IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 15, 2021, accepted August 2, 2021, date of publication August 9, 2021, date of current version August 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3103186

Version-Wide Software Birthmark via Machine Learning

CHIH-KO CHUNG AND PI-CHUNG WANG, (Member, IEEE)
Department of Computer Science and Engineering, National Chung Hsing University, Taichung 402, Taiwan

Corresponding author: Pi-Chung Wang (pcwang@nchu.edu.tw)

This work was supported in part by the Ministry of Science and Technology, Taiwan, under Grant 109-2221-E-005-046.

ABSTRACT Identifying the credibility of executable files is critical for the security of an operating system. Modern operating systems rely on code signing, which uses a default-valid trust model, for executable files to identify their publishers. A malware could pass software validation of operating systems and security software by using counterfeit code-signing certificates. Although the counterfeit certificates can be revoked by CAs, the previous research showed that the revocation delay takes as long as 5.6 months. In this paper, we attempt to identify the credibility of software with multiple-version executable files without relying on public key infrastructure (PKI), where a new-version executable file is usually developed incrementally based on the previous versions. The sharing features among different versions can be extracted for identifying the software. Accordingly, we present a software-birthmark scheme to serve our purpose. Our scheme generates a cross-version software birthmark for executable files of the same software. The proposed software birthmark is a binary-classification model of a machine learning algorithm based on imported and exported function names extracted from different-version executable files. To evaluate the performance of version-wide software birthmarks, our experiments include 138 versions of Windows kernel32.dll and 545 versions of firefox.exe. We also use multiple machine learning algorithms for performance comparisons. The results show that proposed software birthmark can effectively identify the derivations of these executable files. The proposed software birthmark can be used by operating systems or security software to evaluate the credibility of executable files with suspicious certificates.

INDEX TERMS Software birthmark, executable file format, Authenticode, digital signature, machine learning, content digest, import/export address table.

I. INTRODUCTION

Code signing [1] is a process of digitally signing an executable file to confirm the software publishers. The process also ensures that the file would not be altered or corrupted after it is signed. For example, Microsoft Authenticode allows developers to embed the publisher information in their programs through the use of digital signatures. It binds an executable file to the identity of a software publisher by using Public-Key Cryptography Standards (PKCS) #7 and X.509 certificates. The digital-signature protocol is performed upon a cryptographic digest generated by a one-way hash of the file. Both the hashing and digital signature algorithms are agreed upon beforehand. The existing practices may use SHA-1 or SHA-256 to generate the cryptographic digest.

The associate editor coordinating the review of this manuscript and approving it for publication was Ting Wang.

Authenticode is facing several threats, including fraudulent digital signatures, hash-collision attacks, and malicious-code insertions. It is possible to generate a fake certificate or purchase a good certificate from the dark market for illegal purposes [2]. Papp *et al.* [3] listed several certified malware programs, e.g., Stuxnet, Duqu, and Flame. These programs are digitally signed such that they appear to originate from legitimate software publishers. Lawrence has stated that the effectiveness of code signing depends on the security of the underlying PKI [4]. Since Windows and other modern operating systems trust signed software for installation and execution [5], a fake-signed executable file could bypass validation mechanisms of operating systems and security software to cause damages. A recent report revealed that the total number of malicious signed binaries has been increased by 12 million during the fourth quarter of 2020 [6]. Although the compromised code-signing certificates can be revoked, the average revocation delay is 5.6 months after these certificates were

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

first used to sign malware [7]. The hash-collision attacks may allow two unrelated files to share the same digital signature. The previous studies have shown the procedures to make hashing collisions on MD5, SHA-1 and related hash families [8]–[13]. Advanced hash algorithms, such as SHA-256 or SHA-3, are thus required to address this issue. In addition, an executable file could potentially be modified without compromising the validity of the certificate. Malicious codes could be appended to the end of the certificate table without being detected [14]. This is because the hash calculation of operating systems does not include the certificate table. The malicious codes could be loaded into memory with the infected executable file to result in potential risks. We are thus motivated to identify software credibility without relying on PKI.

Both software birthmark and watermark can be used to identify the content of an executable file without relying on the existing PKI. Software birthmark is generated to indicate the inherent characteristics of a program, and software watermark is used to detect illegal distribution by embedding additional information into the original file. However, software watermark of a file could be removed or altered to invalidate its usage [15]. In addition, software watermarks lack of the flexibility of renewing released files. In contrast, software birthmark provides better flexibility since its calculation and detection can be performed remotely and dynamically. The related research of software birthmark focuses on extracting either static or dynamic characteristics from one single program, whose characteristics can be preserved, even after the program is transformed. Software theft and piracy detection are the main applications of the existing software-birthmark algorithms [16].

