Computers & Security 142 (2024) 103856

Contents lists available at ScienceDirect

Computers & Security

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

Malware similarity and a new fuzzy hash: Compound Code Block Hash (CCBHash)

Jose A. Onieva [a,*], Pablo Pérez Jiménez [b], Javier López [a]

[a] Network, Information and Computer Security (NICS) Lab, University of Malaga, Spain
[b] VirusTotal, Malaga, Spain

ARTICLE INFO

Keywords:
Hashes
Fuzzy hashes
Code similarity
Malware

ABSTRACT

In the last few years, malware analysis has become increasingly important due to the rise of sophisticated cyberattacks. One of the objectives of this cybersecurity branch is to find similarities between different files or functions used by malware programmers, thus allowing malware detection, classification and even attribution in a timely manner. In this article we survey the state of the art in this area, reviewing the different techniques that can be applied to the field, with the objective of studying similarity, and therefore detecting, classifying and attributing malware samples. We have developed a fuzzy hash capable of characterizing malware by generating an easily comparable and storable signature of its functions. Since our goal is to detect these similarities in huge amounts of data within a reasonable time-frame, the size of the hash must be limited while retaining as much information as possible.

1. Introduction

In current cybersecurity threats, there exists a persistent endeavor by malevolent entities to elude detection mechanisms. For instance, the deployment of polymorphic and metamorphic malware (Rad et al., 2012) by cybercriminals exploits the capacity of even minor alterations in binary code or script to circumvent conventional antivirus software. For this reason, traditional signature-based methods face challenges in identifying variants and polymorphic strains. Threat actors tailor their malicious payloads to specific target organizations, thereby enhancing the likelihood of infiltrating and traversing an entire corporate network and orchestrating data exfiltration with minimal traces of their activities. The clandestine market thrives on an array of tools, including malware builders, trojanized versions of legitimate applications, and assorted services facilitating the deployment of highly elusive malware.

The escalating volume of threats in the wild brings up a shift from evaluating security solutions solely based on their proficiency in detecting unique malware instances. Instead, contemporary security frameworks must embody resilient architectures capable of defending against both prevailing and future attack vectors. It is clear to the community that AI (Artificial Intelligence) will play a key role in this mission, but the needed data can be pre-processed or enriched so as to give even faster responses or to enrich already available data with similarity information.

The underlying objective of characterizing the similarity of samples is to elevate the overall accuracy in classifying malware and diminish the temporal gap between malware release and subsequent detection and classification. Its proficiency in recognizing and obstructing malware at first encounter holds critical significance in safeguarding against a diverse spectrum of threats, including sophisticated cyberattacks. This objective has been displayed as an important tool already avoiding newly created attacks (Lazo, 2021).

This is of paramount importance. For instance, over the course of the first six months in 2022, Fortinet documented 10,666 ransomware variants across its platform (Labs, 2022). Obviously, not each of them belongs to a different malware family. On the contrary, all of these variants belong to about a hundred malware families. In each family, there are similar features that characterize all their samples. And these similar features are key to catch as soon as possible new samples belonging to known malware families.

There are several tools that allow a large number of samples to be simultaneously compared trying to find similarities among them. Similarly, other tools take two files and analyze their assembly code in order to detect whether there are comparable functions in both samples. How-

* Corresponding author.
E-mail addresses: onieva@uma.es (J.A. Onieva), pablopj@virustotal.com (P. Pérez Jiménez), javierlopez@uma.es (J. López).

https://doi.org/10.1016/j.cose.2024.103856
Received 26 January 2024; Received in revised form 24 March 2024; Accepted 15 April 2024
Available online 21 April 2024

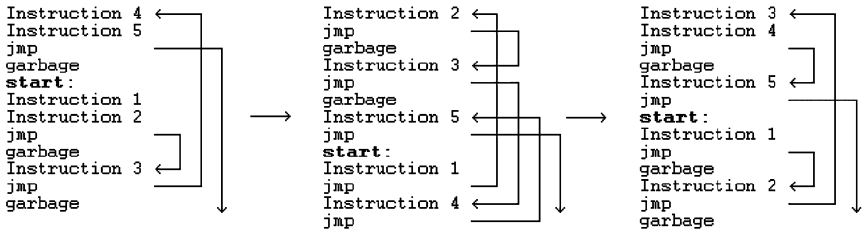


Fig. 1. Code transposition (Ször and Ferrie, 2001).

ever, if we want to find similar code functions in a large number of samples, the number and efficiency of these tools is reduced drastically.

Our aim is to create a signature for each malware sample's assembly code function, in such a way that for each function, similar functions can be detected in a database (e.g. VirusTotal database) storing a very large number of functions. In addition, for the same sample, the generated signatures can be concatenated to form a single hash that identifies the whole file. For this purpose, we used fuzzy hashing strategies, where a final hash is composed of the concatenation of partial hashes. In this way, we have the opportunity to detect those functions that are reused in different files, being able to find samples with similar functionalities, different versions of the same malware or that have been developed by the same actors.

We do not aim at providing machine learning (ML) techniques to search for similarities among samples, but rather to produce a similarity signature that can be stored, integrated with malware (or fragments of it) and ready to be used for the search of similarity among petabytes of information.

In this article we survey, analyze and categorize the theoretical and practical foundations of malware similarity that we esteem more relevant for the design of a new algorithm. All of this effort has been based in related and referenced work. This provides two differentiated blocks in the corpus text of this article. In the first block, more concretely we differentiate between techniques in Section 2, in which we present techniques used by malware authors in order to thwart detection as well as techniques used for general similarity search; tools in section 4 in which we provide a brief overview of existing implementations for malware similarity; and metrics in section 3 in which comparison metrics are presented and categorized.

In the second block of contributions we detailed our design, implementation and analysis of CCBHash. More concretely, in section 5 we propose the design and implementation of the second iteration of a new tool (the first one was published in (Jiménez et al., 2022)) that fulfills our aim. In section 6 we detailed the experiments carried out in order to validate our design.

In this paper, we present several contributions to the field of malware similarity that to the best of our knowledge is not present in similar works:

- We identify three types of components to work with when treating with malware similarity: techniques, metrics and tools.
- We categorize these components and analyze them from the malware investigation perspective, highlighting those more accurate and valid for malware similarity
- We use these theoretical and practical foundations to design and implement a new fuzzy hash.
- We extensively analyze and compare this new hash with similar approaches, identifying its main advantages.

Additionally, we discuss the implications of these contributions and their potential impact in malware characterization in Section 7.

2. Techniques

As we already mentioned in section 1, it is key for malware developers and campaigns to reduce the probability of detection. With this objective, these actors use known techniques in order to change malware code while retaining its functionality. Among the commonly used strategies (Eresheim et al., 2017; Caviglione et al., 2020; Gao et al., 2014), the most common ones are:

- **Instruction replacement**. Some instructions are replaced by other equivalent ones, generating a different code but with the same functionality. For instance, a multiplication can be reduced to a number of additions.
- **Instruction reordering**. The attacker changes the order of certain instructions, obtaining an apparently different sample but with the same functionality. Even different functions can be called into the code in different order without affecting the final functionality.
- **Junk code insertion**. Sometimes the attacker adds multiple operations, or even NOP statements,[1] which do not change the functionality of the malware but make it more difficult for defenders to understand. This not only obstructs the malware analyst job, but also allows for the generation of different code for the same malware family. These operations may be also inverse operations (i.e. one cancels the other), and in general any operation that does change functionality (e.g. setting to zero a register that is not used).
- **Register reassignment**. Although this action, on some occasions, is not caused by the attacker, we must bear in mind that two identical samples may have different information depending on the system for which they are compiled and linked. That is, compiling and linking of the same source code can generate different final binary code depending on compiler and linker options.
- **Code transposition**. Attackers sometimes change the Control Flow Graph (CFG, see section 2.2) of malware, introducing unnecessary unconditional jumps. That is, the instructions are rearranged differently in the code but are executed in the same order making use of these jumps. For example, this is achieved by splitting a function into smaller functions, or by creating functions whose only instruction is a function call, thus changing the schema of executed functions. See Fig. 1 for a graphic example.
- **Insertion of Conditional jumps**. Like in the previous strategy, the attacker makes changes to the CFG but, in this case, through conditional jumps. Normally, the new code blocks introduced after these jumps are not executed, or else they are in charge of redirecting the flow towards the desired direction.

Our hash design needs to take into account these techniques in order to minimize their impact. In this way, we will be able to detect what are the really important characteristics that different samples that behave in the same way have in common.

It is worth noting that there are additional techniques, such us encrypting or program packing that helps in avoiding detection. In such

[1] The assembler nop instruction commands the processor to do an empty operation.

Table 1
Techniques comparison.

Technique	Effectiveness	Complexity	Detectability
Instruction Replacement	High	Moderate	Low
Instruction Reordering	Moderate	Moderate	Low
Junk Code Insertion	High	High	Moderate
Register Reassignment	Low	Low	Low
Code Transposition	Moderate	Moderate	Moderate
Insertion of Cond. Jumps	Moderate	High	Moderate
Code Obfuscation	High	High	High

cases, the malware generally has a decryption/unpacking engine (and optionally a encrypting engine if the malware wants to mutate after the infection process), encrypted/packed code and, if encrypted, a decryption key embedded in its code. The malware uses its engine to unpack/decrypt the code in memory before execution. As similarity search in malware focuses on functionality, and functionality is impossible to be extracted from encrypted/packed samples, all the techniques assume that, if needed, these samples are preprocessed in order to obtain the code even if this is further obfuscated. Unfortunately in some cases, this process needs to be performed manually (Monnappa, 2018).

