Enhancing Malware Classification via Self-Similarity Techniques

Fangtian Zhong, *Member, IEEE*, Qin Hu, *Member, IEEE*, Yili Jiang, *Member, IEEE*, Jiaqi Huang, *Member, IEEE*, Cheng Zhang, and Dinghao Wu

***Abstract*— Despite continuous advancements in defense mechanisms, attackers often find ways to circumvent security measures. Windows operating systems, in particular, are vulnerable due to fewer restrictions on downloading software from unknown sources, facilitating the spread of malware. To address this challenge, researchers have focused on developing techniques to identify Windows malware, crucial for mitigating potential damage. Traditional approaches typically categorize threats into broad classes such as trojans or adware, often failing to capture the full spectrum of malicious behaviors exhibited by diverse malware variants. In response, we propose a novel approach to malware categorization that incorporates both the general malware family and subfamily for each sample. Our method leverages self-similarity techniques to extract local semantics and similarities within the blocks of malware binaries while preserving correlations between these blocks. We utilize a VGG11 model to capture these features, enabling accurate classification. Central to our approach is the conversion of malware binaries into self-similarity descriptors, facilitating space savings while capturing essential semantics within blocks. By focusing on local self-similarities and their geometric layouts across malware, our method effectively identifies repetitive patterns indicative of malware behavior. Our proof-of-concept implementation demonstrates the effectiveness of our framework, achieving an impressive average precision of 98.2% on a newly gathered dataset with over 25,000 samples. Moreover, our method offers significant space savings, outperforming recent research efforts by a factor of over 96. These results underscore the efficacy of incorporating self-similarities and correlations within blocks for robust malware classification, making our approach a promising solution for real-world malware detection and prevention.**

***Index Terms*— Malware classification, malware family, variants, self-similarity.**

Manuscript received 16 March 2024; revised 25 June 2024; accepted 11 July 2024. Date of publication 25 July 2024; date of current version 1 August 2024. This work was supported in part by startup funds from Montana State University. The associate editor coordinating the review of this article and approving it for publication was Dr. Xiaojing Liao. *(Corresponding author: Dinghao Wu.)*

Fangtian Zhong is with the Gianforte School of Computing, Montana State University, Bozeman, MT 59718 USA (e-mail: fangtian.zhong@montana.edu).

Qin Hu and Yili Jiang are with the Department of Computer Science, Georgia State University, Atlanta, GA 30303 USA (e-mail: qhu@gsu.edu; yjiang27@gsu.edu).

Jiaqi Huang is with the Department of Computer Science and Cybersecurity, University of Central Missouri, Warrensburg, MO 64093 USA (e-mail: jhuang@ucmo.edu).

Cheng Zhang is with the Department of Computer Information and Decision Management, West Texas A&M University, Canyon, TX 79016 USA (e-mail: czhang@wtamu.edu).

Dinghao Wu is with the College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802 USA (e-mail: dinghao@psu.edu).

Digital Object Identifier 10.1109/TIFS.2024.3433372

I. Introduction

MALWARE attacks have become the most potent automated tool for hackers to launch different cyberattacks targeting individuals, as well as small, medium, and large businesses around the world. These attacks can result in financial losses, data breaches, disruption of services, and damage to reputation. Cybercrime Magazine [1] reported that cybersecurity ventures expect global cybercrime costs to grow by 15% per year over the next year, reaching $10.5 trillion annually by 2025, up from $3 trillion in 2015. This represents the greatest transfer of economic wealth in history, risks the incentives for innovation and investment, is exponentially larger than the damage inflicted by natural disasters in a year, and will be more profitable than the global trade of all major illegal drugs combined.

To avoid great economic loss, cybersecurity researchers have made significant contributions to detecting malware. Their techniques are primarily based on two dimensions: *features* and *algorithms*. The most popular techniques [2], [3], [4], [5], [6], [7] utilize static analysis, dynamic analysis, and hybrid analysis to extract features, such as opcodes, API calls, control flow graphs, n-gram byte sequences, and assembly code frequency. Static analysis requires security professionals to have a good command of the structure of malware binaries. For instance, to extract an imported Dynamic Link Library (DLL) in a Windows program, developers must first understand the Portable Executable (PE) format [8]. They then need to parse two structures in the order of DataDirectories followed by Import or Import Address Table (IAT). Although dynamic analysis can obviate the conundrums of static analysis, it can only collect the behaviors of specific execution paths since it is hardly possible to traverse all of them. Hybrid analysis seeks to strike a balance between static and dynamic approaches, yet encounters similar obstacles as both methods. Algorithms utilized for malware classification can be categorized into signature-based and AI-driven algorithms. Malware signatures are usually the hash values of features extracted by static analysis, dynamic analysis, or hybrid analysis. Signature-based algorithms [9], [10], [11], [12] look for these similar signatures for malware classification. Obviously, signature-based techniques are only capable of detecting malware that does not undergo significant changes in the order of features and codes [13]. Artificial intelligence-driven approaches [14], [15], [16], [17] are applied to automatically learn the features from the analysis techniques or abstract useful features from neural network embeddings for malware classification.

To conclude, the performance of signature-based algorithms and AI-driven algorithms largely depends on feature extraction and the learning abilities of machine learning models. Building on this fact, some researchers are redirecting their focus towards applying data visualization techniques to malware classification. The advantages of data visualization in this context include the incorporation of domain knowledge, enhanced efficiency, and simplified validation of performance. To acquire these benefits, data visualization aims to understand the essence of malware similarity and to train small machine learning models for correct classification. Unlike existing works [8], [18], [19], [20], [21] that apply data visualization directly to a malware dataset more than 10 years old without much domain knowledge and work on the general categorization, we have recollected a new dataset that combines general categorizations with their variants by relying on a set of third-party antivirus products that reflect the current trend of malware attacks. This study aims to understand malware similarity and discover underlying repetitive patterns for malware classification. It's important to note that our study is not intended to replace existing dynamic-behavior-based systems but to complement them, enabling more efficient malware analysis and broader coverage of malware programs. Since more than 85% of the malware samples received by a company belong to a known family or subfamily [22], removing these known malware executables can greatly reduce the consumption of resources and manpower. Our multi-fold contributions can be summarized as follows.

1. We propose the block based self-similarity (BBSS), a context-sensitive malware visualization classification framework, to effectively distinguish different types of malware families and their variants while in the meantime maintaining high accuracy.
2. We collect a new malware dataset of over 25,000 samples, meticulously categorized with finer granularity. This dataset has undergone validation by more than 70 third-party anti-virus products, dynamic analysis and manual analysis, offering a more accurate representation of the prevailing trends in malware attacks.
3. We provide visualizations of Windows feature maps to aid in the analysis and confirm the efficacy of both self-similarities within blocks and correlations between blocks for accurate malware classification.
4. Our methodology ensures the preservation of similarities among malware samples from the same family and subfamily while reducing the size of the training dataset by a factor of 96 compared to other approaches. Additionally, we find that most antivirus products in VirusTotal [23] do not distinguish malware samples from their families and subfamilies simultaneously.

The rest of this paper is organized as follows. Section II introduces related work. Section III details the system design and implementation. Section IV introduces the experiment setup. Section V reports the evaluation results and describes the limitations of the latest research. We conclude the paper in Section VI.

II. Background and Related Work

A. Malware Classification by Data Visualization

IBM provides a formal definition of data visualization as the portrayal of data using familiar graphics, such as charts, plots, infographics, and animations. These visual representations effectively convey intricate data relationships and insights in an easily understandable manner. The Vision Research Lab at the University of California, Santa Barbara, is a pioneer in this field, being the first to introduce the idea of data visualization into malware classification [24]. Their methodology is to convert a malware binary into an image by numeric conversion, which outputs an array consisting of numbers with a scope between 0 and 255; then the array is reshaped according to its file size and written to the disk as an image.

Since the initial release of the work "Malware Images: Visualization and Automatic Classification [24]," thousands of works applying data visualization to malware classification have been coming out within only twelve years. The superiority of techniques applied in these works can be mainly divided into three dimensions: advanced image processing, advanced machine learning algorithms and advanced feature extraction algorithms.