In this paper, we propose a software-birthmark scheme for the software with multiple-version executable files, where each executable file may provide new features, fix bugs or patch security holes over its previous-version executable files. Since each executable file is usually developed incrementally based on previous versions [17], the features shared by these files can be used for identifying them. Accordingly, we extend the application of software birthmarks to support cross-version program detection. Our scheme generates one software birthmark to characterize different-version executable files based on their IAT (import address table) and EAT (export address table), where the birthmark is a binary-classification model generated by using machine learning algorithm. The birthmark can then be used to validate whether a file is a different-version executable file of the same program. The proposed scheme of version-wide software birthmark (VWSB) could identify the credibility of an executable file without relying on PKI. We use the portable executable (PE) format for Microsoft executable files to demonstrate the performance of our scheme. Two notable programs, kernel32.dll and firefox.exe, are used in our experiments. The first program, kernel32.dll, provides the kernel functions of the Windows operating systems. Another program, firefox.exe, is a notable open-source web browser. We conduct the experiments upon 138 kernel32.dll files and 545 firefox.exe files of different versions. Multiple machine learning algorithms are employed to analyze the extracted features of these PE files. The results show that our scheme can achieve high accuracy of identifying the derivations of these PE files. The contributions of this work is summarized as below.

- We present the first scheme of generating cross-version software birthmarks by using machine learning algorithms. The proposed software birthmark is a binary-classification model for identify whether an executable file is a different-version PE file of the same program. It is not an alternative to code signing and does not rely on PKI.
- We develop a procedure for extracting commonly available features from unstripped PE files. These features are input into the machine learning algorithms for training to generate a binary-classification model. The experimental results show that the models generated by several machine learning algorithms can identify cross-version PE files with high accuracy.
- Various versions of two notable programs, kernel32.dll and firefox.exe, are collected for evaluation, where the oldest kernel32.dll is released in 1993 and the oldest firefox.exe is released in 2004. The evaluation shows that our scheme is feasible for software even with a long history of development.

This paper is organized as follows. We first introduce the background of code signing and summarize the related work in Section II. In Section III, we present the proposed software birthmark with IAT and EAT features based on machine learning algorithms. Section IV describes the experiments and results of validating the proposed scheme. Finally, we conclude this work in Section V.

II. BACKGROUND

A. CODE SIGNING

Digital signature has long been employed to avoid potential attacks such as dynamic-link-library (DLL) hijacking or module replacement [18]. Authenticode, the code-signing solution released by Microsoft, embeds digital signature into the PE format. Fig. 1 shows the format of Windows PE and that of the Authenticode signature. The Authenticode signature uses both PE file hash value (contentInfo in Fig. 1) and the signed hash value to detect the origin and integrity of an executable file. Only the PE files with valid signatures are approved to install and execute. Other modern operating systems, including macOS, Linux, Android and iOS, also support enforcing signed-software execution [5].

A PE file with a valid digital signature is completely trusted, and the trustworthiness of third-party software relies on the code-signing PKI. The PKI includes certificate authorities (CAs) that issue certificates to software publishers for identification. Software publishers use these certificates to sign the software to be published, and users verify these signatures to decide whether the software is trustable.

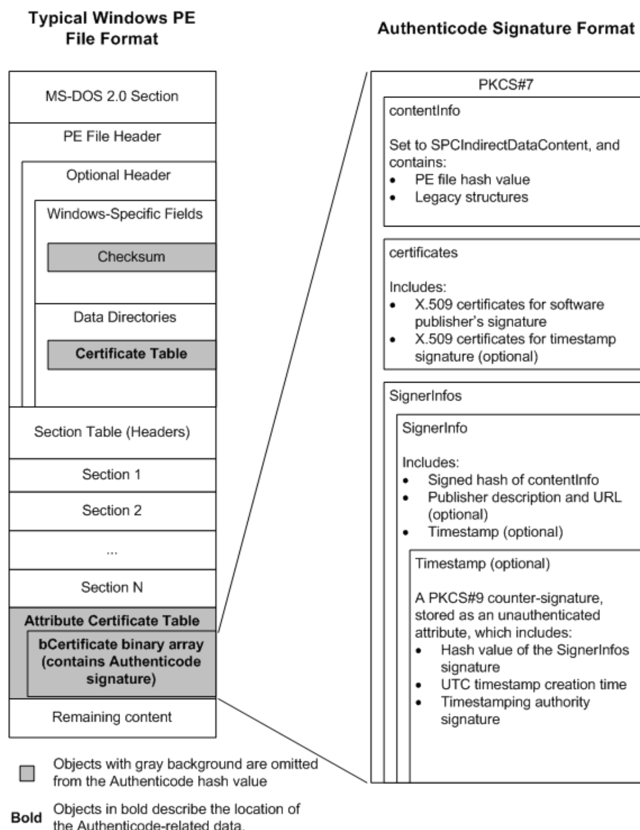


FIGURE 1. The formats of windows PE and authenticode signature [19].

An attacker may destroy the code signing certificate by generating valid signatures for malware. Papp *et al.* [3] shared a predicament of the existing PKI. The existing PKI involves many participants in numerous countries with different CAs and software publishers. Due to the fact that the infrastructure has a multitude of procedures and practices, enforcing common rules in the complex infrastructure to meet the same standard is thus difficult. For example, there are many implicitly trusted root certificates. The existing PKI cannot detect key leakage and fake certificates [20]. The cases of key leakage and fake certificates are described as below.