These techniques present different degrees of *effectiveness*; that is, how well a particular technique achieves its intended goal of changing the code without altering its functionality, *complexity*; that is, the level of difficulty or intricacy involved in implementing and applying that particular technique and *detectability*; that is, the likelihood or ease with which the modified code can be detected or identified by tools, techniques, or human analysts. A comparison table for these techniques and criteria is provided in Table 1.

Taking this into consideration, the community has developed different techniques to improve detection among modifications of the same sample or to search for similarities among different pieces of software so as to identify analogous (even if partial) functionality which could denote partial re-utilization of code or even collaboration among different groups of malware developers. Note that both, the software and malware defense communities push forward this research, since the same techniques and algorithms can be used for software plagiarism and intellectual property protection. In the following we briefly survey some of these techniques used for malware similarity search.

2.1. N-grams

There are several solutions that use n-grams to compare different files. The n-grams are sliding windows of variable size, n in this case, where the goal is to store all possible windows of n elements for later comparison. These elements can be very diverse. As far as malware is concerned, these elements are usually bytes or assembler instructions, the latter allowing a higher lever of functionality comparison.

However, many of the existing studies have gone a step further and do not try to use only bytes or exact instructions, but do a little analysis of the input by filtering out those features that are more relevant. For example, there are solutions (Liu et al., 2017) where the elements used for the n-grams are the instruction opcodes without registers and operands. This method allows for a comparison focusing on the malware functionality, rejecting other characteristics. In fact, this instruction normalization process is generally applied in most of the solutions we have reviewed and can be better understood in Table 2.

Table 2
Normalization process.

Original instructions	Normalized instructions
push ebp	push reg
mov ebp, esp	mov reg, reg
sub esp, 0C0h	sub reg, value
push ebx	push reg
lea edi, [ebp+12]	lea reg, mem
mov ecx, 30h	mov reg, value

While the above solution may be interesting in many contexts, it also has some weaknesses. One of the main disadvantages is the accuracy required to be able to find similarities. It will be better understood with two examples. As we know, in many occasions the execution order of a group of instructions produces the same result. For example, when performing inverse operations (e.g. a consecutive addition and subtraction by a constant) the final result will be equivalent regardless of the order in which they are performed. In the same way, there are several opcodes in assembly code that are used to perform the same operations, being able to achieve identical goals with different patterns. In both cases, this technique will struggle to find similarities. Another limitation of short n-grams is the loss of long-range information within analyzed binaries, that is, significant relationships between distant code will not be detected.

To solve some of the existing problems with n-grams of opcodes, there are studies that introduce the concept of n-perms. The n-perms are a version of n-grams but using permutations. Thus, if there are several identical n-grams stored, the repeated ones are removed leaving only one n-perm of each class. This technique of eliminating repetitions is used in different similarity search techniques (see, for instance, (Kornblum, 2006)). In addition, n-perms are insensitive to execution order, so two n-grams containing the same elements in different order will be treated as the same n-perm.

Using n-perms results in higher similarity detection for files using the same opcodes. However, this higher level of abstraction can also lead to a higher number of false positives where two files with different functionalities use similar instructions. We thus arrive at a point where we must decide between two models. One where there is a lower but more accurate detection of positives, and another where detection is higher but the false positives rate may be increased.

2.2. Graphs

Control Flow Graphs store the execution flow that follows a given executable, usually at the assembly code level. Several disassemblers and decompilers (e.g. IDA Pro (Hex-Rays, 2022)) use this representation of the samples in order to ease their analysis.

As in all graphs, there are two types of elements: nodes and edges. The directed edges store the information about the existing jumps in the execution. Thus, each jump, conditional or unconditional, will be reflected with one or more edges. As for the nodes, they can store very diverse information. Some studies leave them practically empty (Bonfante et al., 2007), giving all the importance to the flow. However, in other solutions, such nodes are used to store more valuable information. For instance, in (Kruegel et al., 2006), authors proposed a fingerprint based on k-subgraphs of CFGs. For this generated subgraph, a fingerprint is calculated using an adjacency matrix which is then applied for searching similarities in polymorphic worms.

In an effort to enrich these graphs, (Yan et al., 2019) proposes the use of attributes to characterize each node, creating the Attributed CFG (ACFG). This approach tries to take the nodes generated by a tool like IDA Pro and creates a signature based on a vector of attributes for each of these nodes. The attributes used in this case are those listed in Table 3.

The use of ACFG is very interesting since it allows to store very diverse information in the nodes and at the same time it allows to combine this strategy with others. For example, (Li et al., 2019) proposes to use n-grams of the CFG, where the elements over which the windows advances are the nodes, which, in this work, are characterized by the number of incoming and outgoing edges, covering all possible paths of n consecutive nodes following the established flow. Another simpler technique is to use the CFG to sort the code and then take n-grams of opcodes.

Other authors (Bergeron et al., 2001) try to combine CFGs with subsystem API calls, creating a graph where the nodes are characterized by

Table 3
Some attributes to characterize nodes in CFG in (Yan et al., 2019).

Attribute type	Attribute description
From code sequence	# Numeric constants # Transfer instructions # Call instructions # Arithmetic instructions # Compare instructions # Mov instructions # Termination instructions # Data declaration instructions # Total instructions
From vertex structure	# Offspring, i.e., degree # Instructions in the vertex

the API calls made by the sample to the subsystem. This method, though advantageous for non-malicious executables, has serious drawbacks if we take into account that malware is many times created specifically not to use subsystem API calls.

Finally, there are solutions like those in (Mayrand et al., 1996; Kim et al., 2016; Krinke, 2001) that also use Data Flow Graphs (DFGs) and Program Dependence Graphs (PDGs), where a graph about the data dependency is stored (i.e. the processor registers used in the code). Although these approaches provide great results, they require high computation times (due to the computational complexity of extracting, finding and searching operations). Additionally, while an n-gram strategy needs linear computation time, any graph approach increases the cost to $O(n^3)$ (Liu et al., 2016).

Among the tools analyzed in this Section, there are two that particularly catch our attention, Machoc (Berre et al., 2022) and Machoke (Bogard, 2022). Both are very similar in functionality, but differ slightly in their design. These tools obtain the CFG of each function of an executable, calculate a hash code for each one and concatenate the hashes to form a fuzzy hash.

2.3. Fuzzy hashing

There are some works that study the precision of the aforementioned strategies (Song, 2014). However, there are other types of approaches with very interesting characteristics, such as fuzzy hashes. Fuzzy hashing emerges as a promising solution, allowing for the identification of similarities even in the presence of slight modifications. It retains the capacity to yield similar hashes for comparable inputs, crucial for identifying previously unseen malware with resemblances to already known instances.

A fuzzy hash is a type of hash that is used to summarize a sample into a relatively small signature and allows similarities to be detected between samples that are similar but not identical. Therefore, they are commonly used in malware detection. Fuzzy hashes differ from traditional hashes in that they do not seek to unambiguously identify a sample, but instead to summarize a sample into a signature that is small enough to be stored and compared quickly, and at the same time allows to identify similarities. There are different types of fuzzy hashing algorithms, such as the Block Based Hashing algorithms (BBH) like (Harbour, 2006), the Context Triggered Piece wise Hashing algorithms (CTPH) like (Kornblum, 2006), the Statistically-Improbable Features (SIF) like sdhash (Roussev, 2010), Block Based Rebuilding (BBR) like (Breitinger et al., 2013) and Locality Sensitive Hashing (LSH) like (Oliver et al., 2013). Many well-known tools use fuzzy hashes to find similarities between different files. However, none of these hashes are fully effective as it depends on the nature of the sample being treated. In VirusTotal (VirusTotal, 2022) we can find numerous samples where, depending on the fuzzy hash used, the similarity varies drastically.

While fuzzy hashing proves to be a valuable asset, it is not without challenges. Obfuscation techniques, false positives, and scalability concerns are among the key issues. An awareness of these challenges is imperative for cybersecurity professionals adopting fuzzy hashing as part of their analytical toolkit.

2.4. Similarity based on training and ML algorithms

As numerous machine learning algorithms have been developed for clustering and this, in turn, serves the purpose of grouping samples by similarity, there is a myriad of ML algorithms and techniques that can be used in the field of malware similarity.

Singular value decomposition (SVD) is a very popular linear algebra technique to break down a matrix into the product of a few smaller matrices. In fact, it is a technique that has many uses. One example is that we can use SVD to discover the relationship between items. This is done reducing the malware test and training samples to matrices that allow the generation of SVDs. The SVDs of malware samples are used for comparison using different metrics like the cosine or the Euclidean distance (see Section 3).

In (de Geus and Grégio, 2015), authors use Profile Hidden Markov Models (PHMM) to create statistical models using supervised learning. Inspired by DNA sequencing, the models are generated by applying a Multiple Sequence Alignment (PSA) algorithm to the sequence of actions executed by the malware sample inside a dynamic analysis environment, being able to identify similar behavior among different malware families by comparing the labeling results present in the confusion matrix. Before similarity calculation the possible behaviors performed by malware are encoded and a specific HMM allowing multiple observation sequences is applied. As the authors state, in their model, metamorphic malware that do permutation of the order of the actions executed to change its behavior can not be modeled well with PHMMs simply due to the fact that this kind of transformations is not common in a bioinformatics scenario of DNA Sequencing and thus PHMM states are no designed to model it. Nevertheless, other types of metamorphic behavior can successfully be classified (Attaluri et al., 2009). Authors have developed the code which can be accessed at (Baruque, 2015).