B. Advanced Image Processing for Malware Classification

VisMal [25] applies a contrast-limited adaptive histogram equalization algorithm to capture local patterns in the data and enhance the discernibility of malware samples by improving local contrasts of the regions. This technique enhances the correlation between byte codes for final classification. Nataraj et al. [24] primarily exploit the Global Image Structure Tensor image processing technique [26], [27] to transform images based on their global structural characteristics. This technique provides a compact and informative representation of an image's spatial layout. Makandar and Patrot [28] introduced a method for malware classification based on wavelet statistical features. The proposed model consists of three stages. In the first stage, pre-processing is conducted by applying wavelet transforms. In the second stage, feature extraction is performed using the Discrete Wavelet Transform (DWT), which decomposes the image into four levels. By leveraging wavelet analysis, this method captures both frequency and time-domain characteristics of malware data, enabling effective discrimination between different malware classes. Cui et al. [21] introduced data augmentation techniques to improve the quality of malware images. They empirically adjusted parameters relevant to quality, including rotation range, horizontal translation, vertical translation, image magnification, projection transformation, random zooming, and nearest padding.

C. Advanced Machine Learning Algorithms for Malware Classification

Hsiao et al. [29] made use of a set of deep convolutional neural network architectures and Residual networks to extract features from malware images and classify them. Vasan et al. [30] utilized Siamese neural networks and one-shot learning to learn representations of malware images in an unsupervised manner and uses these representations to classify new malware samples with minimal labeled data. Singh et al. applied deep neural network architectures, ResNet-50, including a dense Convolutional Neural Network (CNN) for classifying images. Jin et al. [18] introduced a different approach for imaged malware detection by an autoencoder that embeds convolutional neural networks. They trained autoencoder models to reconstruct these images and learn the functional characteristics of malware. The reconstruction error is then used as a measure of anomaly, enabling the detection of previously unseen malware samples. TL-CNN presents a method for malware classification on Android devices utilizing transfer learning and deep convolutional neural networks (CNNs) [31]. The authors leveraged pre-trained CNN models on large-scale image datasets and fine-tune them using malware images to classify them into different categories. By transferring knowledge learned from general image recognition tasks to the specific domain of malware classification, the proposed approach achieves high accuracy and efficiency in identifying malware samples on Android devices.

D. Advanced Feature Extraction Algorithms

MDMC [32] converts a malware binary into byte sequences and calculates byte transfer probability matrices. These matrices are created by correlating neighboring bytes and computing the probability of each byte pair across all byte pairs. The transfer probability matrices are then transformed into Markov images, which are analyzed using a CNN to capture sequential dependencies. Liu et al. [33] extracted features at various locations and scales within a malware image, which is divided into small patches. Each patch is further subdivided into smaller bins. As a sliding window moves across the image, it computes gradient histograms for each local bin. The image feature descriptors are then obtained using cascaded connection functions. These descriptors are organized into multiple blocks, with each block containing m $\times$ m sub-blocks. All feature blocks from the training images are clustered into k centers, and each feature block is encoded using the index of its closest center. Finally, the encoded feature blocks are processed by a histogram operator to form the feature vector. Chen et al. [34] disassembled a malware executable into assembly code, which is further divided into basic blocks. They collect opcodes from these basic blocks and then apply the Simhash algorithm to obtain hash bits, which are used as features for each basic block. Each hash bit is converted into a pixel, where 0 is converted to 0 and 1 is converted to 255. Finally, these pixels form an image. Adkins et al. [35] adopted similar methods to extract features but replaced registers with “REG”, locations with “LOC”, constant memory references with “MEM”, constant values with “CONST”, and variable references with “VAR”. Then, every four instructions are grouped to generate n-grams, which are further processed by MD5. The lowest truncated bits of the MD5 sum are used as the feature hash index.

E. Summary

Although the above methods have achieved good results in malware classification, most of them primarily focus on the distribution of bytecode within the program as a whole [18], [21], [24], [28], [29], [30], [31]. Fortunately, another set of methods takes into account the local correlations between bytecodes [25], [32], [33], [34], [35]. However, for some malware families with a large number of variants, their methods are greatly discounted. Therefore, we need to return to the essence of malware classification. For the general malware categorization, it is based on the behavior of malware. However, these behaviors can vary significantly in the form of code across different variants. A variant refers to a new version of malware based on existing malware with modifications. Therefore, we should focus on finding these similar code sequences, their inherent semantics, and their relative geometric positions. In this paper, we collected more than 25k malware samples and obtained their family labels as well as variant labels (subfamily labels) by grouping the results from more than 70 antivirus products in VirusTotal. We present BBSS, which converts malware binaries into byte codes that are divided into blocks. It uses the self-similarity technique to construct semantics similarity within blocks and VGG11 [36] to capture these similarity descriptors as well as their geo-correlations. VGG11 is a convolutional neural network (CNN) architecture that is part of the Visual Geometry Group (VGG) family of models, which were developed by researchers at the University of Oxford. It is a foundational CNN architecture that combines simplicity with competitive performance and makes it a valuable model in the field of computer vision.

III. Design

A. Overview of the BBSS Framework

The four primary objectives of BBSS are: i) Effectively classify malware samples into their respective families and subfamilies, ii) Resist the influence of non-rigid deformation within blocks for recognizing malware variants, iii) Preserve similarities for malware belonging to the same family and subfamily while significantly reducing the overhead on model training, and iv) Provide a data visualization validation method to assist in malware analysis. The BBSS framework, illustrated in Fig. 1, integrates several key components to achieve its objectives. Third-party antivirus products provide family and subfamily labels, which are refined and consolidated by the label cleanser using algorithms such as the longest common subsequence and MalwareBazaar malware family dictionary matching [37], alongside dynamic and manual analysis. The program analysis framework disassembles malware samples into instructions, identifies functions and basic blocks, and extracts essential program metadata. The semantics and feature engineering processor then converts malware binaries into byte sequences, organizes them into blocks, employs context-sensitive self-similarity techniques to extract their local semantics and similarities, preserves their geo-correlation between blocks, resizes them for input into the classifier, and generates Windows feature maps to aid in analysis. Finally, the classifier utilizes a VGG11 model to capture local and global

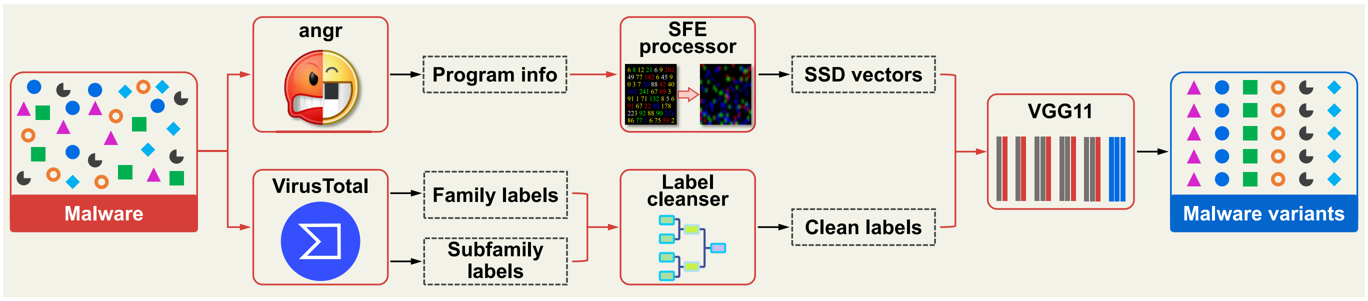


Fig. 1. The overview of BBSS framework.

features in a global ensemble of self-similarity descriptors, enabling accurate malware classification based on a hierarchy of concepts.

Once the VGG11 model is fully trained, it can be used to classify malware samples through a multi-step process. First, the malware program is processed by the semantics and feature engineering processor, which calculates self-similarity descriptors for each block. These descriptors are then processed using a binned log-polar representation, followed by concatenation and normalization. The resulting normalized descriptors are resized to a fixed size and then fed into the VGG11 model for classification.