- **Mismanagement of Publisher Keys** Top-level certificates can be acquired from dark web marketplaces [21]. Certificates could be stolen from legitimate organizations due to system compromise from a malware infection. Kim *et al.* [22] identified 72 compromised certificates. These certificates come from eight publishers, and five of them were not aware of the abuse initially. The authors also analyzed a malware family that infects developer machines and embeds malicious code into files to be compiled and signed. Alsaid and Mitchell elaborated on risks caused by inserting self-issued public key into the list of root public keys of a computer [20]. The certificates issued by some publishers, including Digicert, Symantec, and Verisign, were used to sign malware [23]. CopyKittens, Suckfly, Turla, Regin, and Destover are well-known malware with signatures stolen from software publishers. Papp *et al.* [3] also listed several certified malware.
- **Verification Failures of CAs** CAs may issue certificates to attackers that mimic a legitimate organization based on social engineering techniques. Kim *et al.* [22] reported that 27 certificates are issued to attackers who impersonate legitimate companies. Moreover, the report of 2018 showed that 66% downloadable malware are signed [23]. The same report also claimed that some CAs cannot properly validate certificate requests. For example, more than ten percent binary codes signed with the certificates issued by Comodo and Certum were malicious. A certificate from a Russian financial broker is used for the Razy ransomware.

The fake-signed malware could cause serious damages. It is found that antivirus software may not detect a malware with an Authenticode signature extracted from a legitimate file, even though the signature does not match the file digest [22]. Moreover, the protection mechanism can be bypassed to install a digital certificate as the local root certificate so that any malicious programs can be validated [3], [20]. A publisher of antivirus software revealed that a large number of malware are signed by trusted authorities to pass software validation mechanisms of operating systems and browsers [23]. Currently, revoking the abusive certificates is the primary mechanism for mitigating these threats, but CAs may take a long time period to discover and revoke the compromised certificates [24]. As a result, the malware with compromised certificates may remain a threat for a long time without being detected.

B. BINARY CODE SIMILARITY

The algorithms of binary code similarity compare binary-code segments to identify their similarities and differences [25]. The applications of detecting binary code similarity include detecting reused code segments [26]–[29], malware [30], and vulnerable/buggy binary-code segments [27], [31]–[36].

The algorithms of binary code similarity can be categorized into three types according to the number of queried and target binary-code segments: one-to-one (OO), one-to-many (OM), and many-to-many (MM). The algorithms of the first two categories compare one queried binary-code segment with one or multiple target code segments. The algorithms of the last category usually perform clustering upon all input binary-code segments.

Among these categories, the OM algorithms may extract binary-code segments from multiple versions of the same program as the targets. However, the comparisons among binary-code segments are usually performed in one-to-one basis to yield the top k similar binary-code segments [25]. To the best of our knowledge, no algorithm has been created that yields a set of features shared by different versions of a specific program for binary code similarity.

C. SOFTWARE BIRTHMARKS

Software birthmarks are developed for characterizing a software [37]. They can be used for detecting binary-code similarity [38]. The idea of software birthmarks is similar to a computer-virus signature, which uses small pieces of data to identify whether a program is infected or not. A recent survey article identifies 24 types of techniques for generating software birthmarks [39]. These techniques mainly focus on detecting software theft and piracy [16], [40]. There are also birthmarks designed for software asset management [41] or malware detection [42]. Nazir *et al.* [43]–[45] presented fuzzy-based classification for software birthmark estimation.

The existing techniques of software birthmarks can be either static or dynamic. A static birthmark is extracted from the information stored in an executable file. Choi *et al.* [46] developed a statistical method by calculating the distribution of IAT and imported DLL names as a birthmark for program identification. The k-gram birthmark [47] divides n instructions into sequences of length k and uses them as the birthmark [48]. Kim *et al.* [49] extracted strings from an executable file as the birthmark, where these strings could be exception handling messages inserted in the compilation process or texts from the programmers. They used the Jaccard coefficient and the n-gram method to compare similarities between extracted strings.

A dynamic birthmark is obtained from runtime of a program [15], [37]. A dynamic birthmark proposed by Myles uses whole program path (WPP) [15]. The birthmark represents execution paths of a program in a graph. The similarities of programs can thus be measured by comparing graphs. This birthmark may convert its graph into another form for efficient comparison. Another dynamic birthmark is based on the API call frequency and sequence of a program [37]. Tamada *et al.* [50] extended Grover's concept and presented a dynamic birthmark method based on the API call sequence. However, extracting birthmarks for different window sizes requires running the program multiple times [51].

For both static and dynamic software birthmarks, effective feature selection is crucial to the accuracy of software recognition. We are thus motivated to employ machine learning algorithms to generate software birthmarks. Multiple machine learning algorithms are used to show the applicability of our scheme. We also note that the previous approaches of software birthmarks focus on identifying the origin of one single program. In other words, these approaches operate in a one-to-one manner for identifying the source of a program. In contrast, our scheme attempts to generate one single birthmark by using multiple-version executable files of a program. To the best of our knowledge, there is no cross-version software birthmark for identifying authenticity of a program.

