Computers & Security 149 (2025) 104193

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

Beyond the sandbox: Leveraging symbolic execution for evasive malware classification

Vasilis Vouvoutsis [a], Fran Casino [b,c,*], Constantinos Patsakis [a,c]

[a] Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou str., 18534 Piraeus, Greece
[b] Department of Computer Engineering and Mathematics, Universitat Rovira i Virgili, Spain
[c] Information Management Systems Institute, Athena Research Centre, Artemidos 6, Marousi 15125, Greece

ARTICLE INFO

Keywords:
Malware
Symbolic execution
Machine learning
Malware classification
Clustering

ABSTRACT

Threat actors continuously update their code to incorporate counter-analysis techniques designed to evade detection and hinder the blocking of their malware. The first line of defence for malware authors is often to bypass static analysis, a relatively straightforward task using readily available tools such as packers and cryptors. To address this shortcoming, defenders send potential malware samples for execution in a sandbox environment. While sandboxing can provide valuable insights into the behaviour of software on an information system, advanced techniques like anti-virtualisation and hooking evasion allow malware to escape detection. The primary objective of this work is to complement sandbox execution with symbolic execution frameworks to detect new malware strains efficiently. Symbolic execution offers a distinct advantage over sandboxing by achieving greater coverage of all possible execution traces, as it can explore every potential execution path, regardless of the evasion methods employed by the malware authors. By carefully selecting the samples to be analysed, we can significantly reduce the workload while extracting essential dynamic features in a fraction of the time and with far fewer computational resources compared to sandboxing. To this end, we leverage machine learning in an automated pipeline, enabling the accurate detection of sophisticated malware using a real-world dataset. Our approach yields average F1 scores of 0.93 for the benign class and 0.99 for the malware class in a binary classification setup, surpassing the detection rates reported in the literature. Additionally, our method outperforms a commercial malware sandbox when applied to the same dataset, further highlighting the efficacy of the proposed method.

1. Introduction

Modern ICT infrastructures are vulnerable to hostile infiltration due to the continuous expansion of network-connected devices and the growing complexity of operating systems. There are several difficulties in identifying and preventing malware due to the constant growth in the quantity and complexity of malware, as well as the development of new techniques. Commercial anti-malware solutions serve as an essential line of protection against malware attacks. Traditionally, antivirus software employed the signature-based technique to identify malware. A *signature* is a byte pattern that can be used to identify known malware. However, signature-based detection systems cannot protect against, e.g., zero-day exploits or obfuscated malware (Alkhateeb et al., 2023; Geng et al., 2024) or builders which can produce hundreds of variants of the same binary exhibiting similar capabilities without access to the source code.

Executing software inside a sandbox is one of the most popular ways to examine its behaviour. Practically, the binary is executed in a closely monitored and managed environment that simulates a real user environment. While sandbox analysis provides an attractive set of features regarding resources, accuracy, and performance, malware is currently exploiting evasion techniques such as anti-sandbox methods (Yokoyama et al., 2016), anti-virtualisation strategies or the ability to identify monitoring processes (Yokoyama et al., 2016; Rudd et al., 2017; Bulazel and Yener, 2017; Apostolopoulos et al., 2021) and alter their behaviour. As a result, the malware under inspection does not exhibit the intended malicious behaviour, hindering malware behaviour classification for analysts and detection of the automated identification of malicious actions.

In the standard paradigm of programme execution, a programme is executed with specified inputs following a predetermined control flow path. This conventional method entails the exploration of a singular control flow trajectory during a concrete execution, leading to a focused analysis of relevant attributes pertinent to that specific input and execution path. To circumvent this limitation, symbolic execution (Baldoni et al., 2018) can explore several potential pathways a programme may

* Corresponding author at: Department of Computer Engineering and Mathematics, Universitat Rovira i Virgili, Spain.
E-mail addresses: franciscojose.casino@urv.cat (F. Casino), kpatsak@unipi.gr (C. Patsakis).

https://doi.org/10.1016/j.cose.2024.104193
Received 23 July 2024; Received in revised form 16 September 2024; Accepted 4 November 2024
Available online 13 November 2024

follow in response to various inputs simultaneously. Symbolic execution emulates the execution of a programme by substituting "symbolic" values, representing variables that are not assigned concrete values, in place of the programme's input values. Each time the programme processes input, constraints or conditions are applied to these 'symbolic' values, guiding the simulation and influencing the programme's behaviour.

Since the simulation splits into two routes when a branch condition is encountered (i.e., one when the condition evaluates to *true*, and the other where it does not), it explores all available execution paths a piece of software can possibly have. For instance, an analyst might need to examine the executable's behaviour under certain conditions. With typical sandbox techniques, the analyst would need to deploy several virtual machines in the desired states, run the executable, and monitor its behaviour since malware exhibits different behaviour depending on the environment in which it is executed. In contrast, by taking advantage of the path explosion effect (Krishnamoorthy et al., 2010), the analyst can effortlessly filter out all execution states that do not meet specific criteria and further orchestrate the sample's state.

Several well-known examples of symbolic execution frameworks are EXE (Cadar et al., 2008b), KLEE (Cadar et al., 2008a), Mayhem (Cha et al., 2012), and S2E (Chipounov et al., 2012). They allow analysts to simulate binary execution on a host by replicating functions, system calls, and operating subsystems. In addition to the path exploration capabilities, one of the main advantages of these systems is decoupling the analysis from the need to use a dedicated (virtual) system and a specific operating system environment. Instead, an abstraction at the application layer (e.g., containerisation) is used to enhance the analysis effectiveness drastically.

The increasing volume of malware samples challenges analysts to swiftly identify and respond to emerging threats. In this article, we evaluate the effectiveness of binary simulation in malware analysis and classification. We focus on minimising the manual analysis workload for malware analysts through strategic sample filtering at each level. We propose a systematic approach that enhances efficiency and provides insights into the diverse malware landscape. Our approach employs different simulation levels to filter and prioritise samples, allowing analysts to concentrate on the most complex cases. More concretely, we explore the benefits of binary simulation, such as rapid processing and automated execution, and discuss challenges like addressing evasion techniques. Finally, we validate the efficacy of binary simulation and provide a fruitful discussion, contributing to advancing this research field.

Based on the above, the main novelty of this work can be summarised in the use of symbolic analysis in automated pipelines for malware detection. Further than merely using symbolic analysis to analyse a single sample, we do this in scale, managing to have better results than many works using sandboxes, but at a fraction of time and computational resources. While not generic enough to cover all malware samples, we anticipate that this gap will soon be adequately filled due to the maturity of existing frameworks. For example, with our approach, all necessary features are extracted within 30 s instead of the minimum two minutes that are used by all malware sandboxes. Moreover, the use of symbolic execution prevents the bypasses that malware authors embed in their code and would escape sandbox execution. Note that with proper filtering of the samples that have to be analysed, e.g., by using TLSH clustering, we can further minimise the time and computational resources without compromising the detection accuracy.

The rest of this article is organised as follows. The next section provides the reader with the relevant background and an overview of the related work. Section 3 describes our methodology and details the automated pipeline constructed for symbolic execution and feature extraction. Section 4 presents the datasets, the training methodology used, as well as the experimental setup and the tests performed to evaluate our method, even against commercial malware sandboxes. Section 5 discusses the results and findings. Finally, Section 6 concludes the article and identifies potential directions for future work.

2. Background and related work

In recent years, symbolic execution has gained prominence as a technique for analysing programme behaviour by systematically exploring all possible execution paths of a binary. This capability has proven crucial in contexts such as malware detection and analysis, where dynamic evasion techniques often circumvent traditional sandbox-based analysis. Our approach leverages symbolic execution in malware detection and classification, contributing to an existing body of research focused on improving the efficiency and accuracy of malware detection mechanisms.

2.1. Detection approaches

Malware detection technologies are classified as static or dynamic depending on whether or not the target software sample is executed.

We may extract static information from target programmes using static detection techniques (Han et al., 2019), including strings, opcodes, API calls, n-grams, CFG, imports, and entropy. Signature-based detection was proposed early on. At that point, it was considered that automatic signature development of malware was vital, which boosted pattern matching performance (Griffin et al., 2009; Kephart, 1994). However, signature-based detection requires regular signature improvements and a large maintenance budget. Moreover, code obfuscation and binary transformation techniques that lead to malware in polymorphic forms may be resistant to such approaches (Moser et al., 2007). To address these shortcomings, researchers have used code normalisation to capture the inherent original maliciousness of the executable (Christodorescu et al., 2005) or use features that typically remain the same after such manipulations, e.g., hashes of the imported libraries (Mandiant, 2014; Naik et al., 2020), headers (Joyce et al., 2019), but also mapping the files to images and then use machine and deep learning to classify them (Nataraj et al., 2011; Vasan et al., 2020). For more on the use of machine learning in malware detection and classification, the interested reader may refer to Gibert et al. (2020).

On the other hand, dynamic analysis examines a programme's activities during execution (Egele et al., 2008; Sihwail et al., 2018). A file under examination is executed in a controlled and monitored sandbox to identify what it does regarding file system modifications, network connections, launching processes, or making specified system functions (Or-Meir et al., 2019). Because the file is being executed, the analysis is oblivious to obfuscation and packing because the file will be unpacked, memory dumps may be retrieved, and the resulting execution path will be disclosed.