B. The Components of the BBSS Framework

1) Third-Party Anti-Virus Products: In our study, we utilize VirusTotal as the third-party antivirus product provider. It is a website launched in June 2004, developed by the Spanish security company Hispasec Sistemas, a subsidiary of Google Inc. It aggregates numerous antivirus products and online scan engines to check for malware. The virtualization solution employed by VirusTotal is the Cuckoo sandbox. Users can submit files up to 650 MB to the VirusTotal website through the provided API by specifying the file path and a hash value calculated based on the file using SHA256, SHA1, or MD5 as arguments. Once the file is submitted, VirusTotal runs it through a series of checks, including scanning with a large number of antivirus engines and other security tools. These engines check for known malware signatures, suspicious behavior, and other indicators of compromise. VirusTotal aggregates the results from all the scanning engines and tools into a single report, which provides information about the file's detection status across different engines, as well as additional details such as file metadata. The detection outcomes are stored as a dictionary and can be saved to a JSON file. Due to the limitations of the free API, which allows only 500 samples to be submitted per day, it took us nearly two months to collect all the labels. Presently, VirusTotal handles approximately one million submissions each day. The results of each submission are then shared with the entire community of antivirus vendors who contribute their tools to the VirusTotal service. In return, vendors benefit by incorporating into their products the malware signatures of new variants that their tools have missed but a majority of other tools have flagged as malicious [23]. Popular tools such as McAfee, F-Secure, Tencent, 360, and Microsoft are included in VirusTotal and widely adopted on both laptops and mobile devices. In this study, we leverage VirusTotal to ensure the reliability of our datasets and to provide family labels and subfamily labels for the label cleansing process.

2) Label Cleanser: To counteract the inaccuracy of a single antivirus product in detecting malware executables, we integrate the results from more than 70 antivirus products to ensure the reliability of our dataset. VirusTotal aggregates results from all antivirus products into a single report for a submitted malware sample. However, each antivirus product provides labels in different formats. For instance, for the same malware sample, Bkav may label it as "W32.FamVT.CoinminerFLDTTc.Worm," Lionic might label it as "Trojan.Win32.Generic.4!e," Elastic could mark it as "malicious (high confidence)," and MicroWorld-eScan might identify it as "Trojan.GenericKD.43163708." Due to this inconsistency, direct utilization of these labels is not feasible and may yield insufficient information. Instead, we employ a regular expression to split the label strings by non-alphanumeric characters. This process breaks down each label string into a set of substrings. Subsequently, we filter out substrings that are not alphabetic and have lengths smaller than 3. This filtering process excludes numeric values and short substrings, which are typically less informative for describing malware families or subfamilies.

Moreover, for each malware sample, it will have a set of substrings representing possible family labels or subfamily labels. Although these labels may differ slightly, they should represent the same underlying concept, such as "ransom", "ransomkd", "ransomgen". To determine precise labels, we use the longest common subsequence algorithm to calculate the similarity rate among these labels and establish an appropriate threshold thr to integrate them. To guarantee that they are replaced by the same labels, we sort the labels by their lengths and use the shortest ones as replacements. We then consolidate similar names with identical meanings below the threshold, such as "pua" and "pup", both denoting potentially unwanted applications. We filter labels that are not related to describing a malware family or subfamily by matching them to the MalwareBazaar dictionary, such as "generic", "application", etc. We use substrings with the top t frequencies as potential malware family or subfamily labels. Next, we pair the substring with the highest frequency with each of the remaining $t-1$ substrings to generate candidate malware family-subfamily labels for each sample. This approach helps in excluding categories like "trojan.adware" and "pua.adware" and specifying a more specific category as "trojan", "adware", and "pua" lack specific subfamily distinctions. After collecting candidate malware family-subfamily labels for all malware samples, we employ the same algorithm to combine their labels. This yields a set of malware family-subfamily labels. Finally, after collecting and processing the malware samples, we classify them into their respective family-subfamilies categories, thus completing the creation of our dataset. Furthermore, we validate our datasets through random sampling, dynamic analysis using a prominent open-source malware analysis system for 5 minutes (Cuckoo sandbox [38]), and manual analysis like reverse engineering. This validation follows a similar method akin to [22], sampling 10 instances from categories with fewer than 150 samples and 15 from clusters with more than 150, ensuring the reliability of our dataset.

3) Program Analysis Framework: angr is a multi-architecture binary analysis toolkit capable of performing dynamic symbolic execution and various static analyses on binaries, making it very popular among researchers. We utilize angr [39] as our program analysis framework to disassemble malware binaries, generate instructions, divide them into basic blocks and functions, and extract important program metadata. angr translates binary code into human-readable assembly instructions based on the architecture's instruction set, with a direct mapping between binary codes and instructions. This process includes both opcode and operand decoding. For example, the binary sequence "83 C2 03" is disassembled into the assembly instruction "add edx, 3" based on the x86 instruction set, where "83" specifies the encoding for the "add" opcode, "C2" specifies "edx" as the destination operand, and "03" specifies "3" as the source operand. Once the instructions are recovered, angr identifies sequences of instructions with a single entry point and a single exit point as basic blocks. The entry point of a basic block is typically the first instruction in a function, the target of a jump or branch instruction, or the instruction following a jump or branch instruction. The exit point of a basic block usually includes jumps, calls, and branches.

During the generation of function metadata, we notice that some function names start with "sub" and "loc". It's important to note that all malware samples are stripped binaries, meaning their symbol tables are removed. Consequently, we cannot directly recover their original user function names; instead, they are replaced by combining "sub" or "loc" with function addresses. The remaining functions typically represent internal functions added by the framework or compiler for better analysis and program optimization, as well as Windows system functions that are not publicly available. Therefore, we only collect basic block and instruction metadata within the user functions, as the remaining functions are either not publicly available or do not affect the program's functionality. This program metadata includes the average instruction length (avg_instr), and average basic block size (avg_bb) in bytes. These metrics will be used by the semantics and feature engineering processor for further analysis.

4) Semantics And Feature Engineering Processor: Variants of malicious programs from the same family often reuse code [40], [41], [42]. Using functions as the basic unit for malware classification significantly reduces accuracy because functions can replace a function call site with the body of the called function due to compiler or manual optimizations. Additionally, accurately identifying function boundaries is inherently difficult [43], [44]. Binary analysis frameworks such as angr, IDA, and others also face challenges in accurately recovering function boundaries in binary programs, making the use of functions as a detection unit impractical. Moreover, using individual instructions as the basic unit is not effective because they lack sufficient semantics.

Therefore, we adopt basic blocks as basic units for classification. A basic block is a basic semantic and control unit for a program. It is a sequence of instructions in a program with the property that, if the first instruction in the sequence is executed, all subsequent instructions in the sequence will also be executed in sequence, and no instructions outside of the sequence will be executed until the last instruction in the sequence has been executed. In other words, a basic block represents a straight-line code sequence with no branches in or out, except at the entry and exit points. If we use basic blocks as the fundamental unit for malware classification, it can simplify the analysis process by breaking down complex programs into smaller, more manageable pieces. Analyzing basic blocks individually makes it easier to understand program behavior since each functionality typically uses a certain set of instructions. Furthermore, basic blocks provide a natural boundary for many analysis techniques because they represent units of code that can be executed atomically. This enables analysis algorithms to focus on specific portions of the program without considering the entire program at once, leading to more efficient and scalable analysis techniques. In summary, basic blocks can provide a structured approach to analyzing and understanding the behavior of computer programs.

In this paper, we present a "local self-similarity technique" that captures local self-similarities and preserves the correlations between them. Self-similarity is closely related to the concept of statistical co-occurrence of instruction sequences across programs, which is captured by Mutual Information (MI). Alternatively, internal joint instruction statistics are often computed and extracted from individual programs and then compared across programs. Most existing methods are restricted to measuring the statistical co-occurrence of instruction-wise elements (byte codes, opcodes, operands, or assembly characters) and are not easily extendable to the co-occurrence of larger, more meaningful patterns such as basic blocks. In some cases, such as with MI, this limitation is due to the curse of dimensionality. Additionally, statistical co-occurrence is often assumed to be global (within the entire program), a very strong assumption that is frequently invalid. In our approach, self-similarities are measured locally (within a block). Our framework explicitly models both local and global correlations of self-similarities. Furthermore, we use basic blocks as the fundamental unit for measuring local self-similarities, as they capture more meaningful semantic patterns than individual instructions.