III. IDEA AND FRAMEWORK

We propose an alternative approach to identify software credibility by using software birthmarks based on machine learning. Specifically, a machine learning algorithm analyzes features extracted from executable files to generate a binary-classification model as the birthmark. We attempt to use the software birthmark based on different-version executable files to check whether a file is an executable file of the same program but in a different version. The idea is based on an inference that a software publisher usually has a standard procedure for building their software deliverables. We also assume that all related third-party libraries are used across multiple versions since an executable file is usually developed incrementally based on its previous versions. It is thus possible to generate a VWSB by using the features of different-version executable files. Since a software publisher always has the complete information of its programs, the VWSB provides another advantage in which the extracted features from different-version programs are themselves the secret key of the scheme. For example, the programs of non-released versions could provide hidden feature sets which cannot be easily retrieved.

The executable file format includes a section of import and export address table for module redirection. Executable files usually have an IAT in order to retrieve operating-system services such as file access or memory allocation. The IAT may also specify functions imported from the companion library files. The executable files such as DLLs export functions through EATs for module reuse. To support dynamic address mapping, both IAT and EAT map the code bodies of functions into a lookup table with their addresses and function names. Although the code addresses of these functions may change for newly compiled executable files, their function names usually remain unchanged throughout different versions [46]. Therefore, we collect historical IATs and EATs from cross-version executable files to generate software birthmarks because of their consistency and commonality in executable files. We consider each function name in IAT and EAT as a feature. The function names of IAT or EAT in the same file are not allowed to be repeated. All unique function names are extracted and processed to generate global features. Then, we apply machine learning algorithms to analyze the cross-version features in each executable file and generate binary-classification models to serve as the VWSBs.

We use supervised machine learning algorithms to analyze the features of labeled training dataset, where the dataset includes the target executable files and other random executable files. Machine learning algorithms have been used to generate text classifiers for the text categorization problem [52]. We extend the concept by treating the function names in IAT and EAT of each executable file as a text. The text is used to generate a text feature set based on Salton's vector space model [53], where the feature set is a bag of strings extracted from imported or exported function names of an executable file. By parsing the text to extract the words, the strings of function names are inserted into a string collection to serve as the feature strings of an executable file. A feature string (FS) collection gathers feature strings extracted from all target executable files. Each feature string

must appear in at least one target executable file, but it could be absent in some target executable files. Moreover, each FS in the collection can be characterized by its appearance frequency in the entire collection. The frequency is later used to yield weighted features.

We illustrate the framework of generating the proposed software birthmark in Fig. 2. In this figure, the upper-left humanoid mark represents the starter of the build system operation, and the lower-left humanoid mark represents the publisher who receives and delivers the output. The dotted line indicates the system-execution process. In the framework, the generation of a software birthmark is iterative and accumulative for data training and testing. It contains a feedback loop including the procedures from data pre-processing to model validation. The first procedure invokes the standard build system to compile and build the executable file from the source code in order to extract the required features. In our implementation, the feature strings of IAT or EAT are extracted in each iteration of the build procedure.

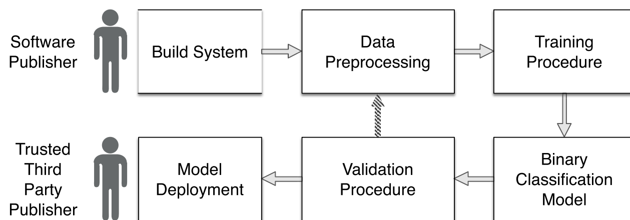


FIGURE 2. The framework of generating VWSB.

The procedure of data pre-processing consists of two steps, data ingestion and data aggregation. Data ingestion extracts FSs from the target executable files and stores them in the set of local FSs. Then, data aggregation merges the local FS sets as a global FS set to yield the total length of all unique strings. Next, the local FS set of each executable file is converted to a local feature vector, where each item is the ratio of the corresponding string length to the total string length. Each item is further assigned a weight value based on the number of its appearance frequency. All local feature vectors are stored in a weighted feature matrix. The idea of generating weights for each feature string is inspired by previous literature of document identification, where the weight and frequency of different terms are calculated [39], [54]. Our scheme also treats the IATs and EATs of executable files as a corpus for executable file identification.

The matrix is then used for training the machine learning algorithms to build software-birthmark models in the training procedure. Once the model is generated, it will be evaluated by testing datasets to measure its effectiveness. If the result is not satisfied, the software-birthmark model is regenerated by updating feature sets until the required accuracy is achieved. Otherwise, the model serves as the software birthmark for executable-file identification.

The procedure of training machine learning algorithms is shown in Fig. 3. In the training phase, not only the executable files of the genuine programs are used, unknown executable files are also included in the training dataset. Accordingly, the feature matrix also includes the class, target or non-target, of each executable file, where a target executable file indicates the genuine program to be detected.

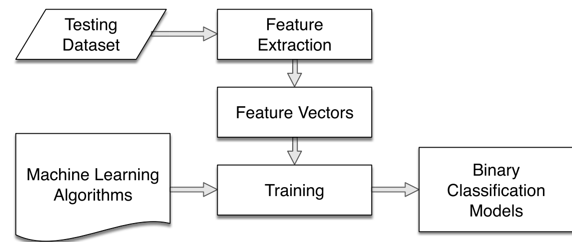


FIGURE 3. Training procedure.

With the software-birthmark model, the validation procedure (Fig. 4) uses a set of additional executable files which are not used in the training procedure to evaluate the performance of a generated model. These files also include both target and non-target executable files. Each tested file is parsed to generate a local feature vector for classification upon the model generated in the training procedure. The performance of a model can thus be evaluated based on the classification accuracy.