However, as malware analysis research evolves, malicious actors are catching up since malware as a service model is rather prosperous (Patsakis et al., 2024). Dynamic analysis in a sandbox has started reaching its limitations. Malware uses evasion and anti-analysis techniques like fingerprinting (Oyama, 2018), stalling, trigger-based tactics, and traps (Afianian et al., 2019) to prevent the analyst from understanding the executable's nature. To handle such cases, researchers have utilised binary emulation to orchestrate the execution towards its malicious path (Mow et al., 2022). The analyst can hook into the binary during execution and redirect execution around the evasive code (Ziegler, 2021).

Feature analysis is crucial to extracting relevant information from malware samples for classification purposes. Various static and dynamic features have been explored to capture the distinctive characteristics of malicious code. Christodorescu et al. (2005) addressed the limitations of signature-based detection by proposing code normalisation techniques to capture the canonicalised original maliciousness of executable code. Han et al. (2019) introduced MalDAE, a malware detection system that utilises static features such as strings, opcodes, API calls, n-grams, control flow graphs (CFG), imports, and entropy, exemplifying the significance of diverse feature sets in enhancing malware detection accuracy.

Meanwhile, researchers have increasingly integrated binary emulation with feature analysis to counter evolving malware strategies that evade detection. This combined approach enhances the effectiveness of malware detection by leveraging both dynamic and static techniques.

Salehi et al. (2017) proposed a model that integrates static and dynamic analysis with machine learning to address zero-day malware detection challenges, reducing model-building time without sacrificing accuracy. Xue et al. (2019) introduced Malscore, a classification system that uses convolutional neural networks for static feature analysis and machine learning for dynamic features, achieving a 98.82% accuracy in malware classification, even against obfuscation techniques.

Binary emulation, a key element of this integration, allows researchers to execute code in controlled environments and observe malicious behaviour dynamically. Mow et al. (2022) demonstrated how binary emulation could direct execution towards malicious paths, enabling analysts to bypass evasive code and monitor behaviour. This approach also underpins broader sandbox-based malware analysis, providing a more robust solution for detecting sophisticated malware.

2.2. Symbolic execution

Symbolic execution has been extensively explored to address challenges in programme verification and vulnerability detection.The authors in Baldoni et al. (2018) and Cadar and Sen (2013) provide comprehensive surveys on symbolic execution techniques, demonstrating their utility in expanding the execution space by exploring multiple control flow paths simultaneously. These frameworks have evolved from earlier symbolic execution platforms such as EXE and KLEE (Cadar et al., 2008b), showcasing how symbolic execution can automatically generate high-coverage test cases for complex software programmes. Building on this, Cha et al. (2012) introduced Mayhem, a framework designed to identify vulnerabilities in binary code using symbolic execution. Similarly, Chipounov et al. (2011) developed S2E, which enables analysts to conduct binary analysis across different system architectures by decoupling the analysis from the environment, thus enhancing the scalability of symbolic execution.

In malware analysis, symbolic execution has been adapted to combat anti-sandbox techniques and detect evasive behaviour. Shoshitaishvili et al. (2016) introduced Angr, a platform combining static and dynamic analysis, which is notable for its ability to track control and data dependencies during symbolic execution, making it highly effective in uncovering malware's evasive strategies. Furthermore, Saudel and Salwan Saudel and Salwan (2015) presented Triton, which extends symbolic execution for taint analysis, enhancing the identification of critical execution paths. These advancements underscore the importance of symbolic execution in the context of malware, where static and traditional dynamic analysis often fail to capture sophisticated behaviours embedded in obfuscated code.

2.3. Behavioural signatures

Behavioural signatures play an essential role in dynamic malware detection. The concept revolves around identifying patterns in the programme's behaviour during execution. Canali et al. Canali et al. (2012) and Christodorescu et al. Christodorescu et al. (2007) proposed methods for mining malicious behaviour by comparing execution traces with known behavioural signatures, enabling the classification of previously unseen malware. These methods focus on capturing key features such as system calls, file system interactions, and network connections that are often manipulated by malware to achieve persistence or evade detection.

A suggested static analysis method for feature extraction based on symbolic execution can be found in the work of Namani et al. Namani and Khan (2020). The goal of the suggested system was to fix indirect jumps and calls as well as dynamic imports, providing a framework for improving detection accuracy and reducing false positives. Sebastio et al. Sebastio et al. (2020) proposed a method that uses symbolic execution to classify malware based on its behaviour. They introduce several optimisations to improve the efficiency of the symbolic execution process, including context-bounded symbolic execution, selective symbolic execution, and abstraction techniques. These optimisations allow the proposed approach to effectively handle large scale malware datasets while maintaining accuracy. Similarly, Bertrand Van Ouytsel and Legay Bertrand Van Ouytsel and Legay (2022) also proposed a method that first uses symbolic execution to generate path constraints from programme inputs, which are then converted into a graph representation. Next, the graph is analysed using a graph kernel algorithm, which extracts features based on the structure of the graph. These features are used to train a machine learning classifier to identify whether a given programme exhibits malicious behaviour or not.

2.4. Angr

Angr is a binary analysis framework incorporating several of the most advanced binary analysis techniques available in the literature (Shoshitaishvili et al., 2016). To make it simple to execute suggested research methods and evaluate their efficacy to one another, Angr offers building blocks for various studies utilising static and dynamic methodologies. It allows an analyst to utilise Disassembly and intermediate-representation lifting, Programme instrumentation, Symbolic execution, Control-flow analysis, Data-dependency analysis, Value-set analysis (VSA) and Decompilation. Angr performs the majority of its analysis on an intermediate representation, a structured description of the fundamental actions performed by each CPU instruction, to be able to analyse and execute machine code from different CPU architectures, such as MIPS, ARM, and PowerPC in addition to the classic x86. To store the different states of a programme during its symbolic execution, Angr uses a special structure called stash.

3. Methodology

One of the main motivations in this work is to explore the balance between the automated detection of malware stains and accuracy. To this end, we argue that various combinations can be used between static analysis and manual reverse engineering of a malware sample. To this end, we argue that before resorting to sandbox execution, other approaches, such as symbolic execution, can efficiently achieve this task. As a result, we favour a multi-layer approach where samples are pruned at each layer before being passed to the subsequent analysis layer. In terms of use of resources, this is mapped in Fig. 1. Therefore, this can lead to prioritising which samples must be manually checked, having already collected a considerable amount of information that the analyst can use. Indeed, as we showcase in our experiments, our approach is far more efficient in terms of the use of resources compared to both sandbox execution and binary emulation without compromising detection accuracy and precision.

Our approach to constructing a malware detection pipeline using symbolic execution is based on a comprehensive methodology designed to maximise efficiency and accuracy. The process begins with collecting diverse data samples, ensuring a broad representation of potential malware variants. To manage large datasets effectively, we employ clustering techniques, such as TLSH clustering, grouping similar binaries together to ease identification. Symbolic execution is then applied to perform in-depth analysis, extracting detailed execution traces for subsequent feature extraction. The derived features capture behavioural patterns indicative of malware and serve as valuable inputs for machine learning classification. Integrating machine learning algorithms enables automated and accurate classification based on learned patterns. Following classification, the pipeline undergoes evaluation, generating detailed reports to provide insights into its performance, strengths, and limitations. Finally, continuous optimisation ensures adaptability to evolving malware characteristics.

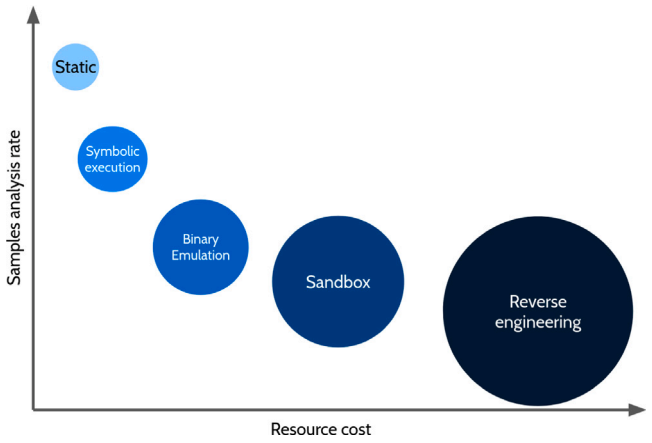


Fig. 1. Resource allocation for each detection approach.

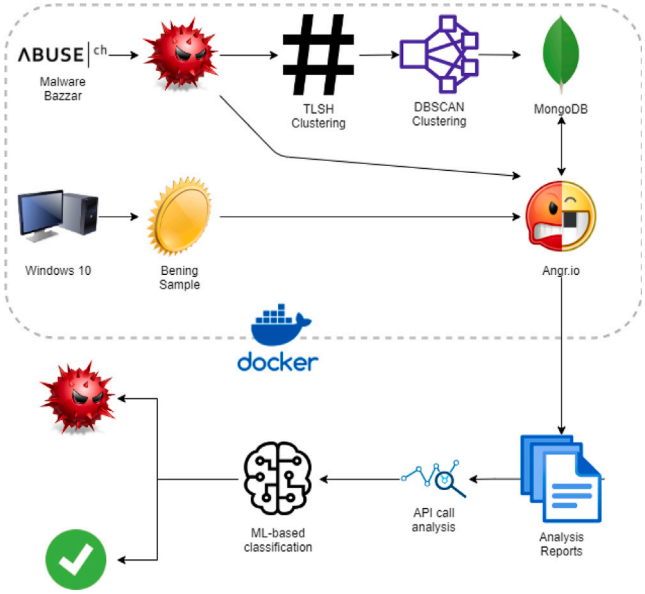


Fig. 2. Overview of the developed workflow.