We aim to compare a "template" malware variant, F, to another malware variant, G. The template malware F serves as a standard baseline, allowing us to assess similarities and identify common patterns between the two variants. Notably, F and G do not need to be the same size. While measuring similarity across objects can be complex, similarities within each malware sample can be easily revealed using simple similarity measures such as the sum of squared distances (SSD). SSD is calculated by taking the difference between each pair of instructions, squaring the difference of each byte, and summing them up. This results in local self-similarity descriptors that can be matched across malware samples. To compute local self-similarity descriptors for a basic block, we must first identify the baseline instruction. However, the fixed calling convention of the instruction set poses a challenge: comparing instructions within this convention to those nearby in the basic block can yield similar self-similarity descriptors for equivalent instructions from different basic blocks ending with call sites in another malware sample. This similarity complicates the differentiation of various malware samples. As a result, we use the center instruction in the basic block for comparison. Additionally, due to the varying sizes of basic blocks, using a uniform number of self-similarity descriptors to encode them is impractical. Analyzing each basic block individually is time-intensive (two times longer), requiring the division of large basic blocks into smaller chunks and locating their center instructions, which may not fully leverage the benefits of static analysis. Furthermore, direct comparison is hindered by the various lengths of instructions. Therefore, we utilize the integral average instruction length along with the integral average basic block size. This mean value provides a measure of the "typical" value in the dataset, incorporating the influence of extreme values related to special basic blocks pertinent to our analysis. To mitigate the impact of instruction disorder caused by using the integral average instruction length and basic block size, we densely compute local self-similarity descriptors around the malware sample. Our result in Section V shows that the similarity between malware belonging to the same family and subfamily is preserved and proves the effectiveness of densely computing local self-similarity descriptors to mitigate the impact. Each descriptor is derived from the squared sum of distances between any average instruction and the center instruction within its corresponding block (using "average instruction" and "block" to represent an integral average instruction and an integral average basic block, respectively). To manage the increasing positional uncertainty with distance and instruction disorder caused by different development environments, we employ a binned log-polar representation for each block [45].

We associate a "local self-similarity" descriptor $d_{i,j}$ with a center average instruction i and any other average instruction j within the same block b. The term "local" denotes a small portion of the malware (e.g., 2%) as opposed to the entire malware. This descriptor is calculated using the simple sum of square differences (SSD, see Eq. (1)) between their instruction bytes, where k represents an instruction byte. Local self-similarity descriptors for a block are calculated densely within a byte window. After all self-similarity descriptors are calculated, each is further transformed by the correlation calculation Eq. (2). The correlation calculation provides a quantitative measure of similarity between instructions, where a higher value indicates a higher degree of similarity. The use of the exponential function implies that small differences between instructions result in relatively large values of $C(d_{i,j})$, emphasizing robustness against minor variations while penalizing larger discrepancies more severely. This process results in a "correlation matrix" C_i for the block. The correlation matrix C_i is then transformed into log-polar coordinates and partitioned into $m * n$ bins (m angles, n radial intervals). The maximum values typically arise from the instructions that exhibit the most similar semantics, representing the most characteristic features of blocks. We select the maximal value in each bin, and these maximal values form the form the $m*n$ entries of our final local "self-similarity descriptors" vector d_i associated with the center instruction i. This approach also addresses instruction disorder as we always take the maximum values. To match an entire malware F to G, we compute the local self-similarity descriptors d_i densely throughout F and G for every block. This significantly accelerates the computation of self-similarity descriptors because the processing of each block is independent, allowing us to parallelize the computation using multi-threading. All the local self-similarity descriptors in F together form a single global "ensemble of self-similarity descriptors," which maintains their relative geometric positions. A good match of F in G corresponds to finding a similar ensemble of self-similarity descriptors in G, similar both in descriptor values and their relative geometric positions. To reduce sensitivity to outliers and improve numerical stability [46], the global "ensemble of self-similarity descriptors" d is normalized by linearly stretching their values to the range $[a, b]$ by the Eq. (3) and Eq. (4) where d_{min} is the minimum value in d and d_{max} is the maximum value in d, d_{std} is the standardized vector with values in the range $[a, b]$. These descriptors are concatenated as a self-similarity descriptor vector, which is reshaped to a size $(m{*}n, h, w)$ where h and w are the height and width of the input to the classifier before being fed into the classifier. The semantics and feature engineering process can also generate a collection of Windows feature maps for malware samples to aid analysis. This is achieved by using the recommended fixed widths for Windows feature maps with variable heights, based on the sizes of the self-similarity descriptor vectors as specified in [24]. Each vector consists of a set of local self-similarity descriptors, which are calculated and normalized for individual blocks, with each block containing three values. By multiplying each self-similarity descriptor vector by 255, we map its values to a range of 0 to 255, representing pixel values.

$$d_{i,j} = \sum_{k=1}^{n} (i_k - j_k)^2 \tag{1}$$

$$C\left(d_{i,j}\right) = exp^{-d_{i,j}} \tag{2}$$

$$d_{std} = \frac{d - d_{min}}{d_{max} - d_{min}} \tag{3}$$

$$d_{scaled} = d_{std} \cdot (b - a) + a \tag{4}$$

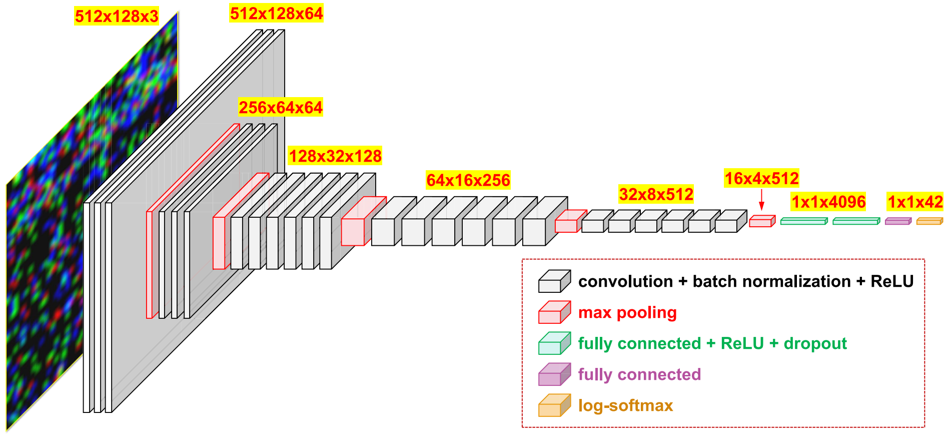


Fig. 2. The architecture of employed classifier.

Properties and benefits of the"self-similarity descriptor":

- Self-similarities are considered as a local property of a malware sample and are therefore measured locally, within the context of a block, rather than globally across the entire malware. This approach broadens the applicability of the descriptor to encompass a wide range of complex malware variants.
- By selecting the maximal correlation value in each bin, the descriptor becomes less sensitive to the precise position of the best match within that bin due to local affine deformations resulting from different development environments. As the bins adapt in size with changes in future instructions and basic blocks, this feature accommodates additional radially increasing non-rigid deformations.
- Employing blocks as the fundamental unit for assessing internal self-similarities captures more meaningful functional patterns than analyzing individual instructions alone.

5) Classifier: estimates the probabilities at which a malware sample is classified into a specific malware family and subfamily. Our goal is to identify a similar ensemble of descriptors between malware samples, which are similar both in descriptor values and in their relative geometric positions. Relative geometric positions represent the correlation between blocks. Using algorithms such as finding the largest common regions in their descriptors poses efficiency challenges and faces the curse of dimensionality. However, convolutional neural networks (CNNs) offer a more efficient solution. CNNs excel at capturing local similarity and the relationships between neighboring regions. As the network depth increases, CNNs can understand malware classification in terms of a hierarchy of concepts. Therefore, we use CNNs for the final malware classification. Specifically, we adopt the widely used VGG11 architecture. The architecture of VGG11, employed in our study for the dataset presented in Section IV, is shown in Fig. 2. It consists of six components: five convolutional blocks and one fully connected block.