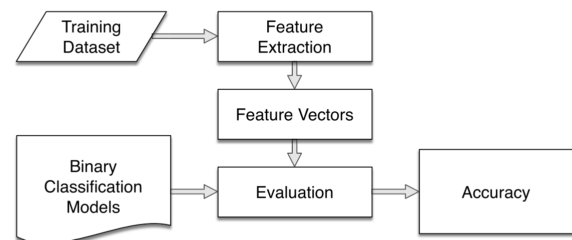


FIGURE 4. Validation procedure.

Note that non-well-managed or non-versioned executable files constructed by random methods are not considered in this work since these files may lose their modularization as well as important features. Our scheme focuses on versioned software with full development because these executable files usually have import and export functions for software modularization to allow programmers collaborate with each other [55]. Moreover, these functions usually provide backward compatibility [46], [56]. These files can thus be characterized by using the function names to generate a birthmark to distinguish them from the other executable files.

We also note that our scheme cannot be applied to stripped binary codes, whose function entry points have been removed. These stripped binary codes could be commercial software for hindering reverse engineering and unlicensed use or malicious programs for resisting analysis [57], [58], where function names are removed from the stripped binary codes. Different types of features must be extracted for birthmark generation, e.g., runtime values from the outcome of machine instructions [59] or runtime call stacks. We also consider the possibility of generating static birthmarks from

unstripped binary codes, which can be applied to stripped binary codes in the further work. While our scheme cannot support stripped binary codes, it still supports various executable files of software with open interfaces for external third-party developers, e.g., base libraries of operating systems, browsers, and document editors.

IV. IMPLEMENTATION OF VWSB

In this section, we describe the implementation of generating the proposed software birthmark. As mentioned above, we use multiple machine learning algorithms to generate VWSBs based on the features in IAT and EAT of executable files. The proposed feature extraction for machine learning is based on global-ranking local feature selection [60]. Our implementation selects FSs based on ALOFT (At Least One FeaTure) and employs FSs as well as their weights for machine learning. The implementation ensures that each target executable file in the training dataset is represented by at least one feature string and contributes to the model generation.

In the following subsections, we describe the procedures for extracting feature strings from IAT and EAT as well as transforming these feature strings into local feature vectors for machine learning algorithms. Finally, VWSB is compared with code signing.

A. FEATURE EXTRACTION FOR IAT AND EAT

An IAT consists of function pointers used to get the addresses of functions when the executable file is loaded. These functions include the wrapper functions of system calls from operating systems. An executable file without an IAT cannot interact with the operating system to access resources, such as computer memory, file storage, and network connection. An EAT is an array of function pointers that contain the names and addresses of exported functions provided by an executable file. Both IAT and EAT provide important features for characterizing an executable file. Our scheme processes the features of IAT and EAT separately because the FSs in IAT and EAT have very different lengths.

Our approach of extracting features is based on the bag-of-words (BoW) model [61]. This model does not consider the order information among FSs but focuses on the occurrence of different FSs in each PE file. On the basis of the BoW model, two types of feature sets, global and local feature sets, are generated from the IATs and EATs of target executable files. The FSs from an IAT include the concatenation of each function name and its original file name, where the FSs from EAT offer only the function names. The FSs of IAT include the file name because the same function name may appear in different files. As a result, the lengths of IAT FSs and those of EAT FSs are quite different. To compensate for the length discrepancy, our scheme builds birthmarks for FSs of IAT and EAT, respectively. The performance differences between IAT and EAT birthmarks are presented in the next section.

The feature strings of an executable file are stored in a corresponding local set of feature strings (LSFS). The global set of feature strings (GSFS) is the union of all local feature sets, which contains all unique feature strings from the collected files. With the LSFS of each file, we can calculate the characteristics of a file based on GSFS to determine the similarity between a file and all collected files. The characteristics of different files are then examined by a machine learning algorithm to generate a binary-classification model for software birthmark.

B. CALCULATION OF FILE CHARACTERISTICS

The characteristics of an executable file is generated from the corresponding LSFS and GSFS. The feature strings must be converted to numbers because they cannot be directly applied to machine learning algorithms. These numbers are stored in a local feature vector (LFV). The sets of all local feature vectors from all executable files in the training dataset are used to train machine learning algorithms.

To generate the LFV for each executable file, the GSFS is used to generate the total length of all feature strings. Next, the length ratio of each feature string is calculated, where the length ratio (LR) is equal to the ratio of the corresponding string length to the total length. The length ratio of feature string x is calculated by using the following equation:

$$LR_x = \frac{x.length}{\sum_{j \in GSFS} j.length} \quad (1)$$

Since a function name found in more files is more representative than that in less files, we further employ the number of executable files importing/exporting a function name as the weight of the function name. More specifically, a feature string weight (FSW) is assigned to each feature string in the GSFS. The FSW value of feature string x is equal to LR_x times *x.filecount*, where *x.filecount* denotes the number of files specifying feature string x. FSW_x denotes the FSW value of feature string x. With the FSW values of all feature strings, an LFV is created for each executable file. The length of each LFV is equal to the number of unique feature strings. The pseudo code of generating the LFV for each file is listed in Algorithm 1.