For our experiments, we utilised the Angr symbolic execution framework primarily because of its platform-agnostic nature. The multiple sequential pipelines we developed for our experiments are illustrated in Fig. 2.

3.1. Sample selection

Instead of blindly analysing all samples available, we opted for a technique to further diversify our tests by implementing TLSH clustering (Oliver et al., 2013, 2021, 2020). TLSH is a software and library for fuzzy matching. TLSH creates a hash value from a file (min 50 bytes) that may be used for similarity comparisons. Similar files will have similar hash values, allowing similar items to be detected by comparing their hash values. Several organisations and malware repositories have used TLSH. Because of its performance and scalability and because it is significantly more difficult to attack and evade than other similarity digests, TLSH is quickly becoming a standard choice for threat hunting and related security processing. We initially consumed all TLSH hashes of the samples available to us, and then we applied the DBSCAN data clustering algorithm proposed by Ester et al. (1996). It is a non-parametric density-based clustering algorithm: given a collection of points in a space, it clusters together points that are tightly packed together (points with many nearby neighbours), identifying outliers as points that lie alone in low-density regions (whose nearest neighbours are too far away). After experimenting with the number of clusters produced and the number of unclustered samples, we used a maximum distance score of 100 between two samples for one to be considered in the neighbourhood of the other since that maximised the inter-cluster distance. Given the high detection and low false positive rates for thresholds approaching 100 for TLSH (Oliver et al., 2013), we opted to use this as a threshold after verifying the above for our dataset; see next section. Once the clustering phase with TLSH is over, all results are stored in MongoDB (MongoDB, 2009) database.

3.2. Execution of the symbolic analysis

Using a representative from each cluster of the previous step, we prune the number of samples that have to undergo symbolic execution. For the symbolic analysis, first, we verify that a sample was not previously analysed, as discussed in Section 3.1. If the sample is valid for analysis, we load the PE file in Angr. For a timeout of 600 s, Angr's simulation manager is executed, which allows for managing symbolic execution across groups of states. To monitor execution and construct the traces, breakpoints that are used to record system calls as they occur are set. All execution paths are organised into stashes based on their current state. After completing the simulation or reaching a timeout, all Angr stashes are iterated to extract the system calls. The results are then written back into the document database.

3.3. Feature extraction

The resulting reports for each sample from Angr were collected, and the API calls were extracted. To extract API calls, all execution traces were collected for all generated paths by the symbolic execution. The algorithm then iterates over all generated paths and extracts all recorded system calls.

We added a unification step to the process, in addition to the API call extraction, to make the resulting dataset containing the feature information less sparse and group relevant artefacts together. This step is critical because there are too many discrete API calls that, if used as-is, would result in a considerably sparser data collection than the one collected (which is already sparse). Be aware that sparse datasets create extra challenges when establishing a dataset's statistical characteristics and correlations, affecting the predictive potential of machine learning approaches in various situations (Kuss, 2002; Zigomitros et al., 2020; Casino et al., 2019; Li et al., 2016).

We adopted a similar approach to our recent research (Vouvoutsis et al., 2022) for the unification process, and is comprised of the following steps:

- **API counter:** Every reported API call was enumerated for each generated report.
- **Feature counter:** For each sample analysed, we added a feature that counts/summarises all features available for each sample in our dataset.
- **State counter:** Based on the fact that symbolic execution tries to analyse all available paths, we enhance our feature collection with the status of each trace. The included statuses are: Excess Loop, Active, Dead-ended, Errored, and Unconstrained.
- **ASCII/Unicode Win32 API merge:** Win32 supports two API versions if any of the API's arguments is a string. These are the ASCII and Unicode versions of the API, which generate the letters A and W, respectively. The ASCII API accepts ASCII strings, whereas the Unicode API takes Unicode-wide character strings (Mohanta and Saldanha, 2020; Microsoft, 2022).

- **Extended Win32 API merge:** A more extended version of some Win32 APIs exists. The suffix 'Ex' denotes the enlarged version of an API. The difference between a non-extended and an extended API is that the extended version may accept more parameters/arguments and may also include more functions (Mohanta and Saldanha, 2020). Some compromises had to be made to accommodate both the old 16-bit Windows API and the new 32-bit Windows API. Because both APIs had several functions available, one had to be altered, and for backward compatibility, it was nearly always the 32-bit version.
- **C runtime library:** The Microsoft runtime library provides routines for programming the Microsoft Windows operating system. Many common programming tasks that the C and C++ languages do not offer can be automated using these methods (Microsoft, 2022). Many of these methods have different names but use the same code.
- **Interesting APIs**. We selected to include the arguments of the invoked Win32 API in our study. The APIs employed are:
  - `LoadLibrary*` function (`libloaderapi.h`). Loads the supplied module into the address space of the caller process. Additional modules may be loaded as a result of the specified module. Malware authors usually prefer not to employ load-time linking when accessing Windows APIs. They conceal such calls instead by dynamically loading and resolving API references.
  - `GetModuleHandle*` function (`libloaderapi.h`). This method returns the module handle for the provided module. The module must have been loaded by the caller process. Some modules are automatically loaded into all non-native-system processes. Such modules are `ntdll.dll` and `kernel32.dll`, therefore `LoadLibrary/FreeLibrary` do not need to be called on these and can instead just call `GetModuleHandle`.
  - `GetProcAddress` function (`libloaderapi.h`). The address of an exported function or variable from the given dynamic-link library is returned (DLL). Most malware researchers understand how APIs can be dynamically resolved via a call to `GetProcAddress`, so they can track down all calls to that function and check the second parameter, or `lpProcName`, for the presence of high-risk APIs (or those that can download and run the malware's actual payload). This generally aids them in narrowing down and eliminating API calls that do not provide a significant security risk.
- **Name mangling:** The practice of encoding function and variable names into unique names so that linkers can discern common names in the language is known as name mangling. Some names are mangled in a manner akin to Visual C/C++ (Agner, 2014). In such case, our technique is to use the name's strings to construct a signature based on Bonfante and Nogues (2016), but further extended to retain the majority of the contained information.

4. Experiments

In this section, we describe the reference dataset and the features of the samples. Moreover, we analyse the experiments performed, including TLSH clustering and machine learning classification.

4.1. Reference database

We choose to acquire active and current malware for a comprehensive and realistic database to analyse our methods appropriately. As a result, we downloaded from abuse.ch (2021) all executable malware samples available, identified by mime type `x-dosexec` and file type guess `exe` in the database. We only retained the Windows PE files, non-dynamic libraries, and unmanaged code executables from these files. This is due to the fact that unmanaged code executables compile directly to machine code and are executed by the operating system. Thus, the total number of malicious files we used is 14761. Beyond these malware samples, we added 1531 benign files. These files were collected from a Windows 10 installation by performing a full drive file listing and then filtering out non-PE, managed executable, that is `.dll`, `.sys`, and `.mui` files.

To speed up the clustering process, we first split our sample database based on the recorded family provided by abuse.ch (2021). We then apply the clustering algorithm to each group of samples. Afterwards, during the sample symbolic analysis, we retrieve the cluster to which each sample belongs. If no other sample from the identified cluster has been previously successfully analysed, then we proceed with the symbolic execution and the execution traces generation. The clustering process helped us to significantly reduce the number of malicious files for analysis. Out of the original 74,712 files, we only had to examine 14,761, a reduction of 80.3%.

To verify that the TLSH clustering would not impede the machine learning classification, we evaluated the TLSH clustering classification on our dataset. As it can be observed in Fig. 3, the empirical threshold of 100 referred to in Oliver et al. (2013) is valid for our dataset. Notably, we have a negligible false positive rate (0.046%) and a significantly lower detection rate (11.3%) than the original. This can be attributed to the differences in our dataset with the original and the inclusion of far more families than the dataset used in Oliver et al. (2013).

4.2. Classification experiments

In the following sections, we describe the feature extraction and classification experiments along with the benchmarks used.

4.2.1. Feature extraction strategies

Given the reference database described above, we used two feature extraction strategies to create two datasets for input to the machine learning algorithms. These strategies are described as follows:

- Summarised strategy: In the Summarised strategy, we chose to summarise all distinct feature counts for each analysed sample to produce one data sample for all execution traces of each analysed sample. This created the DB_{SUM} dataset.
- Longest Path strategy: In the Longest Path strategy, we kept the longest execution trace of each analysed sample. This created the DB_{LP} dataset, see Table 1.