The classifier depicted in Fig. 2 is built using a training dataset comprising a ensemble of descriptors of various malware programs. The input to the classifier is a 4-D vector with dimensions $(batch\_size, c, h, w)$, where $batch\_size$, c, h, and w respectively represent the number of samples per batch, channels, height, and width of an image. The initial convolutional block includes $f1$ zero-padded filters, each with a kernel size of $(k1, k1)$, a stride $s1$, channel-wise batch normalization, ReLU activation, and max-pooling. Batch normalization calculates the mean and standard deviation of the inputs within each batch of data during training and then normalizes the inputs based on these statistics. This ensures that the inputs to each layer have a similar distribution, which helps stabilize the training process and accelerates convergence. ReLU can output zero for negative input values, which can lead to sparse activations in the network. This sparsity can help reduce overfitting and improve generalization. ReLU can also help alleviate the vanishing gradient problem because it has a constant gradient of 1 for positive input values, regardless of the input magnitude. Max-pooling, with kernel size k_2 and stride $s2$, reduces spatial dimensions, retains critical information, and fosters translation invariance, enhancing the network's robustness and computational efficiency. After max-pooling, the output of this block yields Y_1^C with $f1$ channels, having dimensions $(batch\_size, f1, h1, w1)$. Subsequently, another convolutional block follows the first one, differing only in the number of zero-padded filters $f2$. Its output Y_2^C assumes dimensions $(batch\_size, f2, h2, w2)$ and serves as input to the third convolutional block. The third convolutional block comprises two layers, with the first layer lacking max-pooling and the second layer employing it. Both layers share the same number of zero-padded filters $f3$, channel-wise batch normalization, and ReLU activation. The fourth and fifth convolutional blocks follow suit, differing only in the number of zero-padded filters $(f4)$. The output of the max-pooling layer in the fifth convolutional block is flattened and forwarded to the fully connected block, which consists of three feed-forward layers. The first two layers each contain $fc1$ neurons, ReLU activation, and a dropout rate dr, while the last fully connected layer comprises $fc3$ neurons, corresponding to the number of malware families in the dataset. Its output P has a shape $(batch\_size, num\_class)$. Additionally, we employ CrossEntropyLoss to evaluate discrepancies between true labels and predictions as expressed in Eq. (5) and Eq. (6), where P_n and

Q_n represent the output probability for each malware family for a sample in the batch and its corresponding true label. We utilize Stochastic Gradient Descent as the optimizer, with a learning rate lr, to iteratively update parameters using gradient information.

$$l_n = -\sum_{i=0}^{num\_class} \log \frac{\exp(P_{n,i})}{\sum_{j=0}^{num\_class} \exp(P_{n,j})} Q_{n,i} \tag{5}$$

$$l(P, Q) = \frac{\sum_{i=1}^{N} l_n}{N} \tag{6}$$

IV. Evaluation Settings

A. Evaluation Metrics

The metrics for evaluating the performance of BBSS include *accuracy* and *efficiency*. Let $C_{i,j}$ denote the number of malware samples from family i classified into family j by the Classifier, where $i, j \in 1, 2, \cdots, N$ and N represents the total number of malware families. We can utilize the commonly used machine learning parameters TP, FN, FP, and TN to define the performance metrics of BBSS. Specifically, for a malware family i, TP_i represents the number of correctly predicted samples belonging to family i, denoted as $TP_i = C_{i,i}$; TN_i indicates the number of correctly predicted samples belonging to other families, given by $TN_i = \sum_{j=1, j\neq i}^{N} C_{j,j}$; FN_i denotes the number of misclassified samples belonging to family i, calculated as $FN_i = \sum_{j=1}^{N} C_{i,j} - C_{i,i}$; and FP_i stands for the number of samples misclassified into family i, expressed as $FP_i = \sum_{i=1}^{N} C_{j,i} - C_{i,i}$. Thus, the metrics to evaluate the performance of the Classifier can be formally represented as follows:

1) The *accuracy* is defined as the ratio of correctly predicted malware samples to the total number of malware samples:

$$accuracy = \frac{TP_i + TN_i}{TP_i + FN_i + FP_i + TN_i}. \tag{7}$$

2) The *precision* for a malware family i is calculated as the ratio of correctly classified samples to the total samples classified into this family:

$$precision = \frac{TP_i}{TP_i + FP_i}. \tag{8}$$

3) The *recall* for a malware family i is calculated as the ratio of correctly classified samples to the total samples belonging to this family:

$$recall = \frac{TP_i}{TP_i + FN_i}. \tag{9}$$

4) The *F1* score for a malware family i is computed as the balanced average of precision and recall:

$$F1 - score = \frac{2 * (recall * precision)}{recall + precision}. \tag{10}$$

We also evaluate the efficiency of our BBSS framework. As BBSS operates in a time-sensitive and space-sensitive environment, its time efficiency, denoted by *avg_time*, can be assessed by the processing time per malware sample, which includes both feature extraction and classification. Let *total_time* represent the total clock ticks used to process num_files malware samples. We can compute *avg_time* as:

$$avg\_time = \frac{total\_time}{num\_files}. \tag{11}$$

In addition to time efficiency, we also evaluate the performance of BBSS based on the size of the dataset used for training and testing.

B. Experiment Setup

1) Equipment and Dataset: The PC employed by BBSS is equipped with 32 logical processors, 24 kernels, and an installed RAM with a 32.0GB available memory running the 64-bit Microsoft Windows 11 Home operating system. Each processor is configured with 13th Gen Intel(R) Core(TM) i9-13900 CPU @3.0GHz, 3000Mhz. The versions of PyTorch, Pillow and angr to develop our framework are 2.3.0, 10.0.1 and 9.2.74, respectively. Additionally, we collect the malware samples from VirusShare across seven years from 2017 to 2023 [47]. We filter all malware samples that are not Windows executables and not detected by more than 10 anti-virus products, and do not have complete program structures. Therefore, our dataset reflects the trend of Windows malware in recent years. This dataset is comprised of 25739 malware samples in total, which belongs to 40 malware families with a varying number of malware files per family, as detailed in TABLE I. Adware and trojan are major types of malware in the wild. The reason is that adware and trojans are often designed with profit motives in mind. Adware generates revenue through advertisements, while trojans can be used for various malicious purposes such as stealing sensitive information or facilitating other cybercrimes. The potential for financial gain incentivizes attackers to create and distribute these types of malware more widely. Additionally, potentially unwanted applications (PUAs) are also stepped into our eyes. They are usually bundled with legitimate software. Users may inadvertently install PUAs while installing the main software they intended to download.

2) Parameters: The number of considered substrings t is set to 3, as malware samples combining functionality from more than three families are typically identified as new variants [48]. These variants are less likely to be active in the wild and more likely to be detected by antivirus products. We set the threshold thr to 0.75 based on statistical similarity information between substrings. To optimize the efficiency of the semantics and feature engineering processor, we consider the average instruction length, which is 2.86 bytes, and decide to use three bytes as the comparison unit. Similarly, we use a region of 15 bytes for comparison, as using a uniform number of self-similarity descriptors to encode them is impractical and examining every basic block in malware programs would be time-consuming. Consequently, m and n are set to 1 and 3, respectively, to produce Windows feature maps. In identifying

TABLE I
BBSS MALWARE DATASET

Family_Subfamily	No. of Files	Family_Subfamily	No. of Files
adware.bundler	337	trojan.installerex	574
adware.domaiq	465	trojan.kryptik	85
adware.ibryte	151	trojan.loadmoney	216
adware.imali	1209	trojan.morstar	126
adware.multiplug	3132	trojan.msil	1884
adware.outbrowse	653	worm.mydoom	121
adware.softpulse	217	pua.playtech	167
worm.allaple	561	pua.softpulse	75
trojan.antifw	399	pua.syncopate	171
trojan.autorun	109	virus.virlock	233
worm.autorun	98	pua.toolbar	392
trojan.backdoor	172	trojan.python	153
trojan.downloader	561	trojan.ransom	1349
trojan.dridex	4995	virus.sality	147
trojan.ekstak	203	trojan.servstart	118
trojan.emotet	3082	trojan.startpage	402
virus.expiro	1919	trojan.unruy	103
trojan.fareit	119	trojan.virus	248
trojan.gandcrab	107	trojan.wabot	88
pua.inbox	444	trojan_worm	154

TABLE II
FRAMEWORK PARAMETERS

Param.	Implication	Value	Param.	Implication	Value
t	# of substr.	3	s1	stride	1
thr	threshold	0.75	h1	height	64
avg_instr	avg instr. len.	2.86	w1	width	256
avg_bb	avg bb. len.	14.92	f2	# of filters	128
m	# of angles	1	h2	height	32
n	# of intervals	3	w2	width	64
c	channels	3	f3	# of filters	256
h	height	128	f4	# of filters	512
w	width	512	dr	drop rate	0.5
batch_size	batch size	32	fc1	# of neurons	4096
f1	# of filters	64	fc3	# of neurons	40
k1	kernel size	3	lr	learning rate	0.01

a suitable classifier structure, we experimented with various configurations of the popular VGG network, ranging from VGG11 to VGG19. It was determined that VGG11 struck the optimal balance between accuracy and efficiency. The constructed classifier comprises five convolutional blocks and one fully connected block, with detailed model parameters presented in TABLE II. Since the VGG network requires a fixed input size, we surveyed the distribution of shapes for Windows feature maps. We found that when the width and height reach 512 and 128, respectively, a majority of Windows feature maps are accommodated. Therefore, w and h are set to 512 and 128, respectively. Given that our dataset has highly unbalanced samples across classes, we assigned a higher probability of selection to samples from minority classes for each batch to ensure their representation. The probability of a sample being selected is inversely proportional to the ratio of the number of samples in its category to the total number of samples. This means that samples from minority categories have a higher probability of being selected, while samples from majority categories have a lower probability. 90% of the samples in each class in our dataset are used for training, while the remaining are used for testing. Our VGG11 model is trained for 100 epochs.