The algorithm starts from generating the GSFS by merging the LSFSs of all target executable files in the training dataset, as shown in the first **for** loop. In the second **for** loop, the total length of feature strings in GSFS is calculated. The number of executable files referring to a feature string is also yielded. Then, a vector, FSWV, is generated to store the FSW values of different feature string length for the third **for** loop. Finally, one LFV is generated for each file, where the value of each item is the corresponding FSW value in FSWV. In our implementation, both GSFS and LSFS are stored in hash tables to reduce the cost of accessing their feature strings.

To illustrate further, we use different-version PE files of kernel32.dll. Most Windows core functionality can be found in DLLs from Microsoft. Among these DLL files, kernel32.dll provides the primary dynamic library for Windows base functions, including file access and memory management. All function names in the IATs of all collected

Algorithm 1: LFV Generation

Load training dataset $EFtr$
/* $EFtr$: executable files for training */
$GFS = \emptyset$
for $i \in EFtr$ **do**
  if i *is target* **then**
    Generate $LSFS_i$
    $GSFS = GSFS \bigcup LSFS_i$
    /* Insert FSs of File i into $GSFS$ */
$TL = 0$
for $j \in GSFS$ **do**
  $TL+ = j.length$
  for $i \in EFtr$ **do**
    if $j \in LSFS_i$ **then**
      $j.filecount{+}{+}$
Create $FSWV$
for $j \in GSFS$ **do**
  $FSWV(j.length)+ = j.length \times j.filecount \div TL$
for $i \in EFtr$ **do**
  Generate Local Feature Vector, LFV_i
  for $j \in FS_i$ **do**
    if $j \in GSFS$ **then**
      $LFV_i(j) = FSWV(j.length)$
      /* Assign FS Weight */

kernel32.dll in the training dataset are extracted. After transforming these feature strings into local feature vectors, each FS is converted to one of 70 values, since there are 70 distinct string lengths. Then, we use a feature string, "rtlimagedirectoryentrytodata:ntdll.dll", extracted from IAT of kernel32.dll (v5.1.2600.1106) to explain the calculation of FSW value. The length of the feature string is 38, and the total length of all feature strings in GSFS is 11278. This feature string also appears in another 41 kernel32.dll files of the training dataset. Thus, the FSW value of the feature string is equal to 0.13814506118106 ($= 38 \times 41/11278$). The LSV of kernel32.dll (v5.1.2600.1106) is thus updated by assigning the FSW value to the field corresponding to the FS.

We further generate a local feature matrix (LFM), where the matrix includes the same number of rows as the number of executable files in the training dataset. Each row of LFM is used to store the FSW values of each executable file. The number of columns is equal to the length of LFV plus one addition field, which specifies whether the corresponding file is a target or not. The LFM is the input of machine learning algorithms. The LFM of our scheme has the following characteristics:

- Each target executable file can be identified by at least one feature whose FSW value is nonzero. Thus, the features of all target executable files in the training dataset can contribute to the final feature set.
- The algorithm automatically yields the optimal number of features in a data-driven way without pre-calculating the best number of features.
- The algorithm finds the least number of features covering all target executable files in the training dataset.
- The feature extraction based on ALOFT does not require parameter optimization or preliminary tests for best performance.
- The calculation of FSW is fast and deterministic.

C. CLASSIFICATION BASED ON VWSB

In our scheme, the generated software birthmark is created by machine learning algorithms to classify whether an executable file is a target PE file. The LFV of the tested executable file is generated and input into the binary-classification model of the generated software birthmark for classification. As compared to the previous software-birthmark algorithms operating in a one-to-one manner, our algorithm performs in a one-to-many manner to simplify the computation of executable file classification and improve the feasibility of software birthmarks.

D. COMPARISONS WITH CODE SIGNING

Both the proposed scheme of VWSB and code signing attempt to identify the credibility of executable files, but they have very different properties. We show the differences between VWSB and code signing by listing their properties in Table 1. As mentioned above, code signing provides PKI-based authentication and digest-based integrity. Software publishers pay for the certificate issuance. Each signed executable file is then embedded with a signature. While code signing can be applied to virtually all executable files, VWSB only supports versioned executable files with IAT or EAT. The authentication provided by VWSB is based on the prediction performed by the binary classification model of machine learning. VWSB can only detect the modification of features for training the classification model. It does not append any information to executable files.

TABLE 1. Comparisons with code signing and VWSB.

Property	Code Signing	VWSB
Target	Any Executable Files	Versioned Executable Files with IAT/EAT
Authentication	PKI	Prediction
Integrity	Digest	IAT/EAT (features)
Cost	Certificate Issuance	None
File Overhead	Signature	None
Updatability	Renewal with the signed executable file	No modification to the executable file
Operational Overhead	Certificate Distribution and Revocation	Feature Analysis and Model Distribution

Each signature is tightly coupled with the signed executable file, thus the signature can only be updated along with the executable file. Each certificate authority (CA) maintains a repository of issued certificates and distributes these certificates as requested. It also maintains a list of revoked certificates. Currently, the primary mechanism for mitigating the threats of code signing is to revoke the abusive certificates [7]. When abusive certificates are discovered, CAs should revoke

these certificates and disseminate the revocation information. Both software installation and execution should also check the certificate revocation list.