4.2.2. Analysis

For our classification experiment, we utilised the auto-sklearn (Feurer et al., 2015) machine learning toolkit to find the optimum settings, depending on Bayesian optimisation, meta-learning, and ensemble building. We computed 5-fold- and 10-fold cross-validation strategies to assess the models' accuracy and robustness and avoid possible overfitting issues. The resulting auto-sklearn pipelines apply multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone (Rokach, 2010). Pipelines generated on DB_{SUM} and DB_{LP} based on the cross-validation strategies require an order of magnitude fewer models to achieve good predictions. As seen in Table 2, all auto-sklearn pipelines produced an accuracy score of over 0.98, with a slight deviation fluctuating from 0.984 to 0.987 accuracy. 10-fold cross validation pipelines achieved both precision and F1-score of 0.99 for malware samples and over 0.93 for benign. Results for both DB_{SUM} and DB_{LP} are also similar, with slightly better accuracy towards DB_{LP}.

An additional experiment was performed to investigate correlations among families. Thus, a multiclass classification experiment was performed. In this case, since the experiment aims to showcase potential

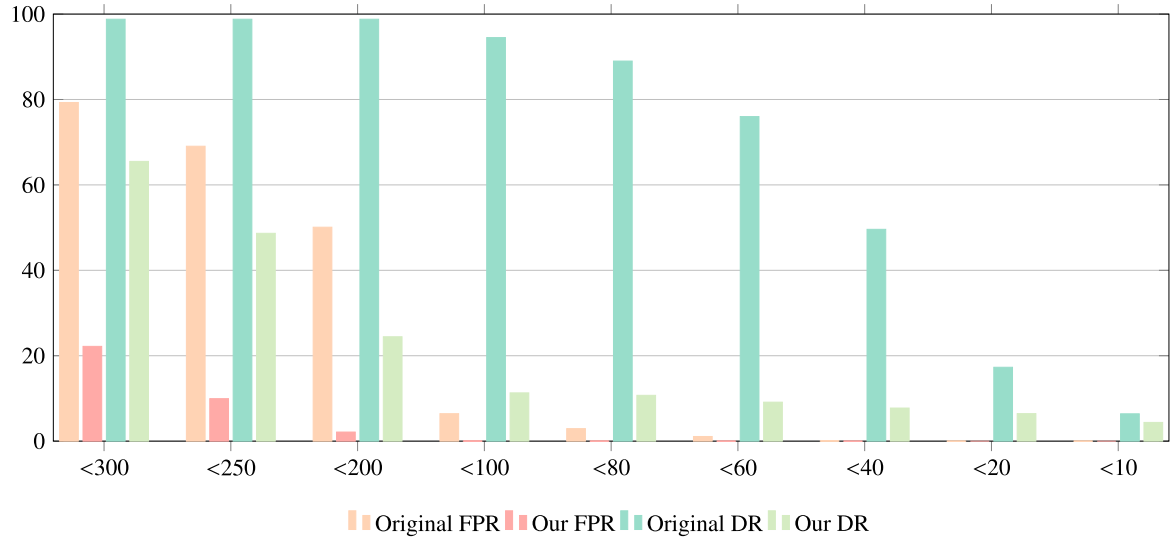


Fig. 3. Comparison of the false positive (FPR) and detection rates (DR) for TLSH in the original (Oliver et al., 2013) and our dataset.

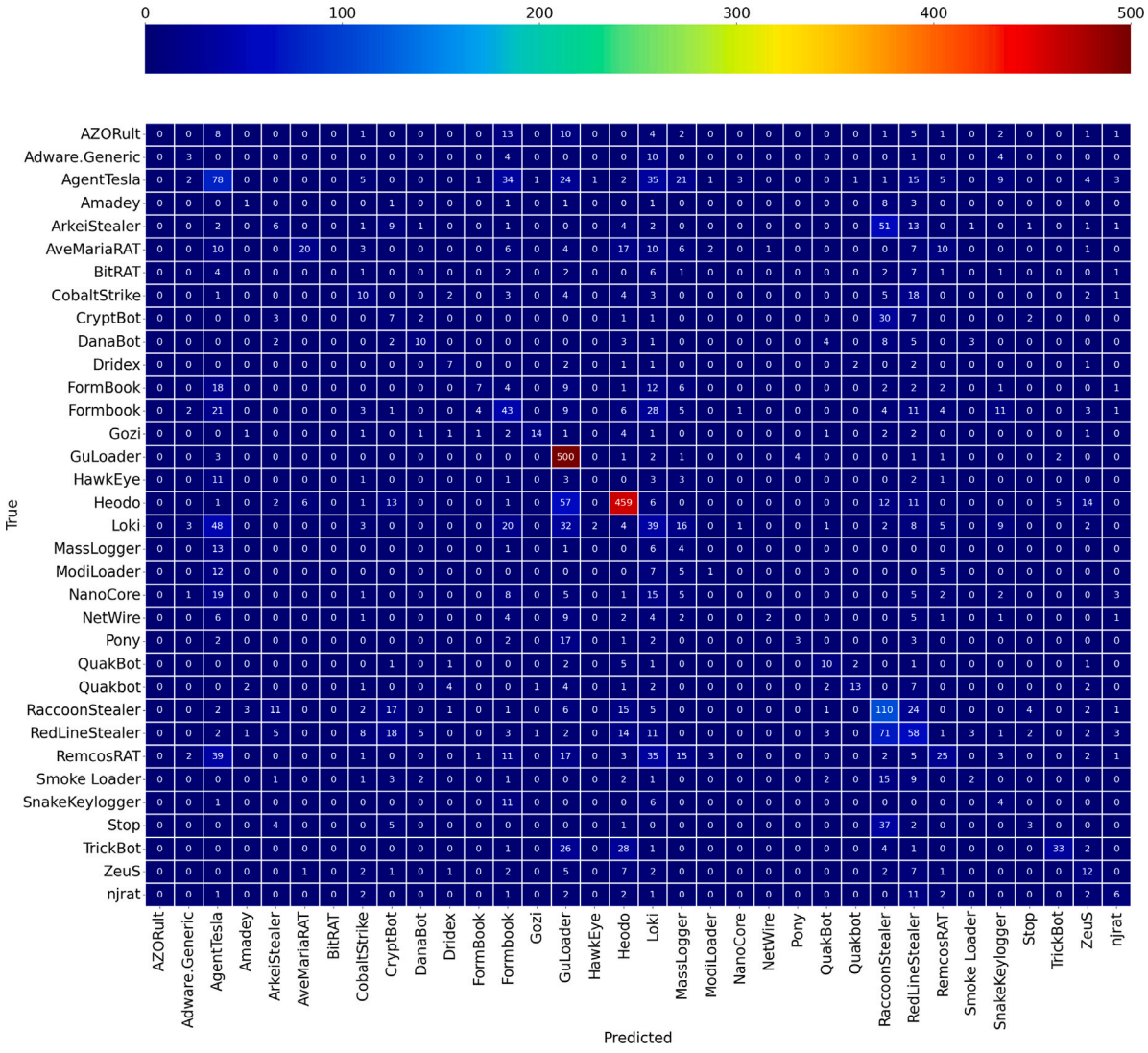


Fig. 4. Confusion matrix corresponding to the average 10-fold cross validation setup.

Table 1
Sample of DB_{LP}.

sha256	Signature	GetStartupInfo	HeapAlloc	HeapFree	...	Count
...159c1b44aed8a0bc1c0cd...	Loki	0	0	0	...	42
...f179174f588fbf28e8766...	ArkeiStealer	26	56	4	...	409
...f5607adfecd39ec459e76...	CobaltStrike	56	94	0	...	1170
...9dfbccf28120fd63299e8...	Heodo	123	0	0	...	1336
...1fb58c88490ab16540f2e...	Heodo	45	135	0	...	1489
...	...	...	...	...	...	...
...f00903a82562ae9eecf70...	BitRAT	5	0	0	...	112
...300389cba0abfa3374775...	Heodo	82	120	0	...	1662
...39721c6c3098e0153ed36...	GuLoader	0	0	0	...	6
...015593f8e2fce0b002289...	GuLoader	0	0	0	...	6
...941bd4bc57f638a8dfd21...	GuLoader	0	0	0	...	6

Table 2
Classification reports from the auto-sklearn pipelines.

Strategy	Dataset	Class	Precision	Recall	F1-score
Summarised	Cross-Validation-5	Benign	0.996	0.865	0.924
Summarised	Cross-Validation-5	Malware	0.994	1.000	0.991
Summarised	Cross-Validation-10	Benign	0.992	0.871	0.934
Summarised	Cross-Validation-10	Malware	0.991	1.000	0.992
Longest path	Cross-Validation-5	Benign	0.963	0.887	0.924
Longest path	Cross-Validation-5	Malware	0.990	1.000	0.992
Longest path	Cross-Validation-10	Benign	0.957	0.912	0.932
Longest path	Cross-Validation-10	Malware	0.990	1.000	0.993

overlapping, we created a confusion matrix with the classification outcomes, as seen in Fig. 4.

According to Fig. 4, several families, such as GuLoader, and Heodo, were accurately identified. In the case of families with a low amount of samples, their identification is prone to errors due to, e.g., not having enough training samples or a feature overlapping issue. In this regard, several families suffered from misclassification issues, with AgentTesla, Formbook, ArkeiStealer, Loki, RaccoonStealer, RedLineStealer, RemcosRAT, and TrickBot being the most impacted cases. From these families, we find that families such as AgentTesla, Formbook and Loki Remcos are usually confused since they have overlapping features. The latter connection has also been observed and documented in the literature (Casino et al., 2023), confirming the possibility that either the same perpetrators are behind different campaigns or malicious actors are using previously published samples and materials to enhance their malware.