TABLE III
CLASSIFICATION REPORT

Category	Precision	Recall	F1 Score	Accuracy
worm.allaple	100%	100%	100%	100%
trojan.msil	96.6%	96.6%	96.6%	96.6%
virus.virlock	94.1%	100%	97%	100%
adware.softpulse	100%	100%	100%	100%
pua.softpulse	100%	100%	100%	100%
trojan.kryptik	75%	60%	66.7%	60%
trojan.wabot	33.3%	66.7%	44.4%	66.7%

V. EVALUATION RESULTS

A. Accuracy

We first evaluate the accuracy of BBSS on the dataset and then compare BBSS with several other methods using different techniques. One can observe from TABLE IV that the average accuracy, precision, recall, and F1 score of BBSS are 98.1%, 98.2%, 98.1%, and 98.1%, respectively. Among the 40 malware categories, over 85% are accurately characterized, with an accuracy of 90% or higher. It is worth noting that malware samples utilizing anti-emulation, anti-debugging, code obfuscation, multi-threaded, or polymorphic techniques, such as worm.allaple, trojan.msil, and virus.virlock, can also be detected with very high accuracy of over 96.6% (see TABLE III). Additionally, malware samples from the same subfamily but different families, such as pua.softpulse and adware.softpulse, achieve an accuracy of 100%. However, some malware samples in specific categories, such as trojan.wabot and trojan.kryptik, exhibit relatively lower accuracy at 66.7% and 60%, respectively. Trojan.wabot is a type of malware that specifically targets the WhatsApp messaging platform and allows unauthorized access to infected systems. Analyzing the disassembly results for trojan.wabot samples using angr revealed an interesting scenario: antivirus products classify two main types of malware under this category. The first type hides itself using a packer, enabling attackers to remotely control the infected device, execute commands without the user's knowledge, and modify registry keys to run automatically. The second type redirects users to websites that download trojan.wabot applications. This discrepancy between antivirus product definitions creates a definition gap. Additionally, as shown in the Fig. 4, the left and right images correspond to the first and second types of malware, respectively, and their Windows feature maps differ significantly. Consequently, our model achieves low accuracy in this category. To understand why our model struggles with trojan.kryptik samples, we conducted reverse engineering on samples with very different Windows feature maps. We identified three main types of malware in this category. The first type behaves normally, the second uses a packer to hide its code, and the third attaches the trojan.kryptik application processed by another packer to its data sections. These variations result in significantly different textures in their Windows feature maps, presenting challenges for our solution in distinguishing samples processed by different packers. While developing a generic unpacker would address this issue, it falls outside the current scope of our work. Consequently, this remains a limitation of our approach.

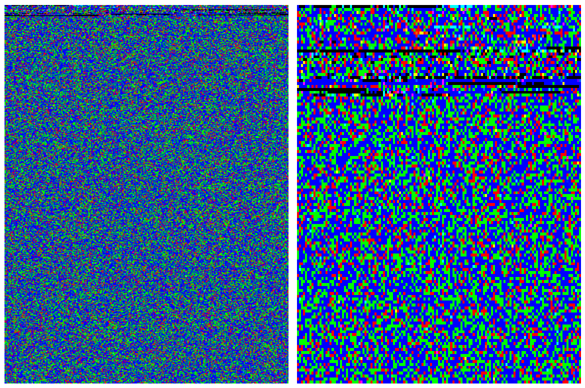


Fig. 3. Trojan.wabot sample1 (left) and trojan.wabot sample2 (right).

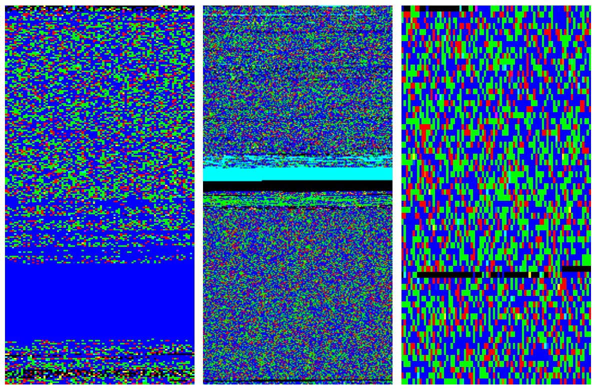


Fig. 4. Trojan.kryptik sample1 (left), trojan.kryptik sample2, and trojan.kryptik sample3 (right).

TABLE IV
ACCURACY ON MALFINER

	Precision	Recall	F1	Accuracy
DNN [49]	0.02%	1.5%	0.04%	1.5%
KNN [50]	92.7%	89%	88.8%	89%
CNN [19]	1.6%	12.7%	2.8%	12.7%
Autoencoder [18]	0.08%	2.9%	0.16%	2.9%
CNN_Attention [20]	3.8e-4%	0.2%	7.5e-4%	0.2%
DRBA [21]	96.8%	96.5%	96.6%	96.5%
GIST [24]	95.7%	95.2%	95.3%	95.2%
VisMal [25]	97.8%	98%	97.8%	97.8%
MDMC [32]	68.6%	64.2%	63.4%	64.2%
VirusTotal [51]	-	-	-	16.5%
LBP [33]	96%	96.1%	96%	96%
bb_img [34]	95%	94.7%	93.9%	94.7%
bb_ngram [35]	90.9%	90.9%	89.5%	90.9%
VGG11_orig	98%	97.8%	97.8%	97.8%
BBSS	98.2%	98.1%	98.1%	98.1%

TABLE IV presents the comparison results among different classification methods applied to our dataset. Due to the unavailability of source codes, we re-implemented the most representative algorithms proposed by other researchers. These algorithms utilize various data visualization techniques to target different aspects of malware features for classification. Specifically, the Deep Neural Network (DNN) [49], K-Nearest Neighbors (KNN) [50], shallow CNN [19], autoencoder [18], and CNN with attention technique [20] focus on employing advanced machine learning algorithms to capture unique features in malware images. These features include linear relationships between malware bytes, similarity of data points through distance comparisons, local features with simple abstraction, reconstruction error, and the higher importance of certain bytes. Nearly all of the classification methods failed to distinguish malware samples from different categories. Specifically, the CNN with attention technique exhibited a notably low accuracy of 0.2%, along with very low precision and F1 score. This suggests that there are no directly important malware byte codes for classification, as our dataset is complex and contains malware samples belonging to the same families or subfamilies, inherently sharing codes. However, KNN performed relatively well with an accuracy of 89%, indicating evident similarities among malware samples within the same category. This suggests that such similarities contribute positively to classification accuracy. Their performance is inferior to that of VGG11_orig, which achieved an accuracy of 97.8% by applying VGG11 directly to malware samples for classification. This demonstrates VGG11's strong learning abilities and its effectiveness in distinguishing malware samples from different categories.

