-030-68133-3_20

Lemmou Y, Lanet JL, Souidi EM (2021) A behavioural in-depth analysis of ransomware infection. IET Inf Secur 15(1):38–58. https://doi.org/10.1049/ise2.12004

Levin DV (2020) Strace. https://strace.io/

Li L, Bissyandé TF, Papadakis M et al (2017) Static analysis of android apps: a systematic literature review. Inf Softw Technol 88:67–95. https://doi.org/10.1016/j.infsof.2017.04.001

Maiorca D, Mercaldo F, Giacinto G et al (2017) R-PackDroid: API package-based characterization and detection of mobile ransomware. In: SAC '17: Proceedings of the symposium on applied computing. Association for Computing Machinery, pp 1718–1723. https://doi.org/10.1145/3019612.3019793

Malik R (2020) Supervisor reduction by hiding events. IFAC PapersOnLine 53-6:1–6. https://doi.org/10.1016/j.ifacol.2023.01.001

Marsland S (2011) Machine learning: an algorithmic perspective. Chapman and Hall/CRC, New York, USA

Onwuzurike L, Mariconti E, Andriotis P et al (2019) Mamadroid: detecting Android malware by building Markov chains of behavioral models (extended version). ACM Trans Priv Secur 22(2). https://doi.org/10.1145/3313391

Paull MC, Unger SH (1959) Minimizing the number of states in incompletely specified sequential switching functions. IRE Trans Electronic Computers EC-8(3):356–367. https://doi.org/10.1109/TEC.1959.5222697

Poireault K (2022) Global ransomware damages to exceed $30bn by 2023. Infosecurity Magazine. https://www.infosecurity-magazine.com/news/ransomware-exceed-30bn-dollars-2023/

Ramadge PJG, Wonham WM (1989) The control of discrete event systems. Proc IEEE 77(1):81–98. https://doi.org/10.1109/5.21072

Scalas M, Maiorca D, Mercaldo F et al (2019) On the effectiveness of system API-related information for Android ransomware detection. Computers & Security 86:168–182. https://doi.org/10.1016/j.cose.2019.06.004

Sekar R, Bendre M, Dhurjati D et al (2000) A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings 2001 IEEE symposium on security and privacy. S&P 2001. IEEE Computer Society, pp 144–155. https://doi.org/10.1109/SECPRI.2001.924295

Sharma S, Krishna CR, Kumar R (2021) RansomDroid: forensic analysis and detection of Android ransomware using unsupervised machine learning technique. Forensic Sci Int: Digital Investigation 37:301168. https://doi.org/10.1016/j.fsidi.2021.301168

Shinde R, Van der Veeken P, Van Schooten S et al (2016) Ransomware: studying transfer and mitigation. In: 2016 International conference on computing, analytics and security trends (CAST). IEEE Computer Society, pp 90–95. https://doi.org/10.1109/CAST.2016.7914946

Sood G (2017) virustotal: R Client for the virustotal API. Virus Total. https://cran.r-project.org/package=virustotal, r package version 0.2.1

Su R, Wonham WM (2004) Supervisor reduction for discrete-event systems. Discrete Event Dyn Syst 14(1):31–53. https://doi.org/10.1023/B:DISC.0000005009.40749.b6

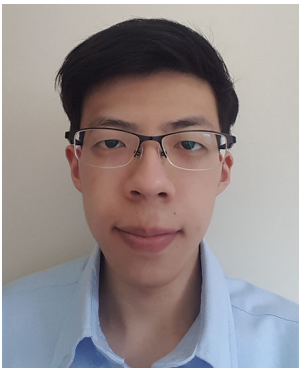
Tam K, Khan SJ, Fattori A et al (2015) CopperDroid: automatic reconstruction of Android malware behaviors. In: NDSS symposium 2015. Internet Society. https://doi.org/10.14722/ndss.2015.23145

Vaz AF, Wonham WM (1986) On supervisor reduction in discrete-event systems. Int J Control 44(2):475–491. https://doi.org/10.1080/00207178608933613

Wyke J, Ajjan A (2015) The current state of ransomware. A SophosLabs Technical Paper

Xu R, Saïdi H, Anderson R (2012) Aurasium: practical policy enforcement for Android applications. In: 21st USENIX security symposium (USENIX Security 12). USENIX Association, pp 539–552

Zhu HJ, You ZH, Zhu ZX et al (2018) DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 272:638–646. https://doi.org/10.1016/j.neucom.2017.07.030



Christopher Jun Wen Chew received his PhD in Computer Science from the University of Waikato in Hamilton, New Zealand. His research focuses on the Identification of Malicious Execution Patterns in Resource Constrained Environments using system call data. His current research interests are in the area of Android malware detection, ransomware, dynamic analysis, and automaton-based models.

Robi Malik received the M.S. and Ph.D. degree in computer science from the University of Kaiserslautern, Germany, in 1993 and 1997 respectively. During 1998-2002, he worked at Siemens Corporate Research in Munich, Germany, towards the development and application of modelling and analysis software for discrete event systems. From 2003-2023, he lectured Computer Science at the University of Waikato in Hamilton, New Zealand. He is participating in the development of the Waters/Supremica software for modelling and analysis of discrete event systems. His research interests include the verification and synthesis discrete event systems and their applications.

Vimal Kumar is a Senior Lecturer at the University of Waikato where he leads the Cyber Security Lab (CROW). He received his PhD in Computer Science from Missouri S&T in 2014 and worked as an Assistant Professor at the University of West Florida before joining the University of Waikato. His current research interests are in the area of attack detection and prevention, focusing on understanding malware behaviour and vulnerability descriptions. He was previously on the executive council of the NZ IoT Alliance and currently is part of the IT-012 joint committee of Standards Australia and Standards New Zealand.

Panos Patros CPEng specializes in AI, performance engineering and language runtimes. He completed the PhD in Computer Science at the IBM Centre of Advanced Studies - Atlantic, University of New Brunswick, Canada. He worked as a Senior Lecturer and the Deputy Head of the Department of Software Engineering at the University of Waikato, New Zealand. He is now serving as the Vice President of Engineering at Raygun, a software monitoring and performance engineering company in Wellington, New Zealand.