4.2.3. DB_{SUM} and DB_{LP} comparison

To better understand our results, first, we compare the results between DB_{SUM} and DB_{LP} for the 10-fold cross-validation experiments. The experiments over DB_{SUM} achieved the same accuracy as the ones in DB_{LP}. Next, to get a better insight into the models that were selected by auto-sklearn, we use PipelineProfiler (Ono et al., 2020). As it can be observed in Fig. 5, the pipeline for DB_{SUM} has generated only one ensemble model that used a Random Forest (RF) classifier. However, the pipeline for DB_{LP} required three ensemble models with different weights, see Fig. 6, to construct the machine learning model. Instead, the model with the biggest weight was based on the Adaboost Classifier, while the rest of the models with a minor ensemble weight used the RF classifier. Moreover, in DB_{LP}, more preprocessing and balancing steps were added in the auto-sklearn pipeline.

For the sake of explainability and interpretability, we computed the Shapley Additive Explanations (SHAP) of DB_{SUM} and DB_{LP} experiments with the cross-validation 10 setup. The SHAP values are shown in Fig. 7.

As it can be observed, the most relevant feature in all cases is *GetProcessHeap*, a function often used by malware for process injection or virtualisation detection (Kemkes, 2019). In the case of DB_{SUM}, as the samples represent all feature counts, they can be seen as the average relevance of all execution traces. In this regard, features such

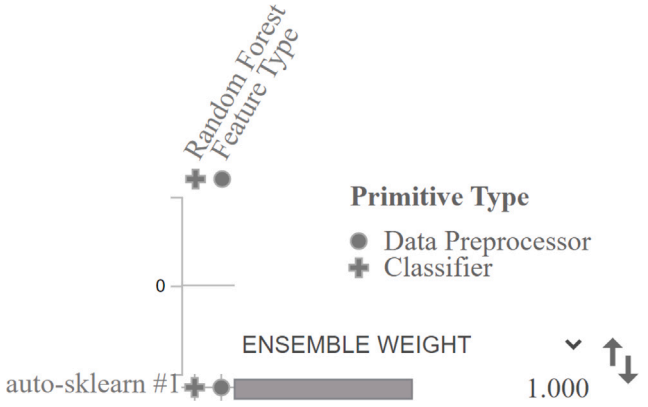


Fig. 5. Profiling of DB_{SUM} experiment with Cross Validation = 10. A single ensemble model was generated in the DB_{SUM} pipeline, utilising the Random Forest (RF) classifier. Generated by PipelineProfiler library (Ono et al., 2020).

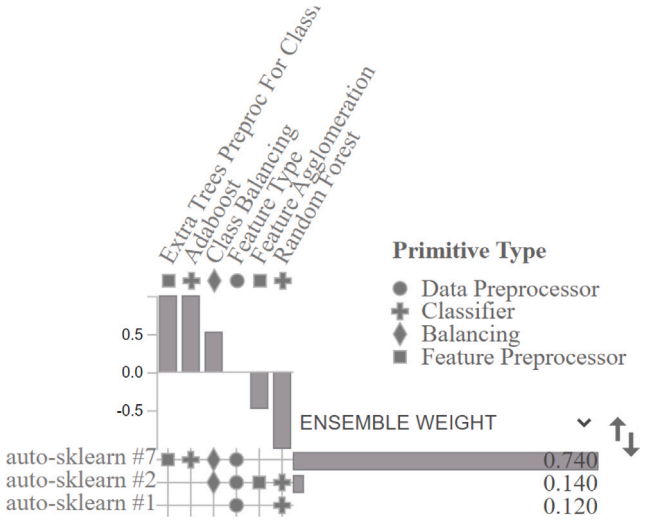
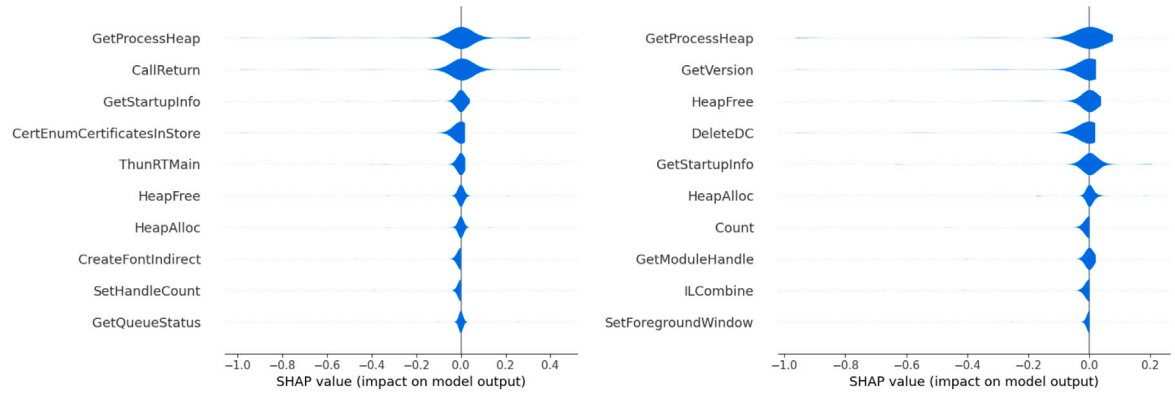


Fig. 6. Profiling of DB_{LP} experiment with Cross Validation = 10. An ensemble of three models was employed in constructing the Machine Learning model for the DB_{LP} pipeline, each with distinct weights. The model with the most significant weight was founded on the Adaboost Classifier. Conversely, the remaining models, which were assigned lesser ensemble weights, utilised the Random Forest (RF) classifier. Generated by PipelineProfiler library (Ono et al., 2020).



(a) SHAP values for DB_{SUM}.

(b) SHAP values for DB_{LP}.

Fig. 7. Detail of the SHAP values for DB_{SUM} and DB_{LP} in the cross-validation 10 setup.

as *CallReturn* and *GetStartupinfo* appear to be the most relevant from the ones appearing in such executions, highlighting their contribution to the classification task. In DB_{LP}, we see different features as the most relevant ones since the longest execution path of each sample is considered. In this case, we see features such as *GetVersion* and *HeapFree* (i.e., the latter also appears as one of the most relevant features in DB_{SUM}) being the most relevant in the binary classification task.

4.3. Comparison with a commercial sandbox

To better understand the capacity of our pipeline and looking beyond academic work, we decided to compare our results with a malware sandbox used in a real world. Hatching's Triage[1] is a well-known malware sandbox that analyses thousands of malware samples in specially crafted sandboxes for Windows, MacOS, Linux, and Android. Using its API, we searched for the analysis reports of the 14 761 malware samples in our database. From them, we retrieved 14 691 behavioural reports. Triage uses a scoring system for each sample, where scores above 7 are malicious files, samples with scores above 4 and below 8 are considered suspicious, and everything below 5 is benign or unknown. Based on the latter, the maliciousness of the samples according to Triage is illustrated in Fig. 8. As it can be observed, more than a 20% (21,7%) of the samples are very evasive and manage to bypass the dynamic analysis of a commercial sandbox, further justifying our dataset choice. Yet, these evasive mechanisms are bypassed since the malicious payload is executed with the symbolic execution, and the maliciousness of the sample is immediately detected. Of course, these results clearly demonstrate the capacity of our detection framework and with a fraction of the resources, as, for example, each sample was executed in a virtual machine for at least two minutes. Certainly, our approach does not manage to record the events happening in a sandbox when the sample is executed; nevertheless, detection-wise, our dynamic analysis approach is far better than a sandbox, showcasing its complementarity.

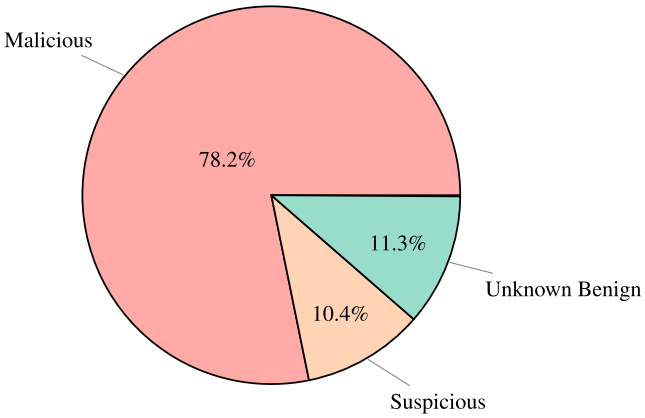


Fig. 8. Classification outcomes of the samples according to the behavioural reports from Hatching's Triage.

5. Discussion

Our research used different techniques previously utilised as standalone for malware analysis to formulate a standardised approach to malware identification. Even though the performance issues affect symbolic execution, we completed a reliable feature extraction from execution traces at scale. TLSH clustering has proved to be a reliable technique to prevent similar binaries from being repeatedly analysed by a cost-consuming procedure. More concretely, in a real-world integration, one would first compute the TLSH of the binary and try to determine whether it belongs to an existing cluster. If the sample does not belong to a cluster, then it has to undergo symbolic execution. Once the API calls are extracted, the features are extracted, and the sample is classified accordingly. Evidently, as in the experiments, this approach significantly limits the number of symbolic executions that have to be performed, especially in the case of dealing with specific malware campaigns. Even with the applied sampling process, we managed to produce accurate classification results. This can further challenge the malware author, as new derivations of an existing malware are highly more likely to be identified as malicious. Despite the plethora of analysis frameworks and methodologies in the literature, at the time of writing, symbolic execution was the only way we could have complete coverage of the executable's inner workings.