Other algorithms, such as DRBA [21], GIST [24], and VisMal [25], employed local or global data augmentation techniques to improve the quality of malware images by addressing specific aspects of visual variation, such as frequencies, orientations, rotation, translation, and image magnification. The local data augmentation technique used by VisMal achieves a relatively higher accuracy of 97.8%. In contrast, DRBA and GIST, which apply global data augmentation techniques, achieve accuracies of 96.5% and 95.2%, respectively. These results suggest that maintaining similarity among different malware samples is crucial for effective classification. While these results are noteworthy, they are lower than the accuracy of our approach, demonstrating the superiority of our method. For further comparison, LBP [33] considers the local numerical magnitude relationships across malware, while MDMC [32] emphasizes the mutual information between neighboring byte codes across malware. Similarly, the approach bb_img [34] focuses on the order and sequences of opcodes in basic blocks to obtain hash bits used as features, while bb_ngrams combines abstracted assembly instructions and mutual information between instructions. MDMC and LBP show very different results: MDMC achieves an accuracy of 64.2%, while LBP achieves an accuracy of 96%. This difference is because MDMC considers the overall correlation across the entire malware sample, whereas LBP focuses on the local correlation within a block. Additionally, bb_img and bb_ngram achieve an accuracy of 94.7% and 90.9%, as they preserve similarity among malware instances to some extent. However, they also introduce more false positives due to only collecting opcodes and instruction abstraction. The performance of antivirus products in VirusTotal, with an average accuracy of 16.5%, highlights a limitation in most antivirus products to provide family and subfamily classifications for malware simultaneously. This deficiency suggests that individual products may not offer sufficiently detailed information to facilitate reliable measures for recovering from attacks. In contrast, our method achieves the highest accuracy, precision, recall, and F1 score. This demonstrates the effectiveness of preserving similarity between malware samples by using internal self-similarities and their correlations for detecting malware. Our approach proves to be a reliable method for

TABLE V
TIME AND SPACE EFFICIENCY

	Storage	avg_time
DNN [49]	14.7GB	26.4ms
MDMC [32]	1.7GB	50.3ms
KNN [50]	14.7GB	37.5ms
CNN [19]	14.7GB	23.8ms
Autoencoders [18]	14.7GB	165.8ms
CNN_Attention [20]	14.7GB	26.6ms
DRBA [21]	14.7GB	167.5ms
GIST [24]	14.7GB	49.3ms
VisMal [25]	16.3GB	30ms
LBP [33]	14.7GB	319.8ms
bb_img [34]	8.66GB	10s
bb_ngram [35]	210.9GB	11.9s
VGG11_orig	14.7GB	26ms
BBSS	0.5GB	9.7s

malware detection, showing promising potential in thwarting attacks.

B. Time and Space Efficiency

Considering the costs associated with storage and computing resources, it is economically impractical to retain original samples for model retraining to accommodate future changes. Moreover, given the time-sensitive nature of malware classification within anti-virus systems, it is crucial that the process of identifying malware samples does not introduce significant delays. Even minor delays could result in missed opportunities to detect malicious processes. Therefore, we evaluate the performance of our solution based on storage and time efficiency. TABLE V compares various malware classification methods in terms of storage requirements and average processing time. Most methods require substantial storage of 14.7GB, with bb_ngram demanding the most at 210.9GB. In contrast, our method, BBSS, stands out with its minimal storage requirement of 0.5GB, saving over 96% of space compared to most other methods. To evaluate the efficiency of our proposed framework, we measured the CPU time required for our dataset to be processed by BBSS during testing, calculating the average Mean Processing Time (MPE) per malware sample. Our framework is designed for practical deployment, avoiding reliance on server infrastructure or GPU utilization. When it comes to efficiency, the processing times of various methods vary significantly, with bb_img and bb_ngram taking 10 seconds and 11.9 seconds per sample, respectively, while others range from 23.8 milliseconds (CNN) to 319.8 milliseconds (LBP). Although BBSS has a processing time of 9.7 seconds, it is substantially shorter than the 120 seconds typically required for dynamic analysis [14]. In summary, BBSS's superior storage efficiency and competitive processing time underscore its effectiveness and reliability in malware detection, making it a promising solution compared to traditional methods. This combination of low storage requirements and reasonable processing times ensures that BBSS can be seamlessly integrated into time-sensitive malware detection workflows, providing a practical and scalable solution for real-world deployment.

VI. CONCLUSION

In this paper, we present a new malware dataset, which comprises over 25,000 samples collected over seven years. This dataset offers a refined level of categorization, encompassing both malware families and subfamilies. It has undergone rigorous validation by more than 70 third-party antivirus products and includes random sampling for dynamic and manual analysis. This comprehensive dataset more accurately reflects current trends in malware attacks. We also introduce a context-sensitive malware visualization classification framework called BBSS. This framework excels at distinguishing between different malware subfamilies, even within the same family, and different malware families, even within the same subfamily–a capability often lacking in current antivirus products on VirusTotal. Furthermore, BBSS significantly reduces the overhead of model training while preserving the similarities among malware samples within the same families and subfamilies.

Additionally, our approach differs from many existing methods that emphasize advanced image processing or machine learning algorithms for malware classification. Instead, we return to the foundational principles of malware classification, focusing on detecting similarities between malware samples. Our method captures the internal semantics and similarities within blocks, as well as the correlations between them. Unlike methods that rely on overall statistical distributions and local geometric information of byte codes, our approach is inherently more resistant to changes because the basic block serves as the fundamental unit of program functionality.

REFERENCES

[1] S. Morgan. (Nov. 13, 2020). *Cybercrime to Cost the World $10.5 Trillion Annually By 2025*. [Online]. Available: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025

[2] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proc. 12th USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2003, pp. 1–19.

[3] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for Android using systematic mutation," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2018, pp. 1263–1280.

[4] F. Gagnon and F. Massicotte, "Revisiting static analysis of Android malware," in *Proc. 10th USENIX Workshop Cyber Secur. Experimentation Test*, Vancouver, BC, USA, Aug. 2017, pp. 1–8.

[5] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti, "Humans vs. machines in malware classification," in *Proc. 32nd USENIX Secur. Symp.*, Anaheim, CA, USA, Aug. 2023, pp. 1145–1162.

[6] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable malware clustering based on static features," in *Proc. USENIX Annu. Tech. Conf.*, San Jose, CA, Jun. 2013, pp. 187–198.

[7] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: Dynamic malware analysis without feature engineering," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, New York, NY, USA, Dec. 2019, pp. 444–455.

[8] F. Zhong, X. Cheng, D. Yu, B. Gong, S. Song, and J. Yu, "MalFox: Camouflaged adversarial malware example generation based on conv-GANs against black-box detectors," *IEEE Trans. Comput.*, vol. 73, no. 4, pp. 980–993, Jan. 2023.

[9] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "LibScan: Towards more precise third-party library identification for Android applications," in *Proc. 32nd USENIX Secur. Symp.*, Anaheim, CA, USA, Aug. 2023, pp. 3385–3402.

[10] H. Seo and M. Yoon, "Generative intrusion detection and prevention on data stream," in *Proc. 32nd USENIX Secur. Symp.*, Anaheim, CA, USA, Aug. 2023, pp. 4319–4335.

[11] D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving Python package index malware scanning," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 499–511.

[12] A. Coscia, V. Dentamaro, S. Galantucci, A. Maci, and G. Pirlo, "YAMME: A YAra-byte-signatures metamorphic mutation engine," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 4530–4545, 2023.

[13] S. Li et al., "PackGenome: Automatically generating robust YARA rules for accurate malware packer detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2023, pp. 3078–3092.

[14] L. Cui, J. Cui, Y. Ji, Z. Hao, L. Li, and Z. Ding, "API2Vec: Learning representations of API sequences for malware detection," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2023, pp. 261–273.

[15] C. Gao, G. Huang, H. Li, B. Wu, Y. Wu, and W. Yuan, "A comprehensive study of learning-based Android malware detectors under challenging environments," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, New York, NY, USA, Feb. 2024, pp. 1–13.

[16] S. Dambra et al., "Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2023, pp. 60–74.

[17] K. Lucas, S. Pai, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif, "Adversarial training for raw-binary malware classifiers," in *Proc. 32nd USENIX Secur. Symp.*, Anaheim, CA, USA, 2023, pp. 1163–1180.

[18] X. Jin, X. Xing, H. Elahi, G. Wang, and H. Jiang, "A malware detection approach using malware images and autoencoders," in *Proc. IEEE 17th Int. Conf. Mobile Ad Hoc Sensor Syst. (MASS)*, Dec. 2020, pp. 1–6.