3. Similarity metrics

Once the samples have been reduced to some form of representation (strings, vectors, graphs, etc.), different similarity metrics can be applied in order to compare them. Any metric designed for calculating similarity can be used. There are many of them depending on the representation used. It is worth noting that some of these similarity metrics could be applied to detect similarity directly to code without the need for a previous representation of its characteristics. This is not however the best option, since when working with malware we would miss the semantic meaning of its functions.

3.1. Strings

Since some similarity techniques reduce the malware samples to (fuzzy) hashes and these, in turn, are sometimes[2] a classic string representation, all the metrics valid for string comparison can be used. There are many of them, the most common being:

- Edit distance: The edit distance between two strings defines the number of edit operations that must be performed to transform one string into the other. The operations can be weighted, that is, some of them may be given more importance than others when analyzing the difference between hashes. For instance, (Kornblum, 2006) defines four types of edit operations: insertion, deletion, substitution ans swaps and give more weight to the latter.

[2] Note that some string representations may be a digest representing sets or blocks and not necessarily encoded bytes.

- Normalized compression distance (NCD): It is used to quantify how similar two entities are by considering the compressibility of their concatenated representation compared to their individual compressibility. The NCD is derived from the idea that similar objects can be compressed into similar representations better suited for calculating their true information distance. Different compression algorithms can be used (e.g., gzip, zlib, Huffman, etc.) and the choice of the compression algorithm impacts the NCD results.
- Smith-Waterman Algorithm: The local sequence alignment of biological sequences is a general technique that can be equivalently used for string similarity. This is the case of the Smith-Waterman Algorithm, which is a local sequence alignment algorithm that finds the optimal local alignment between two strings, considering insertions, deletions, and substitutions. As many of the string similarity metrics, it uses dynamic programming and a scoring system to assign values to match, mismatch, insertion, and deletion events. A positive score is assigned to matches, a negative score to mismatches, and penalties for insertions and deletions. Given two strings of length i and j, $H(i,j)$ represents the score of the best alignment ending at positions i and j with:

$$H(i,j) = \max \begin{cases} H(i-1,j-1) + \text{Score}(i,j), \\ H(i-1,j) + \text{Gap Penalty}, \\ H(i,j-1) + \text{Gap Penalty}, \\ 0 \end{cases}$$

3.2. Vectors

A program, graph and raw data in general can be represented with feature vectors. Therefore, any metric that calculates the proximity of two (or several) feature vectors in its multidimensional feature space can also be used for similarity comparison. There are many of them, the most common being:

- Cosine similarity: It calculates the similarity of two non-zero vectors based on the cosine of the angle between them. The cosine similarity ranges from -1 to 1, with 1 indicating that the vectors are identical in direction (perfect similarity), 0 indicating that the vectors are orthogonal (no similarity), and -1 indicating that the vectors are exactly opposite in direction (perfect dissimilarity). The cosine similarity is particularly useful when the magnitude of the vectors is not important, and the focus is on the direction or orientation of the vectors. In natural language processing, cosine similarity is often applied to represent documents as vectors in a high-dimensional space, with each dimension corresponding to a term frequency or other relevant feature. The similarity between documents is then calculated using the cosine similarity measure. This could also be applied to assembly code similarity.
- Descriptional entropy: This technique, presented in (Desnos and Erra, 2013) tries to capture the notion of complexity (the existence of patterns) into an entropy measure in such a way that two malware functions (or files) that share similar complex subsequences are considered with a high degree of similarity. We consider this metric to be a vector metric since entropy is calculated with the well-known entropy formula over the occurrence vector distributions representing the different subsequences existing in a sequence of objects (which could be bytes, instructions, etc.). Although any entropy related measure suffers with intelligent instruction reordering, the descriptional entropy is more resilient since the reordering needs to be done ensuring that the complexity is not maintained, which reduces the number of possible configurations.

There are other well known vector metrics like the Manhattan, Euclidean or Minkowski distance, all of them distance metrics between vectors in $\mathbb{R}^n$, each of them with its own characteristics, that can be used in similarity calculation.

3.3. Sets

Many of the malware features can be represented in different forms. Such a representation structure are sets, specially when order of the features to be represented is not critical (similarities in different parts of the malware sample or function are still similarities worth spotting). Among these sets' metrics we consider the following to be more relevant to malware:

- Dice's coefficient: The Dice's coefficient is particularly useful when dealing with binary data, such as presence or absence of features. It is calculated using the following formula:

  $D(A,B) = \frac{2|A \cap B|}{|A| + |B|}$ where A and B are two sets.

  Dice's coefficient ranges from 0 to 1, where 0 indicates no overlap between sets, and 1 indicates complete overlap. It provides a measure of the proportion of elements that are common between the sets relative to the total number of elements in the sets. In natural language processing, Dice's coefficient can be used to compare the similarity of two texts by treating words or n-grams as sets. Dice's coefficient is particularly effective when dealing with imbalanced datasets, where one set is much larger than the other. It is less sensitive to the size of the sets compared to some other similarity metrics. Like other set-based similarity measures, Dice's coefficient does not consider the order or frequency of elements within the sets and it can be sensitive to small differences in the set size.
- Bloom filters: It can be used for (fuzzy) hash comparisons (see (Roussev, 2010)). While it is primarily used for membership queries, it can be adapted for similarity search. A Bloom filter consists of a bit array of size m and k hash functions. Initially, all bits are set to 0. To insert an element x into the Bloom filter, we apply each of the k hash functions to the element and set the corresponding bits to 1 in the bit array. To check if a given element y is in the set, we apply each of the k hash functions to this new element and if all corresponding bits are set to 1, the element is considered *possibly in the set*. However, false positives are possible. To query for similarity instead, we apply the same hash functions to the query element and then check the set bits in the Bloom filter. The more matching bits, the higher the similarity. The choice of hash functions and the number of hash functions impact the performance and accuracy of the similarity search.
- Common n-grams[3]: Once we have two sets of n-grams, the number of common n-grams gives a similarity metric. Therefore, applying, for instance, Jaccard definition over two sets A and B as $\frac{|A \cap B|}{|A \cup B|}$, we get a similarity value for the n-grams. In the weighted version, each common n-gram is assigned a weight that reflects its importance or relevance. The weights are determined based on specific criteria, and they provide a more refined measure of similarity. If applied only to malware instructions and those are normalized as described in 2.1, weights can be assigned depending on, for instance, the entropy of the common n-grams (since common rare n-grams explain a higher degree of similarity). Other criteria can be selected (e.g. term frequency or inverse document frequency for more general contexts).

3.4. Graphs

Graph similarity techniques are methods used to compare and measure the similarity between graphs, which consist of nodes and edges. As we already described in 2.2, graphs can be used to represent mal-

[3] This metric (and many others) can alternatively be categorized as a string metric if n-grams are generated by sub-strings.

Table 4
A comparison of similarity metrics.

Metric	Effectiveness	Complexity
Edit Distance	High	Moderate
Normalized Compression Distance (NCD)	High	High
Smith-Waterman Algorithm	High	Moderate
Cosine Similarity	High	Low
Descriptional Entropy	Moderate	Low
Dice's Coefficient	High	Moderate
Bloom Filters	High	Low
Common n-grams	Moderate	Moderate
Graph Edit Distance	High	High
Graph Isomorphism	High	High
Jaccard Similarity for Graphs	High	Moderate
Graph Kernel	High	High
Maximum Common Subgraph (MCS)	High	High

ware code and, consequently, having metrics to compare then is crucial for malware similarity search.

- Edit distance: Graph Edit Distance measures the minimum cost of transforming one graph into another through a series of edit operations (e.g., insertion, deletion, substitution of nodes or edges), similar to the edit distance metric already explained for strings.
- Graph Isomorphism: Graph Isomorphism checks whether two graphs are structurally identical, i.e., whether there exists a one-to-one correspondence between their nodes that preserves edge relationships. Formally, two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are isomorphic if there exists a bijective function $f : V_G \to V_H$ such that for every pair of vertices u and v in G, $(u, v) \in E_G$ if and only if $(f(u), f(v)) \in E_H$.
- Jaccard Similarity for Graphs: Jaccard Similarity measures the similarity between two graphs based on the shared nodes divided by the total number of distinct nodes in both graphs. Formally Jaccard similarity's score is $\frac{|V_G \cap V_H|}{|V_G \cup V_H|}$.
- Graph Kernel: Graph Kernels map graphs into high-dimensional feature spaces and compute the similarity based on inner products in that space. Common types include the Random Walk Graph Kernel, Graphlet Kernel and Shortest Path Kernel.
- Maximum Common Subgraph (MCS): Maximum Common Subgraph finds the largest common subgraph between two graphs. The goal is to identify the maximum set of nodes and edges that are shared by both graphs. It can be formally depicted as:

$$MCS(G_1, G_2) = \max_{\text{common subgraphs}} |V_{\text{common}}| + |E_{\text{common}}|$$

These metrics present different degrees of effectiveness, that is, how well a code similarity metric can accurately measure the similarity between two pieces of code and complexity, the level of computational resources or algorithmic sophistication required to implement and use a code similarity metric. These characteristics can be compared in Table 4.

4. Tools

The presented theoretical foundations for similarity search have been already applied to the implementation of tools. We introduce here some of them, highlighting their use case scenarios.