Notably, our automated pipeline allows us to extract essential features from symbolic execution, enabling us to conduct effective classification with a fraction of the resources and time compared to traditional and commercial sandboxed solutions. While researchers often concentrate on analysing relatively compact datasets, dedicating several days to studying highly specific behaviours, security companies prioritise the implementation of fully automated malware analysis pipelines. These pipelines are designed to efficiently process hundreds of thousands of samples daily, emphasising scalability and the accumulation of generic behavioural data. Companies typically run samples for an average duration spanning from 30 s to 4 min. This choice is rationalised by the necessity to analyse a large volume of new malware samples

[1] https://tria.ge/.

daily, highlighting the importance of achieving scalability in their operations (Küchler et al., 2021). On average, our pipeline would analyse a sample in 31 s, proving its efficiency. Due to the capacity of our pipeline to conduct dynamic analysis within a logically isolated memory environment, the redundancy associated with integrating additional sandbox systems is alleviated.

Our diverges from the existing literature in several significant ways. In the work of Sebastio et al. (2020), authors explored symbolic execution for malware analysis, focusing on the technical aspects of symbolic execution without integrating clustering techniques to handle large datasets efficiently. Similarly, Namani and Khan (2020) investigated symbolic execution's application in malware analysis, emphasising its effectiveness but not addressing the preprocessing step to reduce redundant samples. Our approach extends these foundations by incorporating TLSH clustering, a novel application in this context, to preprocess and deduplicate malware samples before symbolic execution. This integration not only enhances the efficiency of the analysis process but also ensures that the symbolic execution phase focuses on unique and representative samples, thus overcoming the path explosion issue more effectively. Additionally, our methodology diverges by emphasising the extraction of relevant features from the execution traces for use in machine learning-based malware classification, which provides a structured and automated approach to malware detection that was slightly explored in the aforementioned studies. By addressing the limitations of both symbolic execution and dataset redundancy, our work presents a comprehensive and practical solution that advances the state of the art in malware analysis.

As it can be observed in Table 3, our results are at least comparable with the ones of other researchers using dynamic analysis for malware detection in terms of precision, accuracy, and F1 score. Nevertheless, two key advantages of our methodology illustrate that this direction can achieve even better results. The first and most obvious is that our pipelines do not require a devoted virtual machine to execute all binaries. On the contrary, our pruning method allows us to reduce the samples that have to be analysed, but in the meantime, the execution, on average, requires a fraction of the time needed for a sandbox. For instance, in our pipeline, a symbolic execution required, on average, 30 s, while for a sandbox execution, that would be at least two minutes. Therefore, our methodology requires far less computational resources to achieve the same results.

Beyond comparing with purely academia works, which in most cases are performed in very constrained environments, we used real-world samples from many families and compared with a commercial malware sandbox, as seen in Section 4.3. Our approach is proved to be far better in terms of accuracy, precision, and resource allocation than such a sandbox, illustrating that symbolic allocation is a viable and complementary approach in dynamic malware analysis, even on scale. To this end, when dynamic analysis in sandboxes fails but static analysis shows that a sample is malicious, malware analysts should consider an automated symbolic execution pipeline to complement the missing traces from their analysis.

Moreover, much of the literature in Table 3 does not use only features from dynamic analysis but also exploits static features. Clearly, the latter introduces further information that can be used to train the underlying models better. However, we have opted to study symbolic execution independently to determine how much it can offer in large automated pipelines and what processing savings it could imply. Thus, we only use TLSH for filtering similar samples. Nevertheless, we anticipate that adding static features to the machine learning models can improve our measurements without overfitting.

5.1. Limitations

While we claim that symbolic execution is a great option for thorough sample analysis, there are several restrictions and limits. First, because of its maturity, no file or operating system can be arbitrarily analysed. In fact, we only employ a portion of the EXE files in our tests. More precisely, we only retained the executable, non-dynamic libraries, and Windows PE files. This decision was made because symbolic execution is more straightforward for unmanaged code executables as they compile directly to machine code and are run by the operating system. Other file types, such as Windows PE files built on the .NET platform, might cause problems with the symbolic execution at this time. As a result, the demands of arbitrary symbolic execution cannot be met by this analytical technique. Additionally, we had to exclude Dynamic-link libraries since they cannot be directly run without per-file orchestration. An entry point (typically, the main function) is necessary to execute a PE file. Angr, like many community-powered frameworks, has not yet implemented and fine-tuned all system-dependent APIs. As a result, API-complex applications fail to finish full symbolic execution.

Evidently, executing all possible programme pathways symbolically does not scale to huge programmes. The number of potential pathways in a programme rises exponentially with programme size, and in the case of systems with unbounded loop iterations, it can potentially be infinite (Anand et al., 2008). Path-finding techniques are commonly used to enhance code coverage (Ma et al., 2011), reduce execution time by parallelising independent pathways (Staats and Păsăreanu, 2010), or merge related paths to solve the path explosion problem (Kuznetsov et al., 2012). Veritesting is an example of merging since it "*uses static symbolic execution to increase the effect of dynamic symbolic execution*" (Avgerinos et al., 2014).

Symbolic execution is often hampered by limitations such as path explosion, inaccuracies in API modelling, and slow analysis speeds. Our approach addresses these challenges by integrating TLSH clustering to preprocess and reduce the dataset size, thereby focusing symbolic execution on fewer, more representative samples. The latter reduces the computational burden and mitigates path explosion. Additionally, by setting a controlled time budget for symbolic execution, we balance thorough analysis with practical time constraints, ensuring faster and more efficient processing. Finally, we have to consider that when the same memory region can be accessed using various names, symbolic execution becomes more complex (aliasing). Since aliasing cannot always be identified statically, the symbolic execution engine cannot understand that changing the value of one variable also alters the value of the other (DeMillo et al., 1991).

6. Conclusions

Early detection and automated malware analysis are becoming increasingly important as malware design and dissemination continue to progress. For example, executing a binary in a sandbox environment necessitates the reproduction of a real user host in a virtual machine, hooking all calls and monitoring all its interactions in the network, file system level, API and system calls, and so on. As a result, significant resources must be allocated to executing a single binary over an extended period. Such techniques cannot reach this rate when the total number of malware samples in Windows continuously increases, with a magnitude above 800 million (as of January 2024) (av-atlas.org, 2024). Investigating various feature selection approaches and their effectiveness in the context of symbolic execution in malware identification is a relevant research topic we want to pursue. As shown in this article, we introduce a novel pipeline that provides both accurate outcomes and a significant reduction in resource allocation compared to the state of the art and practice, as we analyse samples in a fraction of the time required by sandbox techniques. In addition, the interpretability and readability of the results can provide further insight into the elaboration of effective AI-based malware detection models (Casino et al., 2022).

Our future research will focus on the dynamic update of these models through adaptive processes, as in cognitive security scenarios (Huang and Zhu, 2023) or by using generative AI (Feffer et al., 2024). For instance, one could automate the creation of pipelines that adjust their input variables and performance (e.g., by using federated

Table 3
Comparison with state of the art. **Notation:** Scope: (A)ndroid, (W)indows. Method: A: https://www.apimonitor.com/ C: Cuckoo, D: Detours (Brubacher, 1999) M: Maleur, E: Ether (Dinaburg et al., 2008), SA: Static Analysis, SB: Sandbox, I: Imaging, ACR: API calls reports, SE: Symbolic Execution. Dataset: We report the number of (M)alicious and (B)enign samples that were used. Classification: (B)inary, (M)ulticlass and (#families). The latter classifies samples into worms, trojans, droppers, etc.

Ref.	Scope	Method	Dataset	Classification	Results
Qiao et al. (2014)	W	C&M	3131M	M(24)	F1 between 0.909 and 0.95
Uppal et al. (2014)	W	A	120M/150B	B	Accuracy of 0.985
Naval et al. (2015)	W	E	1209M/1316B	B	Accuracy of 0.954
Ki et al. (2015)	W	D	23,080M/114B	B	Accuracy of 0.998
Tang and Qian (2019)	W	C	9000M	M(9)	TPR, precision, recall and F1 are all above 0.99, while the FPR is below 0.01
Catak et al. (2020)	W	C	7107M	B & M	F1 score of 0.47 in the multiclass setup, and F1 scores between 0.27 and 0.83 in the binary classification according to different malware types.
Ficco (2021)	W	C	4960M/1200B	B	Best Accuracies reported between 0.954 and 0.989.
Xue et al. (2019)	W	I	174,607M	M(63)	Accuracy of 0.988. F1-score is 0.988
Salehi et al. (2017)	W	SB	3009M/1359B	B	Accuracy of 0.981
Han et al. (2019)	W	SA & C	5567M/1174B	B & M(5)	Accuracy of 0.943 for classification and 0.978 for detection
Amer et al. (2021)	W & A	ACR	51,938M/42,783B	B	An average accuracy of 0.997 and 0.977
Chen et al. (2022)	W	C	51,707M/20,000B	B	Accuracy of 0.991
Amer and Zelinka (2020)	W	ACR	30,658M/21,422B	B	An average precision of 0.990 with an average FPR of 0.010, and average FNR of 0.010
Our work	W	SE	14,761M/1531B	B & M	F1-score of 0.99 for malware samples and over 0.93 for benign

learning over models that use the most relevant features) according to the available computing resources and the collected features, providing effective strategies to fight persistent malware campaigns. Further research efforts on improving symbolic execution, to improve path coverage and path prioritisation, along with better resource allocation, can further improve the efficiency of our proposal. For instance, the development of advanced heuristics and pruning techniques to address the exponential growth in the number of execution paths or at least reuse results from previous runs can drastically boost the performance of symbolic execution (Yi et al., 2018; He et al., 2021). The handling of complex or dynamic data structures, often used by malware to hinder analysis, is another promising research direction (Borzacchiello et al., 2019; Schemmel et al., 2022; Trabish and Rinetzky, 2020). On the other hand, given the efficacy of this methodology, evasion methods against symbolic execution, e.g., deliberately increasing the possible execution paths or decreasing the prioritisation of the truly malicious execution paths, could be a fruitful research track and lead to an even more robust solution.