[19] X. Xiao and S. Yang, "An image-inspired and CNN-based Android malware detection approach," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, May 2019, pp. 1259–1261.

[20] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Malware analysis of imaged binary samples by convolutional neural network with attention mechanism," in *Proc. 10th ACM Workshop Artif. Intell. Secur.*, New York, NY, USA, Nov. 2017, pp. 127–134.

[21] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3187–3196, Jul. 2018.

[22] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti, "A close look at a daily dataset of malware samples," *ACM Trans. Privacy Secur.*, vol. 22, no. 1, pp. 1–30, Jan. 2019.

[23] *Analyse Suspicious Files, Domains, Ips and Urls to Detect Malware and Other Breaches, Automatically Share Them With the Security Community*. Accessed: Feb. 14, 2024. [Online]. Available: https://www.virustotal.com/gui/home/upload

[24] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proc. 8th Int. Symp. Visualizat. Cyber Secur.*, New York, NY, USA, Jul. 2011, pp. 1–7.

[25] F. Zhong, Z. Chen, M. Xu, G. Zhang, D. Yu, and X. Cheng, "Malware-on-the-brain: Illuminating malware byte codes with images for malware classification," *IEEE Trans. Comput.*, vol. 72, no. 2, pp. 438–451, Feb. 2023.

[26] M. Torralba and F. Rubin, "Context-based vision system for place and object recognition," in *Proc. 9th IEEE Int. Conf. Comput. Vis.*, Sep. 2003, pp. 273–280.

[27] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *Int. J. Comput. Vis.*, vol. 42, no. 3, pp. 145–175, 2001.

[28] A. Makandar and A. Patrot, "Image-based malware classification using ensemble of CNN architectures (IMCEC)," *Oriental J. Comput. Sci. Technol.*, vol. 92, pp. 400–406, Feb. 2017.

[29] S.-C. Hsiao, D.-Y. Kao, Z.-Y. Liu, and R. Tso, "Malware image classification using one-shot learning with Siamese networks," *Proc. Comput. Sci.*, vol. 159, pp. 1863–1871, Jan. 2019.

[30] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng, "Image-based malware classification using ensemble of CNN architectures (IMCEC)," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101748.

[31] Z. Bala et al., "Transfer learning approach for malware images classification on Android devices using deep convolutional neural network," *Proc. Comput. Sci.*, vol. 212, pp. 429–440, Jan. 2022.

[32] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on Markov images and deep learning," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101740.

[33] Y.-S. Liu, Y.-K. Lai, Z.-H. Wang, and H.-B. Yan, "A new learning approach to malware classification using discriminative feature extraction," *IEEE Access*, vol. 7, pp. 13015–13023, 2019.

[34] J. Chen, "A malware classification method based on basic block and CNN," in *Proc. 27th Int. Conf. Neural Inf. Process. (ICONIP)*, Bangkok, Thailand. Springer, Nov. 2020, pp. 275–283.

[35] F. Adkins, L. Jones, M. Carlisle, and J. Upchurch, "Heuristic malware detection via basic block comparison," in *Proc. 8th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2013, pp. 11–18.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2014, pp. 1–14.

[37] *Malwarebazaar Database*. Accessed: Jul. 14, 2023. [Online]. Available: https://bazaar.abuse.ch/browse/

[38] *Cuckoo Sandbox Overview*. Accessed: May 14, 2024. [Online]. Available: https://www.varonis.com/blog/cuckoo-sandbox

[39] Y. Shoshitaishvili et al., "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[40] A. Calleja, J. Tapiador, and J. Caballero, "The MalSource dataset: Quantifying complexity and code reuse in malware development," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 12, pp. 3175–3190, Dec. 2019.

[41] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2017, pp. 1691–1708.

[42] J.-W. Jang, J. Yun, J. Woo, and H. K. Kim, "Andro-profiler: Anti-malware system based on behavior profiling of mobile malware," in *Proc. 23rd Int. Conf. World Wide Web*, New York, NY, USA, Apr. 2014, pp. 737–738.

[43] X. Meng and B. P. Miller, "Binary code is not easy," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2016, pp. 24–35.

[44] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. 25th USENIX Secur. Symp.*, Austin, TX, USA, Aug. 2016, pp. 583–600.

[45] S. Belongie, J. Malik, and J. Puzicha, "Shape matching and object recognition using shape contexts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 4, pp. 509–522, Apr. 2002.

[46] D. Singh and B. Singh, "Investigating the impact of data normalization on classification performance," *Appl. Soft Comput.*, vol. 97, Dec. 2020, Art. no. 105524.

[47] *Virusshare.com—Because Sharing is Caring*. Accessed: Feb. 14, 2023. [Online]. Available: https://virusshare.com/

[48] (2020). *Malware Classification Guide*. [Online]. Available: https://any.run/cybersecurity-blog/malware-classification-guide/

[49] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, 2015, pp. 11–20.

[50] I. B. A. Ouahab, M. Bouhorma, A. A. Boudhir, and L. El Aachak, "Classification of grayscale malware images using the K-nearest neighbor algorithm," in *Innovations in Smart Cities Applications Edition 3*, M. B. Ahmed, A. A. Boudhir, D. Santos, M. El Aroussi, and I. R. Karas, Eds., Cham, Switzerland: Springer, 2020, pp. 1038–1050.

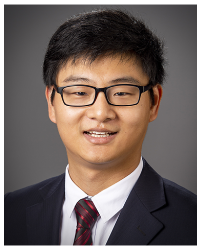
[51] *How It Works*. Accessed: Feb. 14, 2024. [Online]. Available: https://docs.virustotal.com/docs/how-it-works

Fangtian Zhong (Member, IEEE) received the Ph.D. degree in computer science from The George Washington University in 2021. After that, he was a Post-Doctoral Scholar with The Pennsylvania State University and the University of Notre Dame, respectively. He is an Assistant Professor with Montana State University. His research primarily focuses on software security, program analysis, and machine learning for cybersecurity. He is a member of ACM.

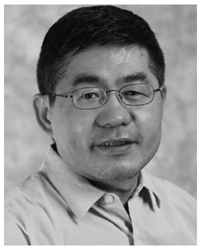
Qin Hu (Member, IEEE) received the Ph.D. degree in computer science from The George Washington University in 2019. She is currently an Assistant Professor with the Department of Computer Science, Georgia State University. Her research interests include wireless and mobile security, edge computing, blockchain, and federated learning. She has served as an editor/guest editor for several journals, the TPC/publicity co-chair for several workshops, and a TPC member for several international conferences.

Yili Jiang (Member, IEEE) received the Ph.D. degree in computer engineering from the University of Nebraska-Lincoln, NE, USA, in 2022. She is currently an Assistant Professor with the Department of Computer Science, Georgia State University, GA, USA. Her research interests include cybersecurity, machine learning, cloud/edge computing, and wireless networks.

Jiaqi Huang (Member, IEEE) received the Ph.D. degree in computer engineering from the University of Nebraska-Lincoln, NE, USA, in 2020. He is currently an Assistant Professor with the Department of Computer Science and Cybersecurity, University of Central Missouri, MO, USA. His research interests include cybersecurity, applied cryptography, machine learning, connected autonomous vehicles, and wireless networks.



Cheng Zhang received the master's and Ph.D. degrees in computer science from The George Washington University, Washington, DC, USA, in 2017 and 2020, respectively. He is currently an Assistant Professor with the Paul and Virginia Engler College of Business, West Texas A&M University, Canyon, TX, USA. His research interests include social network privacy, the Internet of Things, and the applications of machine learning.



Dinghao Wu received the Ph.D. degree in computer science from Princeton University in 2005. He is currently a Professor with the College of Information Sciences and Technology, The Pennsylvania State University. He carried out research in cybersecurity and software systems, including software security, software protection, software analysis and verification, software engineering, and programming languages. He held the PNC Technologies career development professorship, from 2017 to 2020. He was a Visiting Professor with EPFL, Switzerland, from 2018 to 2019. He was a Research Engineer with Microsoft in the Center for Software Excellence and later the Windows Azure Division from 2005 to 2009. He received the NSF CAREER Award, the George J. McMurtry Junior Faculty Excellence in Teaching and Learning Award, and the College Junior Faculty Excellence in Research Award.