4.1. Elsim

The ELSIM (ELement SIMilarity) library (Desnos et al., 2023) offers Python functions designed for evaluating the similarity of byte strings through the utilization of Normalized Compression Distance (NCD). While the library itself is not confined to the comparison of any specific byte sequence, many tools have been designed to facilitate the comparison of Android applications in the form of APK or DEX files. To implement a novel method for comparing objects of choice, a wrapper and specific filter dictionaries are employed to structure the data within the Elsim module. It was initially integrated into Androguard android app (Desnos, 2023), a tool to analyze android applications.

4.2. BinDiff and diaphora

The main goal of these tools is to find similarities and differences between two files, more specifically between the assembly code functions of these files. To do this, BinDiff and Diaphora make a comparison between the different functions both at function level, i.e. using attributes of the function such as its name or its CFG, and at block level, i.e. using characteristics of the block such as the position it occupies in the CFG. Finally, accuracy and confidence are obtained for the similarities found between each pair of functions, depending on the quality and quantity of attributes that matched in the comparison.

These tools use a variety of attributes to find similarities between the functions of two files, many of them based on ideas extracted from the previous Sections. These attributes include: different graphs, such as the CFG or the callgraph, the MD-index (Dullien et al., 2010) for their representation, the name of the function, the existing strings, the instructions of each code block and so on. The use of so many heuristics makes the comparison carried out quite accurate and costly, both in space and time. However, this can be done because only two samples are being compared. If the number of files is increased, this approach would be totally unfeasible due to the required time to compute similarities.

If we focus on the differences between both tools, the first thing to note is that Diaphora is open source while BinDiff is not. Although BinDiff usually gets better results (Arutunian et al., 2021), many cybersecurity engineers prefer to use Diaphora because the use of technologies such as SQL and SQLite to model the heuristics. Also, these heuristics differ slightly between the two tools.

The last difference is the graphical user interface presented. Both of them work as a plugin for IDA Pro, but BinDiff also does so for GHidra (Eagle and Nance, 2020) and Binary Ninja (Vector35, 2022) (Diaphora works as an IDA plugin). While this may seem at first to be a major disadvantage for Diaphora, IDA Pro is, so far, the ultimate tool for malware analysts who want to study a sample, so it does not affect the vast majority of users.

5. A new fuzzy hash: CCBHash

Two files can be similar when the result of their execution is the same, or when it is different but resembling or similar functions are used in the process. We need to consider both possibilities. The objective is to be able to identify similarities in a large number of malicious files at the level of code blocks. When faced with petabytes of malware samples, we must look for a solution that is capable of storing the similarity information as compacted as possible and allowing a fast comparison.

To do this, we must be able to characterize a complete malware sample by breaking it into smaller pieces of assembly code and generating a *compound* hash for each of them. These hashes, which will identify each code segment, will later be joined in a single hash that will identify the entire sample, just like the fuzzy hashes presented in Section 2.3. These hashes are calculated offline, that is, once the CCBHash has been obtained, it is stored for later use. CCBHash calculation time is a limiting factor in our tool, but it is important to distinguish between the CCBHash computation time and the comparison time of these hashes. The latter is the really important one in our scenario.

As our proposal includes both the creation of a hash that identifies a function, and another hash formed by the concatenation of these that identifies the complete sample, in order to avoid confusion we will call the hash of the function CCBHash of the function, formed by the

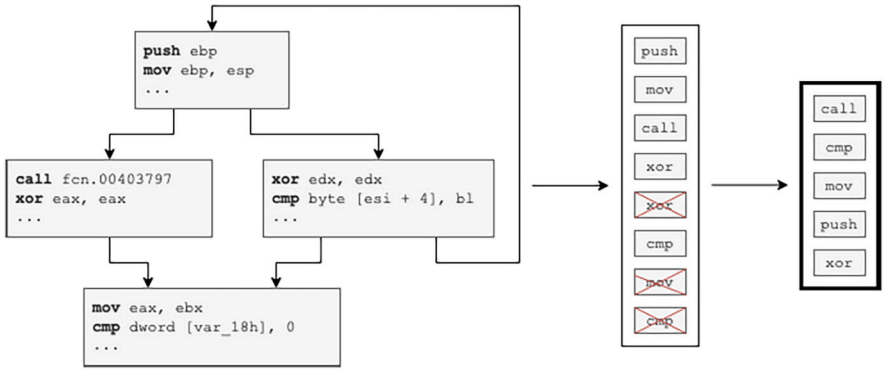


Fig. 2. Opcode types.

concatenation of attributes, and CCBHash of the sample to that of the complete file.

For the purpose of finding similarities in assembler code blocks, we must define the size of these blocks, or rather, the start and the end of each block. Since our goal is to store the resulting hashes in a database, calculating the signatures for all possible blocks of instructions (like in n-grams) would be somewhat unfeasible, both due to space and comparison time. For this reason, CCBHash will use the assembly code functions.

There are tools that look for differences (or similarities) in assembly code segments between two files, such as BinDiff and Diaphora. On the other hand, there are numerous studies and solutions that focus on the similarities between complete malware samples. However, the alternatives to search for similarities at the level of the assembler code block, or functions, among a very large number of malware samples are scarce. Some of these options, in particular fuzzy hashes like *ssdeep*, achieve acceptable execution times but must find exact matches. That is, they divide a sample into smaller blocks, usually bytes, which must be the same in order to detect similarities. Unlike ssdeep, in CCBHash the hash of each function will in turn be a fuzzy hash made up of its attributes, making it possible to find similarities between different functions.

In short, our goal is to combine two of these ideas: finding similarity between functions using as much information from the code as possible, as BinDiff and Diaphora do, and compacting that information into a fuzzy hash as ssdeep does. In this way, we would achieve a similarity analysis that is as accurate and fast as possible on petabytes of functions, which, in turn, and as a side benefit, would allow us to find similarities between samples.

5.1. CCBHash design

The first step in our fuzzy hash design is to characterize each code block function after extraction. We create a signature for each of them, which will be made up of a series of attributes represented by a hash or a numerical value (individually) and which will be concatenated into a single hash. That is, the hash of a sample will be made up of hashes of functions that in turn will be made up of attributes represented as hashes or numeric values. In this way, we will make each attribute individually comparable and as compacted as possible.

These attributes must be chosen carefully. Choosing attributes with too much semantic information could mean that, if all this information is not very similar for two samples, the attribute does not match and is discarded. In the same way, a choice of attributes that is too *weak* will cause matches to occur too often. For this reason, we will try to use heuristics that are somewhere in between, or else, separate certain features into various attributes. This approach will allow us to use some weak similarities. Although this may seem like a problem and may seem to encourage False Positives (FP), the combination of all of them will avoid it. In any case, our concern is not with the FP rate, but with the False Negative (FN) one, since what is really important is that we do not miss any similarity. Therefore, allowing weak attributes is essential in our tool.

We proceed selecting some high-level attributes, which treat the function as a black box, and other lower-level ones that we can find within the function itself. These are:

- **Function's name**. Although the names assigned by a disassembler are not relevant, the existence of debug information in the executable might make this attribute significant enough.
- **Function's CFG**. The CFG includes the execution flow of the function, so that two equal graphs can be a good indication that the functions are similar. To calculate the CFG, a procedure similar to that of (Bogard, 2022) has been followed, but without including the calls to other functions since this information will be included in the following attribute. The reason why this procedure has been used and not the MD-index is basically due to computation time. This attribute is, in most cases, insensitive to garbage embedding and instruction substitution or permutation, as well as register/variable remapping.
- **Function's callgraph**. This graph follows the flow of functions calls generated from the current block. The representation of this attribute is carried out in a very similar way to that of the CFG since the same procedure is followed but replacing the nodes with functions and the edges with calls to functions.
- **Number of inputs** *(indegree)* **and outputs** *(outdegree)*. It provides a function's degree of use within the malware as well as the number of functions used within it. This attribute will be in charge of collecting weak similarities related to function calls, that is, those that escape from the callgraph.
- **Cyclomatic complexity**. Understood as $A - N + 2C$ where A is the number of edges, N the number of nodes and C the number of output nodes. It tries to collect those similarities related to the execution flow that the CFG cannot detect.
- **Types of function opcodes**. The opcode collects the part of the instruction that indicates the operation to perform. With this attribute we collect all the types of opcodes of the function, being able to detect those functions with similar functionalities. This feature is insensitive to instruction reordering and register/variable reallocation. In Fig. 2 we see how the construction of this attribute would be carried out, ordering the opcodes alphabetically.
- **Number of instructions**. Returns the total number of instructions of the function. It complements the types of opcodes.
- **Number of blocks inside the function**. It reflects an intermediate point between the number of instructions and the CFG, detecting those similarities that these attributes do not detect.
- **Type of local variables and function arguments**. Both features (local variables and arguments) are collected separately and try to detect, among other things, information related to the function's prototype.

Table 5
Size according to attribute.

Attribute	Size
Function name	2 bytes
Function's CFG	2 bytes
Function callgraph	2 bytes
Types of opcodes	2 bytes
Type of local variables	2 bytes
Type of function arguments	2 bytes
Number of instructions	1 byte
Number of entries (indegree)	4 bits
Number of outputs (outdegree)	4 bits
Cyclomatic complexity	4 bits
Number of blocks	4 bits
Number of arguments and local variables	4 bits
Stack's size	4 bits
Total	16 bytes

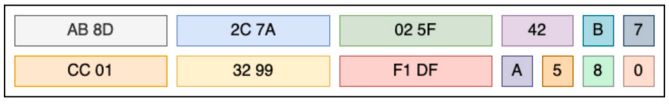


Fig. 3. Prototype of the function's CCBHash. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

- **Joint number of local variables and function arguments**. In this case, only the quantity of both characteristics is taken and they are added, giving rise to a single attribute. This heuristic is insensitive to possible type changes made by the attacker.
- **Stack's size**. This variable provides information about the size of the *stackframe*, which includes, among other things, the size reserved for the aforementioned variables and may find hitherto undetected similarities.