CRediT authorship contribution statement

Vasilis Vouvoutsis: Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Data curation. **Fran Casino:** Writing – review & editing, Writing – original draft, Visualization, Validation, Resources, Investigation, Funding acquisition, Data curation. **Constantinos Patsakis:** Writing – review & editing, Writing – original draft, Visualization, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

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

Acknowledgements

This work was supported by the European Commission under the Horizon Europe Programme, as part of the projects LAZARUS (Grant Agreement no. 101070303), SAFEHORIZON (Grant Agreement no. 101168562), and CyberSecPro (Grant Agreement no 101083594). This work was partially supported by Ministerio de Ciencia, Innovación y Universidades, Gobierno de España (Agencia Estatal de Investigación, Fondo Europeo de Desarrollo Regional -FEDER-, European Union) under the research grant PID2021-127409OB-C33 CONDOR. Fran Casino was supported by the Spanish Ministry of Science and Innovation under the "Ramón y Cajal" programme (RYC2023-044857-I), and by AGAUR with the project ASCLEPIUS (2021SGR-00111). The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

Data availability

Data will be made available on request.

References

abuse.ch, 2021. MalwareBazaar. https://bazaar.abuse.ch/. (Accessed 08 April 2024).
Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D., 2019. Malware dynamic analysis evasion techniques: A survey. ACM Comput. Surv. 52 (6), 1–28.
Agner, F., 2014. Calling conventions for different c++ compilers and operating systems. http://www.agner.org/optimize/calling_conventions.pdf.
Alkhateeb, E., Ghorbani, A., Habibi Lashkari, A., 2023. A survey on run-time packers and mitigation techniques. Int. J. Inf. Secur. 1–27.
Amer, E., Zelinka, I., 2020. A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence. Comput. Secur. 92, 101760.
Amer, E., Zelinka, I., El-Sappagh, S., 2021. A multi-perspective malware detection approach through behavioral fusion of api call sequence. Comput. Secur. 110, 102449.
Anand, S., Godefroid, P., Tillmann, N., 2008. Demand-driven compositional symbolic execution. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 367–381.
Apostolopoulos, T., Katos, V., Choo, K.-K.R., Patsakis, C., 2021. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. Future Gener. Comput. Syst. 116, 393–405.
av-atlas.org, 2024. Av-atlas.org. https://portal.av-atlas.org/malware/statistics. (Accessed 14 January 2024).
Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D., 2014. Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1083–1094.

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 1–39.
Bertrand Van Ouytsel, C.-H., Legay, A., 2022. Malware analysis with symbolic execution and graph kernel. In: Nordic Conference on Secure IT Systems. Springer, pp. 292–310.
Bonfante, G., Nogues, J.O., 2016. Function classification for the retro-engineering of malwares. In: International Symposium on Foundations and Practice of Security. Springer, pp. 241–255.
Borzacchiello, L., Coppa, E., D'Elia, D.C., Demetrescu, C., 2019. Memory models in symbolic execution: key ideas and new thoughts. Softw. Test. Verif. Reliab. 29 (8), http://dx.doi.org/10.1002/STVR.1722.
Brubacher, D., 1999. Detours: Binary interception of win32 functions. In: Windows NT 3rd Symposium (Windows NT 3rd Symposium).
Bulazel, A., Yener, B., 2017. A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web. In: Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium. ACM, ACM, New York, NY, USA, p. 2.
Cadar, C., Dunbar, D., Engler, D.R., et al., 2008a. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, Vol. 8. pp. 209–224.
Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R., 2008b. Exe: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12 (2), 1–38.
Cadar, C., Sen, K., 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56 (2), 82–90.
Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E., 2012. A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 122–132.
Casino, F., Dasaklis, T.K., Spathoulas, G.P., Anagnostopoulos, M., Ghosal, A., Borocz, I., Solanas, A., Conti, M., Patsakis, C., 2022. Research trends, challenges, and emerging topics in digital forensics: A review of reviews. IEEE Access 10, 25464–25493.
Casino, F., Patsakis, C., Solanas, A., 2019. Privacy-preserving collaborative filtering: A new approach based on variable-group-size microaggregation. Electron. Commer. Res. Appl. 38, 100895.
Casino, F., Totosis, N., Apostolopoulos, T., Lykousas, N., Patsakis, C., 2023. Analysis and correlation of visual evidence in campaigns of malicious office documents. Digit. Threats: Res. Pract. 4 (2), 1–19.
Catak, F.O., Yazı, A.F., Elezaj, O., Ahmed, J., 2020. Deep learning based sequential model for malware analysis using windows exe api calls. PeerJ Comput. Sci. 6, e285.
Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D., 2012. Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. IEEE, pp. 380–394.
Chen, X., Hao, Z., Li, L., Cui, L., Zhu, Y., Ding, Z., Liu, Y., 2022. Cruparamer: Learning on parameter-augmented api sequences for malware detection. IEEE Trans. Inf. Forensics Secur. 17, 788–803.
Chipounov, V., Kuznetsov, V., Candea, G., 2011. S2e: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Not. 46 (3), 265–278.
Chipounov, V., Kuznetsov, V., Candea, G., 2012. The s2e platform: Design, implementation, and applications. ACM Trans. Comput. Syst. (TOCS) 30 (1), 1–49.
Christodorescu, M., Jha, S., Kruegel, C., 2007. Mining specifications of malicious behavior. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 5–14.
Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H., 2005. Malware Normalization. Tech. Rep., University of Wisconsin-Madison Department of Computer Sciences.
DeMillo, R.A., Offutt, A.J., et al., 1991. Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. 17 (9), 900–910.
Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. pp. 51–62.
Egele, M., Scholte, T., Kirda, E., Kruegel, C., 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. (CSUR) 44 (2), 1–42.
Ester, M., Kriegel, H., Sander, J., Xu, X., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Simoudis, E., Han, J., Fayyad, U.M. (Eds.), Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. KDD-96, AAAI Press, Portland, Oregon, USA, pp. 226–231.
Feffer, M., Sinha, A., Lipton, Z.C., Heidari, H., 2024. Red-teaming for generative ai: Silver bullet or security theater? arXiv preprint arXiv:2401.15897.
Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F., 2015. Efficient and robust automated machine learning. Adv. Neural Inf. Process. Syst. 28.
Ficco, M., 2021. Malware analysis by combining multiple detectors and observation windows. IEEE Trans. Comput. 71 (6), 1276–1290.
Geng, J., Wang, J., Fang, Z., Zhou, Y., Wu, D., Ge, W., 2024. A survey of strategy-driven evasion methods for pe malware: Transformation, concealment, and attack. Comput. Secur. 137, 103595.
Gibert, D., Mateu, C., Planes, J., 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. J. Netw. Comput. Appl. 153, 102526.
Griffin, K., Schneider, S., Hu, X., Chiueh, T.-c., 2009. Automatic generation of string signatures for malware detection. In: International Workshop on Recent Advances in Intrusion Detection. Springer, pp. 101–120.
Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., Mao, L., 2019. Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233.
He, J., Sivanrupan, G., Tsankov, P., Vechev, M.T., 2021. Learning to explore paths for symbolic execution. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (Eds.), CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15-19, 2021. ACM, pp. 2526–2540.
Huang, L., Zhu, Q., 2023. Cognitive Security: A System-Scientific Approach. Springer Nature.
Joyce, R.J., Bilzer, K., Burke, S., 2019. Malware attribution using the rich header.
Kemkes, Phillip, 2019. Evaluation of current virtual machine detection methods. In: 14. GI FG SIDAR Graduierten-Workshop über Reaktive Sicherheit. p. 15.
Kephart, J.O., 1994. Automatic extraction of computer virus signatures. In: Proc. 4th Virus Bulletin International Conference, Abingdon, England, 1994. pp. 178–184.
Ki, Y., Kim, E., Kim, H.K., 2015. A novel approach to detect malware based on api call sequence analysis. Int. J. Distrib. Sens. Netw. 11 (6), 659101.
Krishnamoorthy, S., Hsiao, M.S., Lingappan, L., 2010. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In: 2010 19th IEEE Asian Test Symposium. IEEE, pp. 59–64.
Küchler, A., Mantovani, A., Han, Y., Bilge, L., Balzarotti, D., 2021. Does every second count? Time-based evolution of malware behavior in sandboxes. In: NDSS.
Kuss, O., 2002. Global goodness-of-fit tests in logistic regression with sparse data. Stat. Med. 21 (24), 3789–3801.
Kuznetsov, V., Kinder, J., Bucur, S., Candea, G., 2012. Efficient state merging in symbolic execution. ACM SIGPLAN Not. 47 (6), 193–204.
Li, J., Zhang, T., Luo, W., Yang, J., Yuan, X.-T., Zhang, J., 2016. Sparseness analysis in the pretraining of deep neural networks. IEEE Trans. Neural Netw. Learn. Syst. 28 (6), 1425–1438.
Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M., 2011. Directed symbolic execution. In: International Static Analysis Symposium. Springer, pp. 95–111.
Mandiant, 2014. Tracking malware with import hashing. https://www.mandiant.com/resources/blog/tracking-malware-import-hashing, (Accessed 15 February 2024).
Microsoft, 2022. MSDN: Microsoft c runtime library (CRT) reference. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference?view=msvc-160. (Accessed 08 April 2024).
Microsoft, 2022. MSDN: Working with strings. https://docs.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings. (Accessed 08 April 2024).
Mohanta, A., Saldanha, A., 2020. Windows internals. In: Malware Analysis and Detection Engineering. Springer, pp. 123–162.
MongoDB, 2009. Mongodb. https://github.com/mongodb/mongo. (Accessed 1 April 2024).
Moser, A., Kruegel, C., Kirda, E., 2007. Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference. ACSAC 2007, IEEE, pp. 421–430.
Mow, W.-L., Huang, S.-K., Hsiao, H.-C., 2022. Laeg: Leak-based aeg using dynamic binary analysis to defeat aslr. In: 2022 IEEE Conference on Dependable and Secure Computing. DSC, IEEE, pp. 1–8.
Naik, N., Jenkins, P., Savage, N., Yang, L., Boongoen, T., Iam-On, N., 2020. Fuzzy-import hashing: A malware analysis approach. In: 2020 IEEE International Conference on Fuzzy Systems. FUZZ-IEEE, IEEE, pp. 1–8.
Namani, N., Khan, A., 2020. Symbolic execution based feature extraction for detection of malware. In: 2020 5th International Conference on Computing, Communication and Security. ICCCS, IEEE, pp. 1–6.
Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. pp. 1–7.
Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M., 2015. Employing program semantics for malware detection. IEEE Trans. Inf. Forensics Secur. 10 (12), 2591–2604.
Oliver, J., Ali, M., Hagen, J., 2020. Hac-t and fast search for similarity in security. In: 2020 International Conference on Omni-Layer Intelligent Systems. COINS, IEEE, pp. 1–7.
Oliver, J., Ali, M., Liu, H., Hagen, J., 2021. Fast clustering of high dimensional data clustering the malware bazaar dataset. https://tlsh.org/papersDir/n21_opt_cluster.pdf.
Oliver, J., Cheng, C., Chen, Y., 2013. Tlsh–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, pp. 7–13.
Ono, J.P., Castelo, S., Lopez, R., Bertini, E., Freire, J., Silva, C., 2020. Pipelineprofiler: A visual analytics tool for the exploration of automl pipelines. IEEE Trans. Vis. Comput. Graphics 27 (2), 390–400.
Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L., 2019. Dynamic malware analysis in the modern era—a state of the art survey. ACM Comput. Surv. 52 (5), 1–48.
Oyama, Y., 2018. Trends of anti-analysis operations of malwares observed in api call logs. J. Comput. Virol. Hack. Tech. 14 (1), 69–85.
Patsakis, C., Arroyo, D., Casino, F., 2024. The malware as a service ecosystem. In: Gritzalis, D., Choo, K.-K.R., Patsakis, C. (Eds.), Malware – Handbook of Prevention and Detection. Springer, (Chapter 16).