As compared to coding signing, VWSB can be updated without renewing executable files. Since VWSBs are generated by software publishers for remote identification, the update procedure is also simplified. The operational overhead includes extracting features for training machine learning algorithms and distributing the binary classification model to security-related partners. When malware can imitate the features used by an existing VWSB, it is possible to generate a new VWSB by using different features.

In summary, VWSB cannot ensure whole-file integrity, and the authentication performance of VWSB relies on the extracted features and employed machine learning algorithms. Therefore, VWSB cannot take the place of code signing. However, VWSB is not embedded with the executable files to provide the resilience of renewing features and employing different machine learning algorithms. While CAs take a long time period for abuse certificate revocation, VWSB could identify the credibility of executable files with suspicious certificates without relying on PKI.

V. EXPERIMENTS

In order to evaluate the proposed scheme of VWSB, we carried out experiments for different executable files and different machine learning algorithms. We used the Python language to implement our scheme. Our implementation includes twelve notable machine learning algorithms selected from scikit-learn [62], as listed in Table 2. We did not optimize any parameters for these algorithms to show the baseline performance of our scheme, even though they may have different performance upon the features.

TABLE 2. Selected machine learning algorithms.

Abbreviation	Algorithm
DT	Decision Tree Classifier
ET	Extra-trees Classifier
GB	Gradient Boosting
GNB	Gaussian Naive Bayes (GaussianNB)
KNN	Classifier Implementing the k-nearest Neighbors
AdaBoost	Adaptive Boosting Classifier
LR	Logistic Regression (aka logit, MaxEnt)
MLP	Multi-layer Perceptron
RF	Random Forest
SVM	Support Vector Machine
SVM LinearSVC	Linear Support Vector Classification
XGBClassifier	Extreme Gradient Boosting Classifier

The executable files used in our experiments include various versions of kernel32.dll and firefox.exe. They come from two notable software, Microsoft Windows operating system and Mozilla Firefox. The file, kernel32.dll, was selected because of its importance that most programs of the Windows operating systems rely on its functions. We collected globally published 138 kernel32.dll files since 1993. These files cover five major versions, v3.x, v4.x, v5.x, v6.x, and v10.x. Firefox was created in 2002 and released in 2004. It was selected because of its popularity and open-source codes. For Firefox, 545 files of firefox.exe across 31 major versions (from 0.8.x, to 77.0b9.x) since 2004 were collected. It is important to note that the files of version 3.0.x to 48.0.2.x (597 files of about 50 major versions) did not have EATs and were excluded from our experiments. The collected executable files can demonstrate the performance of our scheme even though their IATs and EATs may have been changed constantly because the collected executable files span up to more than two decades. We also prepared another random 10900 executable files from "Windows" and "Program Files" folders for training and testing. None of these executable files are stripped binary codes. To effectively validate the classification performance, these non-target PE files were retrieved from the applications of the same or similar environments as the target executable files.

The performance evaluation of VWSB consists of two parts. In the first part, we used 80% of the collected files for training and the other files for testing to show the accuracy of VWSB. In the second part, we divided the collected target PE files into different groups to generate training datasets of different sizes, where each dataset has different number of target and random PE files. We used these training datasets to show the importance of cross-version features.

A. STATISTICS OF EXTRACTED FEATURES

We show the statistics of different-major-version files in Table 3 and 4, where the major version is defined based on the leftmost number in the version number. For both kernel32.dll and firefox.exe, we categorize the collected files into five groups according to their versions, and there is another group for all files. The feature strings of IAT and EAT in an executable file are usually similar to the other executable files of the same major version because software publishers usually avoid significant upgrades to maintain consistent functionality and quality.

Table 3 lists the total numbers of feature strings from IAT and EAT of the kernel32.dll files in the same group. The kernel32.dll files were extracted from Windows operating systems of different versions, thus the names of these Windows operating systems were also revealed. It is shown that

TABLE 3. Statistics of selected kernel32.dll versions.

Versions	Files	IAT FSs	EAT FSs
3.10.x∼3.50.x (Windows NT 3.1∼3.5)	5	1190	2830
4.0.x∼4.10.x (Windows NT 4.0)	5	1350	3375
5.0.x∼5.5.x (Windows 2000, Windows XP, Windows Server 2003, Windows Neptune)	27	10395	25191
6.0.x∼6.3.x (Windows Vista, Windows 7, Windows 8, Windows Server 2012 R2)	45	39105	63135
10.0.10x∼10.0.20x (Windows 10)	56	68600	90160
3.10.x∼10.0.20x (All collected versions)	138	120612	184644