The choice of the mentioned attributes has taken into account three simultaneous requirements. Firstly, find code that acts in a similar way, that is, that performs similar operations. Secondly, detect code that performs operations in the same way (even if not identical). And the third and last approach tries to make the tool insensitive to the strategies used by the attackers, discussed in Section 2. This choice has been strongly influenced by tools such as BinDiff and Diaphora, which make use of a large number of high-quality attributes but require storage and comparison times that exceed our requirements.

Once we have gathered the necessary attributes, in order to calculate the fuzzy hash of the function, we generate a hash, or numerical value, for each of the attributes prior to their concatenation. The size of the representation of each attribute should be as small as possible keeping a probability of collision low enough. In Table 5 we present the size of the chosen attributes.

Attributes with a size of 2 bytes will be represented by a hash calculated on the attributes themselves. The rest of the attributes, will be stored with numeric values, either the value of the attribute itself, or an adaptation calculated by means of a piecewise defined function. These values will be defined in Section 5.2. In Fig. 3 we can see the result of the proposed prototype. Each color represents a different attribute.

CCBHash is limited by the choice of attributes. The more attributes are selected, the better the comparison will be, the worse the hash size and needed time for comparison. The same happens if we increase the space allocated to each of them. In this way, it has been necessary to reach a balance between quality, space and speed. Since sizes are fixed by design, function comparison can be made by attribute. In this way, given the fuzzy hashes of two functions, we can compare the fuzzy hash part corresponding to each attribute in parallel. Finally, we would know which attributes have presented coincidences, being able to determine a similarity percentage based on the number of identical attributes, as well as the quality of the result based on said attributes, since some attributes are more semantically important than others.

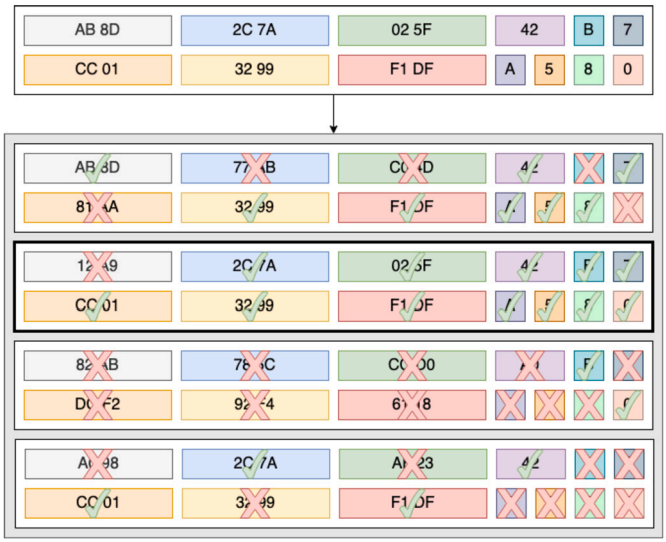


Fig. 4. Search for similarities between functions.

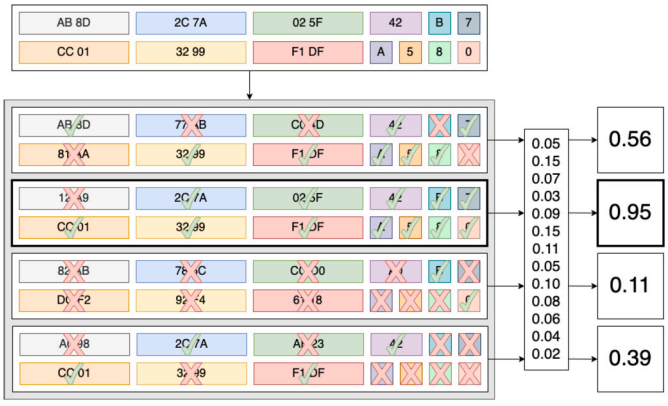


Fig. 5. Similarity calculation between functions.

Depending on the scenario where this fuzzy hash is used, the result of the comparison can be used differently. For example, we may be interested in knowing which function in our database is the most similar to the function studied, as we can see in Fig. 4. Additionally, we may be interested in finding similarities among complete samples. To do this, all the functions of the file would be studied and, for each sample in the database, they would be compared with all the functions of the target sample, storing the maximum similarity found for each function and calculating a similarity percentage for the file.

As previously stated, not all attributes have the same importance. Attributes such as the CFG or the type of instructions used should have a higher score than others such as the function's variables or name since the latter ones are more easily modifiable. Therefore, the attributes need to be weighted. The values used for these weights are collected in Section 5.2 and the operating scheme appears in Fig. 5.

Since our goal is to compare malware samples, and functions, it would be interesting to include a filter that discards those functions that are not interesting from the point of view of the malware analyst, such as, for instance, functions in which the only instruction is a call to another function, or legitimate functions from known libraries.

Finally, all that remains is to calculate the CCBHash of the sample. Once we have the hashes of each function, we concatenate them and we already have the CCBhash of the file as a result. Because the hash size of each function is known, different functions can be compared

simultaneously. In this way, the sample's CCBHash will be variable in size since it depends on the number of functions. Specifically, its size is $n \times f$ bytes, where n is the number of functions and f is the size of the hash of the function.

Although CCBHash allows generating the fuzzy hash of a file, the strength of the tool lies in the characterization of the function. We want to emphasize that the use should not consist exclusively on looking for similarities between different files, but also looking for similarities between a single function and a large dataset of functions. In this way, we can detect if a function resembles functions used in other malware samples, making it easier to perform attribution tasks.

5.2. Implementation

CCBHash is a fuzzy hash where each function is represented with a 16-byte hash, which in turn is made up of a series of attributes. However, we have not yet justified the reason for the size of each attribute. In order to obtain the attributes we have chosen to use the *radare2's r2pipe* library (Radare, 2022), available in *Python*. The choice of r2pipe is due to several reasons. It allows us to obtain a multitude of attributes of a malware sample, as well as its functions. This library is found in Python, a programming language in which we can find many other libraries like *hashlib* (Foundation, 2023), used to calculate the hash of certain attributes. Finally, r2pipe is *open source*, unlike other tools like *IDAPython* (Carrera, 2023).

Some software functions' attributes are usually related. For example, the attributes *number of blocks, number of edges and cyclomatic complexity* are correlated, so we do not need them all. In the same way, the size of a function and the number of instructions in it are usually proportional, so we may include only one of them. A correlation study helped us to come up with the attributes that finally make up our CCBHash as seen in Section 5.1.

Some of the CCBHash attributes are two bytes long. These attributes are represented by a hash calculated on the characteristic itself. This is because their values are not numeric, but vectors or other data structures that can be easily converted to *strings*. We apply a selected hash function to these in order to compress their information into two bytes. After evaluating different proposals, we selected *blake2b* (Aumasson et al., 2015) as the hash algorithm. This choice is mainly due to two reasons. First, this algorithm allows quickly generating hashes of arbitrary size (between 1 and 32 bytes), surpassing the speed of algorithms such as MD5, SHA-1, SHA-2 or SHA-3 (Ghoshal et al., 2020). The second reason is that it is available in the Python hashlib library, so it can be easily integrated along with r2pipe.

As for the rest of the attributes, one byte or four bits long, they can be represented with numerical values. If we extract these attributes from all the functions of selected samples (see Section 6.2) and we represent them graphically, we can observe their distributions. Fig. 6 shows the number of instructions, and Fig. 7 represents the rest of the numerical attributes, with four bits. In all cases we see that the relative frequencies are concentrated in the initial values. In fact, for high values we only find isolated cases that seem to hold for a very small number of functions.

Attributes do not have to be stored with their real value, since most of them are concentrated around low values. For example, regarding the function's number of instructions, for a selected dataset, we observed more that 90% of the values are less than 200 with some outliers being much greater (more than 1500). For the purpose of saving space we use range values. To determine these ranges we will use the Python *numpy* (Project, 2023) library, which allows us to quickly perform statistical operations. More specifically, we use the quantile calculation. For the # of instructions attribute, it will be necessary to obtain 255 equispaced quantiles, since one byte can store 256 values. For the rest of numerical attributes, represented with four bits, we will calculate 15 equispaced quantiles, since they only allow 16 values to be stored.

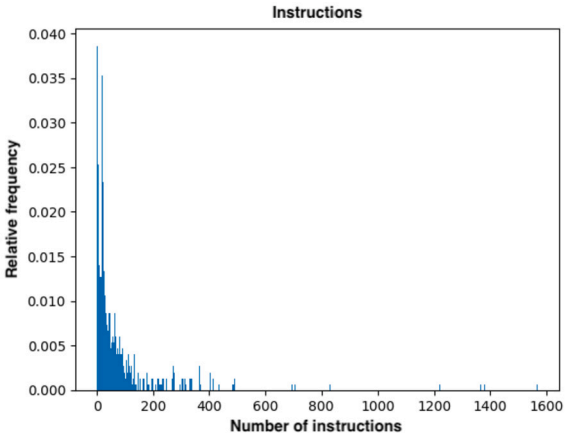


Fig. 6. # of instructions relative frequency.