Qiao, Y., Yang, Y., He, J., Tang, C., Liu, Z., free, Cbm:., 2014. Cbm: free automatic malware analysis framework using api call sequences. In: Knowledge Engineering and Management: Proceedings of the Seventh International Conference on Intelligent Systems and Knowledge Engineering, Beijing, China, Dec 2012. ISKE 2012, Springer, pp. 225–236.

Rokach, L., 2010. Ensemble-based classifiers. Artif. Intell. Rev. 33 (1), 1–39.

Rudd, E., Rozsa, A., Gunther, M., Boult, T., 2017. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. IEEE Commun. Surv. Tutor. 19 (2), 1145–1172.

Salehi, Z., Sami, A., Ghiasi, M., 2017. Maar: Robust features to detect malicious activity based on api calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102.

Saudel, F., Salwan, J., 2015. Triton: A dynamic symbolic execution framework. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications, SSTIC, France, Rennes. pp. 31–54.

Schemmel, D., Büning, J., Busse, F., Nowack, M., Cadar, C., 2022. A deterministic memory allocator for dynamic symbolic execution. In: Ali, K., Vitek, J. (Eds.), 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. In: LIPIcs, vol. 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:26. http://dx.doi.org/10.4230/LIPICS.ECOOP.2022.9.

Sebastio, S., Baranov, E., Biondi, F., Decourbe, O., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J., 2020. Optimizing symbolic execution for malware behavior classification. Comput. Secur. 93, 101775.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al., 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 138–157.

Sihwail, R., Omar, K., Ariffin, K.Z., 2018. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. Int. J. Adv. Sci. Eng. Inf. Technol. 8 (4–2), 1662–1671.

Staats, M., Păsăreanu, C., 2010. Parallel symbolic execution for structural test generation. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. pp. 183–194.

Tang, M., Qian, Q., 2019. Dynamic api call sequence visualisation for malware classification. IET Inf. Secur. 13 (4), 367–377.

Trabish, D., Rinetzky, N., 2020. Relocatable addressing model for symbolic execution. In: Khurshid, S., Pasareanu, C.S. (Eds.), ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020. ACM, pp. 51–62. http://dx.doi.org/10.1145/3395363.3397363.

Uppal, D., Sinha, R., Mehra, V., Jain, V., 2014. Malware detection and classification based on extraction of api sequences. In: 2014 International Conference on Advances in Computing, Communications and Informatics. ICACCI, IEEE, pp. 2337–2342.

Vasan, D., Alazab, M., Wassan, S., Safaei, B., Zheng, Q., 2020. Image-based malware classification using ensemble of cnn architectures (imcec). Comput. Secur. 92, 101748.

Vouvoutsis, V., Casino, F., Patsakis, C., 2022. On the effectiveness of binary emulation in malware classification. J. Inf. Secur. Appl. 68, 103258.

Xue, D., Li, J., Lv, T., Wu, W., Wang, J., 2019. Malware classification using probability scoring and machine learning. IEEE Access 7, 91641–91656.

Yi, Q., Yang, Z., Guo, S., Wang, C., Liu, J., Zhao, C., 2018. Eliminating path redundancy via postconditioned symbolic execution. IEEE Trans. Softw. Eng. 44 (1), 25–43. http://dx.doi.org/10.1109/TSE.2017.2659751.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., et al., 2016. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer, pp. 165–187.

Ziegler, J., 2021. Edge of the Art in Vulnerability Research Version 4 of 4. Tech. rep., Two Six Labs.

Zigomitros, A., Casino, F., Solanas, A., Patsakis, C., 2020. A survey on privacy properties for data publishing of relational data. IEEE Access 8, 51071–51099. http://dx.doi.org/10.1109/ACCESS.2020.2980235.

Vasilis Vouvoutsis holds a B.Sc. in Applied Mathematics from the University of Crete and an M.Sc. in Security Management Engineering from University of Piraeus. Currently, he works as a software and security engineer at Viva.com and is a Ph.D. candidate at the University of Piraeus. His main research interests are focused on security engineering, malware analysis, and the use of machine learning in cyber security.

Fran Casino is a postdoctoral researcher at the Department of Computer Engineering and Mathematics at Rovira i Virgili University, Spain, and Athena Research Center, Greece. He obtained his B.Sc. degree in Computer Science and his M.Sc. in Computer Security and Intelligent Systems, both from URV. He holds a PhD in Computer Science from URV with honours (A cum laude) and the best dissertation award. He completed several stays in international research institutions such as ISCTE-IUL and the University of Piraeus. His research has an interdisciplinary focus and combines several knowledge areas with disruptive technologies. Some keywords related to his research are pattern recognition, cognitive security, privacy protection, cybercrime and digital investigations, recommender systems, supply chain, and blockchain. He has authored more than 60 publications in peer-reviewed international conferences and journals. He was listed in the World's Top 2% most influential scientists in his field by Stanford University.

Constantinos Patsakis holds a B.Sc. in Mathematics from the University of Athens, Greece and an M.Sc. in Information Security from Royal Holloway, University of London. He obtained his PhD in cyber security from the University of Piraeus. His main research areas include cryptography, malware, security, privacy, and cybercrime. He has participated in several national and European R&D projects. Additionally, he has worked as a researcher at the UNESCO Chair in Data Privacy, at Trinity College, Dublin, Ireland, and the Luxembourg Institute of Science and Technology.