Table 6
Weights according to attribute.

Attribute	Score
Types of function opcodes	0.25
Function's CFG	0.16
Cyclomatic complexity	0.15
Outdegree	0.09
Number of instructions	0.07
Function's callgraph	0.06
Type of function arguments	0.04
Type of local variables	0.04
Number of local variables and function arguments	0.04
Number of blocks	0.04
Indegree	0.03
Stack's Size	0.02
Function name	0.01

Two of the strongest attributes are the CFG and the opcode types. In the same way, the weakest attributes are the function name and the stack size, since they are more likely to vary even for equal samples. However, we cannot assign weights based solely on our theoretical knowledge. We empirically approximated them (with our selected dataset).

Proceeding as mentioned with 70% of the samples and using the remaining 30% to validate the results, we obtained the optimal weights found in Table 6.

Finally, we added a filter where functions with fewer than ten instructions are dropped. In this way, we will not take into account some functions that, from the point of view of malware analysis, are not relevant.

6. Dataset

Below we analyze our design and implementation. Before exposing the results of our analysis we clearly set the initial dataset used for the different instances that provide context for this malware families.

6.1. Scenario and problem statement

We have already presented a study and classification of the foundations to be applied in previous Sections. Nevertheless we have also analyzed different versions of some of the most relevant malware families since there are multiple versions and samples for each of these families ensuring therefore that functionality similarity is present in our dataset. In the experiments we conducted to verify the effectiveness of our hash function we also used legitimate pieces of software. Following, we provide some brief information about these malware samples together with some context, so we have a better general understanding of its functionality if the experiments are to be reproduced:

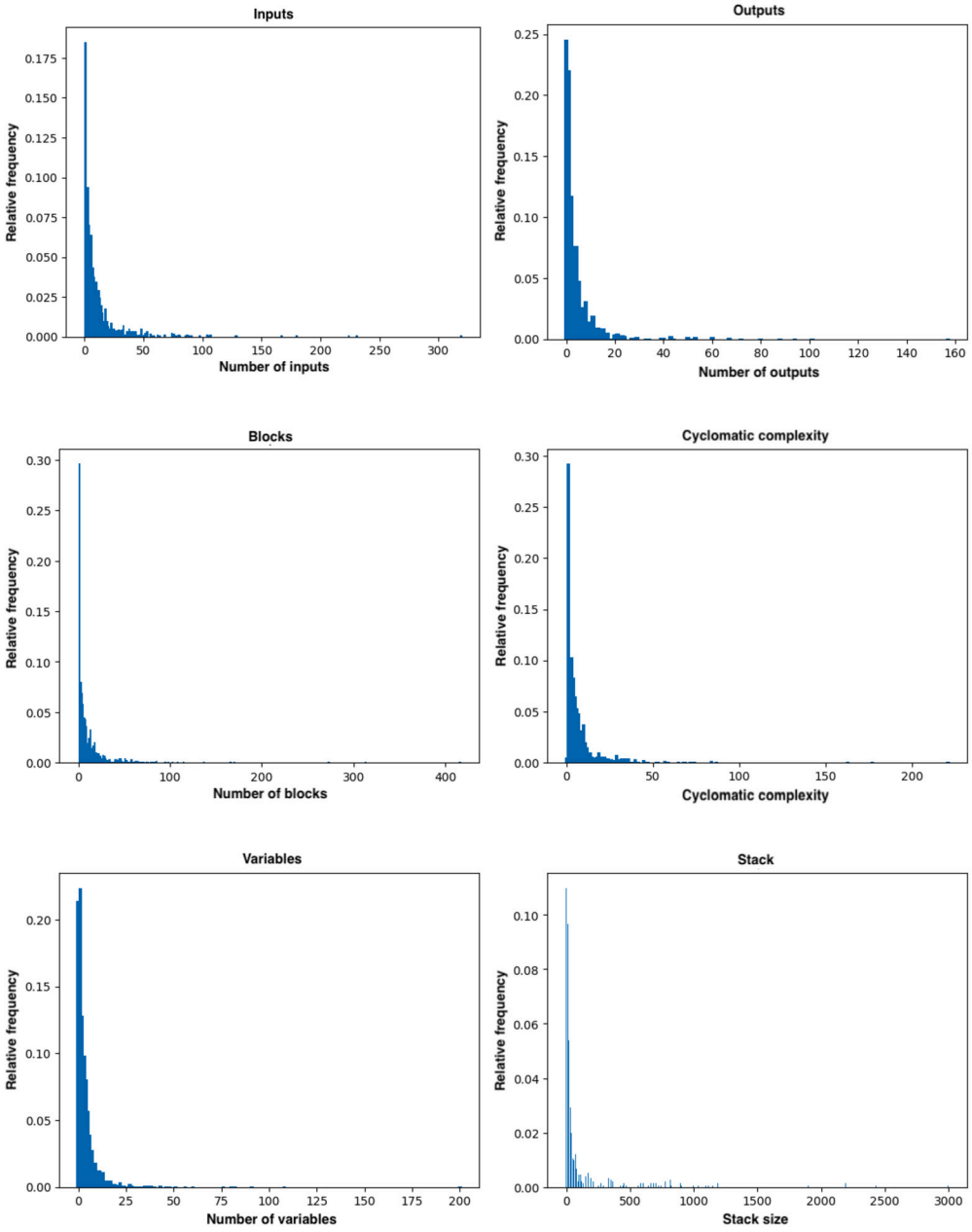
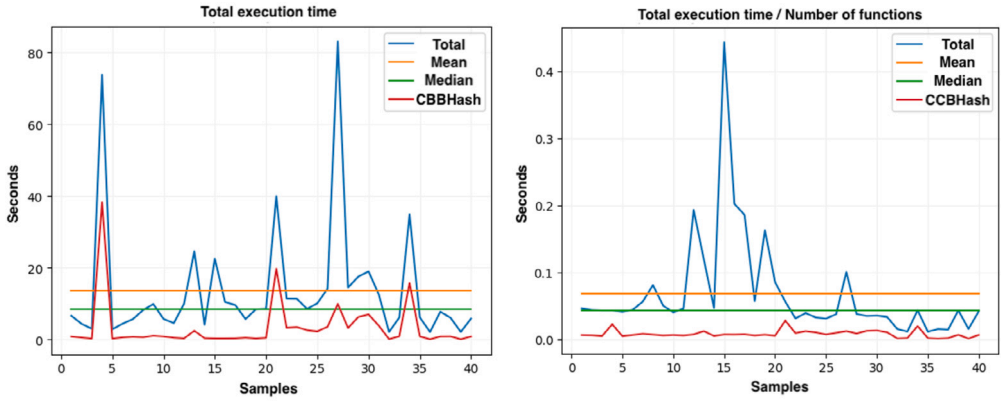


Fig. 7. Relative frequency of numeric attributes.

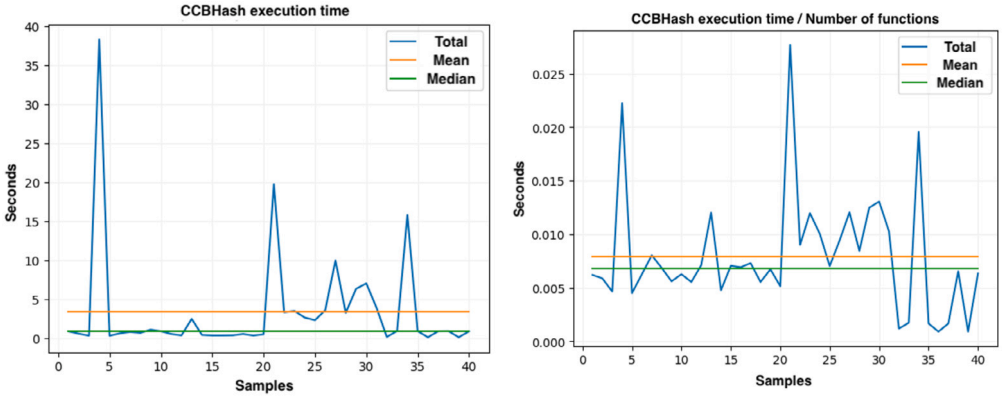
- **WannaCry** (Martin et al., 2018; Ghafur et al., 2019). It is the ransomware par excellence. A massive attack took place in 2017 and affected approximately 230,000 computers worldwide, especially the Telefónica company and thousands of hospitals and NHS clinics in the United Kingdom. WannaCry affected computers running Microsoft Windows and demanded a ransom using cryptocurrencies in exchange for the files' recovery.
- **DarkSide** (Nuce et al., 2021). This is another famous ransomware that attacked between 2020 and 2021. Among other large companies in the industrial sector in more than 15 countries, one of the most affected companies was Colonial Pipeline, the largest oil pipeline company in the US. As with the rest of the ransomware, the objective was to obtain a monetary ransom after encrypting the files of the affected computer.
- **Ryuk** (Li, 2021). Another ransomware. It was detected for the first time in 2018. It stands out for being aimed at large public entities. It typically encrypts the data on the infected system, making it inaccessible until a ransom is paid.
- **Zeus** (Etaher et al., 2015; Binsalleeh et al., 2010). This malware is a Windows Trojan allowing the infected computers to be part of a botnet. It was first identified in 2007 although it was active at least until 2010. It has become one of the most effective botnets in the world, infecting millions of computers and generating a wide variety of similar components from its code. It was originally designed to steal online banking credentials from attacked computers.

We used a dataset with 40 samples from these families, including ten samples for each family. Although the dataset may seem insufficient, it contains a total of 12,402 features, that is, approximately 310 per sample. WannaCry has an average of 200 functions per sample, Ryuk 428, Zeus 354 and DarkSide 276. Median values are 125 features for



(a) Run time per sample. (b) Average execution time per function.

Fig. 8. Total execution time.



(a) Computation time per sample. (b) Computation time per function.

Fig. 9. CCBHash computation time.

WannaCry, 381 for Ryuk, 142 for Zeus, and 102 for DarkSide. These samples have been downloaded via VirusTotal.

6.2. Empirical analysis

After programming the tool[4] we carried out experiments with a simple 2.9 GHz Intel Core i5 processor. As we commented in Section 6.1, the CCBHash generation time is not critical in our context. We studied each sample average time to calculate all its functions' CCBHashes. The total average time for each function is 67 milliseconds. The median value drops to 43 milliseconds. More details can be found in Figs. 8a and 8b.

Although, in general, execution times for each function are as expected, we can observe some exceptions. These belong to the WannaCry family. To this problem we must add that a sample can contain many functions, so the time to calculate all the functions' CCBHashes is not negligible at all (over one minute in some cases).

Our algorithm has two main components. The first one obtains the functions attributes using r2pipe, and the second one actually generates our hash, but also extracts some attributes not returned directly by r2pipe. In Fig. 8 we appreciate the total time (blue line) as well as the time taken by our second component (red line). We observe that there is a big difference between both values. Ignoring the time consumed by r2pipe, our algorithm takes an average of 7.9 milliseconds to execute, that is, r2pipe uses 89% of the total time needed. In Fig. 9 we can see all the information related to the computation of CCBHash.

In terms of comparison times, direct attribute comparisons involve operations that are not complex. Therefore, it is reasonable to anticipate low times, aligning with our objectives. Indeed, generating the CCBHash for each function in every sample, with over 12,000 functions in our database, takes less than an average of 24 ms. If deployed on a more powerful computer with multiple processors and/or cores, concurrent programming techniques could further reduce this time.

Given the impracticality of comparing over 76 million possible pairs of functions, we distribute these functions across complete malware samples. This enables a per-sample analysis, facilitating similarity checks within the same family and discerning dissimilarity across distinct families.

The calculation of similarity between files allows for various approaches. In our case, we consider two values: the average similarity of functions and the count of functions with a similarity exceeding 75%. While the average provides an overview, the latter is crucial for detecting cases where a low average similarity masks very similar functions.

Another consideration is the comparison of samples with a varying number of functions. In this scenario, the order of comparison is pivotal, impacting the perceived similarity. Therefore, we account for this order of comparison, restricting files to be compared only when one contains more functions than the other.

Commencing with the analysis of samples from the same family, we initially present generic values, progressively delving deeper. Table 7 illustrates the general similarity obtained for files from the same family

[4] https://github.com/nicslabdev/vtgraph-utils/tree/main/ccbhash.

Table 7
Malware family average precision.

Malware Family	Same family	Distinct family
WannaCry	73%	37%
DarkSide	71%	38%
Ryuk	85%	42%
Zeus	79%	36%
Total	77%	38%

Table 8
CCBHash vs ssdeep.

Malware family	Average number of files					
	CCBHash			ssdeep		
	> 90%	> 75%	> 50%	> 90%	> 75%	> 50%
WannaCry	4.5	0	0	0.4	1.1	2.3
DarkSide	2	2	2.3	0.2	0.2	0.7
Ryuk	6	0	2	0	0	0
Zeus	3.3	0	0	2.2	0	0
Total	4	0.5	1.1	0.7	0.3	0.8

and different families. The average similarity between samples from the same family is 77%. These values provide a qualitative indication that samples from the same family exhibit higher similarity compared to samples from different families, even though the average values may not precisely represent the precision of our tool.

Upon closer examination of the test results, we observe that, for each and every sample of the same family, multiple files are identified with over 90% similarity. Notably, there are instances where various samples from the same family exhibit similarities of approximately 55%. A detailed scrutiny of these cases reveals that, in most instances, these samples manifest scenarios where around 30% of the functions display more than 75% similarity. This aligns with the previously mentioned technique where malware conceals itself within a legitimate sample.

For samples from distinct families, it is noteworthy that, in none of the cases, the percentage of similar functions with a similarity exceeding 75% surpasses 3% of the total functions in the file. In fact, the mode —indicating the most frequently occurring value— for similar functions is zero.

6.3. CCBHash comparison

We explored the behavior of CCBHash in comparison to other existing fuzzy hashes. To initiate this exploration, we examined our sample set using ssdeep, the current preeminent fuzzy hashing algorithm, and compared it with CCBHash. In our conducted study, each sample was compared with the rest of the samples in the database using both algorithms, and the number of similar files found for each sample was recorded. To facilitate this, we categorized the comparisons into three similarity intervals based on the obtained scores: $(50\%, 75\%]$, $(75\%, 90\%]$ y $(90\%, 100\%]$.

Table 8 presents the average comparison results for each family. CCBHash outperforms ssdeep for all malware families. Notably, in the specific case of the Ryuk family, CCBHash identifies an average of around six files for each sample with over 90% similarity and even two files with similarity in the interval $(50\%, 75\%]$. In contrast, ssdeep fails to detect any similarity among different samples in this family. Furthermore, for the Ryuk family, all samples in the interval $(50\%, 75\%]$ had, on average, more than 50 functions with over 90% similarity. Thus, CCBHash not only surpasses ssdeep but also provides information about similar functions even in cases where the similarity is not extremely high.

The next fuzzy hashing algorithm we will compare our tool to is TLSH. Comparing an algorithm like ours, where similarity is measured as a percentage, with TLSH, where scores are presented as distances, is not a trivial task. CCBHash, like ssdeep, utilizes a range from 0% to 100% to measure similarity, where 100% signifies perfect similarity and 0% denotes no similarity at all. Conversely, TLSH's range is unbounded, with a distance of 0 indicating that two files are identical (or nearly identical), and the higher the value, the greater the difference, potentially exceeding 1000. In our case, and drawing from the study by (Oliver et al., 2013), we will constrain the score range based on the False Positive (FP) rate, not accepting rates exceeding 6%, achieved for a distance of 100, which is already excessively high. Therefore, these intervals are defined as $[0, 10)$, $[10, 50)$, and $[50, 100)$.

Table 9
CCBHash vs TLSH.

Malware family	Average number of files					
	CCBHash			TLSH		
	> 90%	> 75%	> 50%	< 10	< 50	< 100
WannaCry	4.5	0	0	0	0	0.6
DarkSide	2	2	2.3	0.2	0.8	0.5
Ryuk	6	0	2	0	0.2	1
Zeus	3.3	0	0	2	0	0
Total	4	0.5	1.1	0.6	0.3	0.5

Table 9 presents the obtained results. Once again, our algorithm outperforms TLSH. Even when aggregating the averages obtained for all TLSH intervals, it does not surpass the result obtained for the most restrictive interval in CCBHash, with similarities exceeding 90%.

7. Conclusions

Malware similarity is crucial to malware detection, classification, analysis and response. If an unknown sample is detected, there is no doubt that AI and ML algorithms will help systems in these tasks. Nevertheless these algorithms can be complemented and enriched with existing techniques, tools and metrics and even substituted when fast investigation processes are time critical. With this in mind we surveyed and classified such techniques, tools and metrics in order to provide foundational context on available procedures and to extract sufficient knowledge for the design of a new tool ready to aid in malware investigations over petabytes of information: CCBHash. In addition to comparing functions, it can also operate as a traditional fuzzy hash, enabling the search for similarities among different malware samples and improving upon widely accepted solutions used by malware analysts, such as ssdeep or TLSH.

Having examined our algorithm and compared it with two widely used fuzzy hashing algorithms, CCBHash identifies similarities more accurately. Therefore, we have a compelling alternative to consider in the quest for similarity detection in malware. However, this capability comes with a trade-off. Our algorithm is designed to compare functions in assembly language, leading to a larger storage requirement for the fuzzy hash of an entire sample— i.e., a concatenation of the CCBHashes of its functions—compared to the storage utilized by ssdeep or TLSH. We aim to address this in the future through enhancements such as a more sophisticated function filters.

There are still further improvements to be done. We conducted several empirical experiments to rule out collisions on our dataset with successful results, although our specification shows that they can occur. More thorough research and experiments are needed. Among these are conducting a comparative analysis between Machine Learning Algorithms and/or a Deep Learning architecture with the same purpose. There is also the need to conduct larger experiments in real scenarios since the hash function has been designed and implemented to be applied over a vast amount of information.

Exploring new file disassembly tools is one avenue for enhancing execution times. However, optimization extends beyond this aspect. Upon scrutinizing the computation time of CCBHash without r2pipe,

instances with elevated values share a commonality —they feature a substantial number of functions characterized by intricate CFGs. Hence, there is a need to investigate more efficient algorithms for calculating CFGs. All these activities constitute the next steps to be taken in this work.

CRediT authorship contribution statement

Jose A. Onieva: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Formal analysis, Data curation, Conceptualization, Investigation. **Pablo Pérez Jiménez:** Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Javier López:** Writing – review & editing, Project administration, Funding acquisition.

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.

Data availability

I have shared the link to my code at the Attach File Step.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Innovation through the SECAI project (PID2022-139268OB-I00), and by the European Union under HORIZON-TMA-MSCA-SE Project (GA ID: 101086308). Funding for open access charge: Universidad de Málaga / CBUA.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cose.2024.103856.

References

Arutunian, M., Hovhannisyan, H., Vardanyan, V., Sargsyan, S., Kurmangaleev, S., Aslanyan, H., 2021. A method to evaluate binary code comparison tools. In: 2021 Ivannikov Memorial Workshop (IVMEM). IEEE, pp. 3–5.

Attaluri, S., McGhee, S., Stamp, M., 2009. Profile hidden Markov models and metamorphic virus detection. J. Comput. Virol. 5, 151–169. https://doi.org/10.1007/s11416-008-0105-1.

Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C., 2015. Blake2b de hashlib. https://www.blake2.net/.

Baruque, A.O.C., 2015. Malware HMM toolset. https://github.com/OrBaruk/Malware-HMM.

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M.M., Lavoie, Y., Tawbi, N., 2001. Static detection of malicious code in executable programs. Int. J. Requir. Eng.

Berre, S.L., Chevalier, A., Pourcelot, T., 2022. Machoc. https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md.

Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A., Debbabi, M., Wang, L., 2010. On the analysis of the zeus botnet crimeware toolkit. In: 2010 Eighth International Conference on Privacy, Security and Trust. IEEE, pp. 31–38.

Bogard, L., 2022. Machoke. Conix. https://github.com/conix-security/machoke.

Bonfante, G., Kaczmarek, M., Marion, J.Y., 2007. Control flow graphs as malware signatures. In: International Workshop on the Theory of Computer Viruses, pp. 1–6.

Breitinger, F., Astebøl, K.P., Baier, H., Busch, C., 2013. mvhash-b - a new approach for similarity preserving hashing. In: 2013 Seventh International Conference on IT Security Incident Management and IT Forensics, pp. 33–44.

Carrera, E. Introduction to idapython, pp. 1–64. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=700f71f80a8bc651591841e86af99232066c919d.

Caviglione, L., Choraś, M., Corona, I., Janicki, A., Mazurczyk, W., Pawlicki, M., Wasielewska, K., 2020. Tight arms race: overview of current malware threats and trends in their detection. IEEE Access 9, 5371–5396.

Desnos, A., 2023. Androguard. https://github.com/androguard/androguard.

Desnos, A., Erra, R., 2013. Descriptional entropy: application to security software analysis. In: Advanced Infocomm Technology, pp. 225–230.

Desnos, A., Lircyn, Grosse R., Nikoli, Halder S., Bachmann, S., 2023. Elsim. https://github.com/IKARUSSoftwareSecurity/elsim. (Accessed 27 November 2023).

Dullien, T., Carrera, E., Eppler, S.M., Porst, S., 2010. Automated attacker correlation for malicious code. Technical Report. Bochum Univ., Germany FR.

Eagle, C., Nance, K., 2020. The Ghidra Book: The Definitive Guide. No starch press.

Eresheim, S., Luh, R., Schrittwieser, S., 2017. The evolution of process hiding techniques in malware-current threats and possible countermeasures. J. Inf. Process. 25, 866–874.

Etaher, N., Weir, G.R., Alazab, M., 2015. From zeus to zitmo: trends in banking malware. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1. IEEE, pp. 1386–1391.

Foundation, P.S., 2023. Documentación de la librería hashlib de python. https://docs.python.org/3/library/hashlib.html. (Accessed 8 November 2022).

Gao, Y., Lu, Z., Luo, Y., 2014. Survey on malware anti-analysis. In: Fifth International Conference on Intelligent Control and Information Processing. IEEE, pp. 270–275.

de Geus, S.P.L., Grégio, A.R.A., 2015. Classifying Malware Using Dynamic Analysis and Profile Hidden Markov Models. Master's thesis. University of Campinas, Institute of Computing.

Ghafur, S., Kristensen, S., Honeyford, K., Martin, G., Darzi, A., Aylin, P., 2019. A retrospective impact analysis of the wannacry cyberattack on the nhs. npj Digit. Med. 2 (1), 1–7.

Ghoshal, S., Bandyopadhyay, P., Roy, S., Baneree, M., 2020. A journey from md5 to sha-3. Trends Commun. Cloud Big Data, 107–112.

Harbour, N., 2006. Dcfldd. http://dcfldd.sourceforge.net.

Hex-Rays, 2022. Ida pro. https://hex-rays.com/ida-pro.

Jiménez, P.P., Onieva, J.A., Fernandez, G., 2022. Ccbhash (compound code block hash) para análisis de malware. In: XVII Reunión Española Sobre Criptología y Seguridad de la Información, pp. 168–173.

Kim, J., Choi, H., Yun, H., Moon, B.R., 2016. Measuring source code similarity by finding similar subgraph with an incremental genetic algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. ACM, New York, NY, USA, pp. 925–932.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. In: The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). Digit. Investig. 3, 91–97. https://doi.org/10.1016/j.diin.2006.06.015.

Krinke, J., 2001. Identifying similar code with program dependence graphs. In: Proceedings Eighth Working Conference on Reverse Engineering, pp. 301–309.

Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G., 2006. Polymorphic worm detection using structural information of executables. In: Recent Advances in Intrusion Detection, pp. 207–226.

Labs, F., 2022. Global Threat Landscape Report. Threat Report. Fortinet.

Lazo, E.G., 2021. Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion techniques. In: Microsoft Security Blog: Microsoft Threat Intelligence. https://www.microsoft.com/en-us/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques.

Li, A.S., 2021. An Analysis of the Recent Ransomware Families. Project Report. Purdue University.

Li, Y., Jang, J., Ou, X., 2019. Topology-aware hashing for effective control flow graph similarity analysis. In: Chen, S., Choo, K.K.R., Fu, X., Lou, W., Mohaisen, A. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 278–298.

Liu, J., Wang, Y., Wang, Y., 2016. The similarity analysis of malicious software. In: 2016 IEEE First International Conference on Data Science in Cyberspace (DSC). IEEE, pp. 161–168. http://ieeexplore.ieee.org/document/7866123/.

Liu, L., Wang, B.s., Yu, B., Zhong, Q.x., 2017. Automatic malware classification and new malware detection using machine learning. Front. Inf. Technol. Electron. Eng. 18 (9), 1336–1347. https://doi.org/10.1631/FITEE.1601325. http://link.springer.com/10.1631/FITEE.1601325.

Martin, G., Ghafur, S., Kinross, J., Hankin, C., Darzi, A., 2018. Wannacry—a year on. BMJ 361, 1336–1347. https://doi.org/10.1136/bmj.k2381. http://link.springer.com/10.1631/FITEE.1601325.

Mayrand, Leblanc, Merlo, 1996. Experiment on the automatic detection of function clones in a software system using metrics. In: 1996 Proceedings of International Conference on Software Maintenance, pp. 244–253. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=565012.

Monnappa, K., 2018. Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware. Packt Publishing Ltd.

Nuce, J., Kennelly, J., Goody, K., Moore, A., Rahman, A., Williams, M., McKeague, B., Wilson, J., 2021. Shining a light on darkside ransomware operations. FireEye Blogs.

Oliver, J., Cheng, C., Chen, Y., 2013. Tlsh–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, pp. 7–13.

Project, N., 2023. Numpy. https://numpy.org. (Accessed 8 November 2023).

Rad, B.B., Masrom, M., Ibrahim, S., 2012. Camouflage in malware: from encryption to metamorphism. Int. J. Comput. Sci. Netw. Secur. 12, 74.

Radare, 2022. R2pipe de radare2. https://www.radare.org/n/r2pipe.html.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: Chow, K.P., Shenoi, S. (Eds.), Advances in Digital Forensics VI. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 207–226.

Song, W., 2014. A framework for automated similarity analysis of malware. Master's thesis. Concordia University. https://spectrum.library.concordia.ca/id/eprint/978935/. unpublished.

Ször, P., Ferrie, P., 2001. Hunting for metamorphic. In: Virus Bulletin Conference, pp. 123–144.
Vector35, 2022. Binary ninja. https://binary.ninja/.
VirusTotal, 2022. Virustotal. https://www.virustotal.com. (Accessed 8 November 2022).
Yan, J., Yan, G., Jin, D., 2019. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 52–63. https://ieeexplore.ieee.org/document/8809504/.

Jose A. Onieva received his PhD degree from the University of Malaga (2006) and is an Associate Professor in the Computer Science Department. He has been actively involved in ICT European and national funded information security related projects. He has published in several international journals and conferences in the field of Information Security. He is coauthor of the book "Secure Multi-Party Non-Repudiation Protocols and Applications" published by Springer. Currently he is involved in the research of core security services for edge computing, malware detection and response and digital evidence. Is a full professor and head of the Network, Information and Computer Security (NICS) Lab at the University of Malaga, Malaga, 29071, Spain. His research activities are mainly focused on network security, security protocols, and critical information infrastructures, leading a number of national and international research projects in those areas. He is Senior Member of IEEE.

Pablo Perez Jimenez holds a Bachelor's Degree in Telecommunication Technology Engineering and a Master's Degree in Computer Science with a specialization in Cybersecurity. While working at NICS Lab as a Researcher, he has participated in the RECSI (The Spanish Meeting on Cryptology and Information Security). Currently, he is employed as a Software Engineer at VirusTotal, Google.

Javier López is a full professor and head of the Network, Information and Computer Security (NICS) Lab at the University of Malaga, Malaga, 29071, Spain. His research activities are mainly focused on network security, security protocols, and critical information infrastructures, leading a number of national and international research projects in those areas. He is Senior Member of IEEE.