kernel32wfs_import.data

filename	internalname	companyname	total	ntcanceliofile:ntdll.dll	lockresource:api-ms-win-core-libraryloader-l1-2-0.dll	rtlinitansistringex:ntdll.dll	ntqueueapcthread:ntdll.dll
3.10.404.kernel32.dll.selected	kernel32	microsoft corporation	6726	0	0	0	0
3.10.438.kernel32.dll.selected	kernel32	microsoft corporation	6772	0	0	0	0
3.50.756.kernel32.dll.selected	kernel32	microsoft corporation	7084	0	0	0	0
3.50.854.1.kernel32.dll.selected	kernel32	microsoft corporation	7064	0	0	0	0
4.00.1381.1.kernel32.dll.selected	kernel32	microsoft corporation	7451	0.103073413	0	0	0.090726077
5.00.2191.1.de.kernel32.dll.selected	kernel32	microsoft corporation	9304	0.082545142	0	0	0.072656922
5.00.2191.1.kernel32.dll.selected	kernel32	microsoft corporation	9304	0.082545142	0	0	0.072656922
5.1.2600.0.kernel32.dll.selected	kernel32	microsoft corporation	11241	0.068321324	0	0	0.060136998
5.1.2600.1106.kernel32.dll.selected	kernel32	microsoft corporation	11278	0.06809718	0	0	0.059939706
5.1.2600.1106-1.kernel32.dll.selected	kernel32	microsoft corporation	11278	0.06809718	0	0	0.059939706
5.1.2600.1106-polish.kernel32.dll.selected	kernel32	microsoft corporation	11278	0.06809718	0	0	0.059939706
5.1.2600.2180 (xpsp_sp2_rtm.040803-2158).de.32.kernel32.dll.selected	kernel32	microsoft corporation	11393	0.067409813	0	0	0.059334679

FIGURE 5. A part of the local feature matrix (kernel32.dll, IAT).

the number of feature strings increases as the version number raises. The collected kernel32.dll files have 2402 and 1808 distinct FSs in IAT and EAT, respectively.

The total numbers of feature strings for each group of firefox.exe files are listed in Table 4. Using regular rapid-release cycles, there has been one major-version release of firefox.exe every six weeks since June 2011. The collected files cover 31 different major versions; thus, each file group for firefox.exe includes executable files from approximately six major versions. The first group of firefox.exe includes files before April 2009 and after September 2016, because the files of versions from 3.0.x to 48.0.2.x were excluded from our experiments. Table 4 shows that the files of the first group have many more FSs in both IAT and EAT than the files of the other groups. For the files of the other groups, both the FS numbers of IAT and EAT gradually increase as the version number increases. The results suggest that software publishers may change the layouts of IAT and EAT of their executable files in some major-version upgrades. The collected firefox.exe files have 3953 and 823 distinct FSs in IAT and EAT, respectively.

TABLE 4. Statistics of selected firefox.exe versions for windows.

Versions	Files	IAT FSs	EAT FSs
0.8.x∼52.9.x (excluding 3.0.x∼48.0.2.x)	142	155490	39334
53.0.x∼58.0.2.x	94	23500	4418
59.0.x∼63.0.3.x	105	26145	5460
64.0.x∼68.7.0.x	95	26980	5035
69.0.x∼77.0b9.x	109	36515	6540
0.8.x∼77.0b9.x (All collected versions)	545	268685	61040

After extracting feature strings from the executable files of training datasets, the FSW value of each feature string is calculated. For kernel32.dll files, the total string length varies from 6726 (v 3.10.404) to 54464 (v10.0.10586.0). Then, one local feature matrix was generated for each training dataset. We show a part of the local feature matrix for the training dataset of kernel32.dll in Fig. 5. We observed that some function names of IAT appear in all PE files, ntreadfile:ntdll.dll, ntopenfile:ntdll.dll, ntcreatemailslotfile:ntdll.dll, to name but a few. There are also some function names appearing intermittently, e.g., _local_unwind:ntdll.dll, rtlvirtualunwind:ntdll.dll and _aulldiv:ntdll.dll. As a result, the differences among executable files of different versions are dynamic and difficult to classify without obtaining these executable files.

TABLE 5. Statistics of selected firefox versions for linux.

Versions	Files	IAT FSs	EAT FSs
10.0.x∼19.0.x	10	542	280
20.0.x∼29.0.x	10	700	649
30.0.x∼39.0.x	10	824	836
40.0.x∼49.0.x	10	915	1154
50.0.x∼59.0.x	10	908	1339
60.0.x∼69.0.x	10	996	1621
70.0.x∼79.0.x	10	1349	3305
80.0.x∼88.0.x	9	1144	2679
10.0.x∼88.0.x (All collected versions)	79	7378	11863

We further show the number of IAT and EAT FSs in the executable files of firefox for Linux and macOS. The executable file format of Linux is Executable and Linkable Format (ELF) and that of macOS is Mach Object File Format (Mach-O). We collected the executable files of all major versions since 2012 and categorized them into eight groups. Because of the different build system of macOS, the Mach-O executable file of firefox heavily relies on another dylib file of dynamic shared libraries for most functions. For example, the dylib file for firefox v10.0 is libmozutils.dylib and that for v88.0 is libmozglue.dylib. Accordingly, we exact the IATs and EATs from the companion dylib files. The number of FSs in both IATs and EATs of each group are listed in Table 5 and 6. Both tables show that the executable files of Linux and macOS have a similar count of FSs in both IATs and EATs. Specifically, there are 151 and 279 FSs in the IAT and EAT of libmozglue.dylib for firefox (version 88.0). The same-version ELF file has 128 and 273 FSs for IAT and EAT, respectively. As a comparison, there are 364 IAT and 98 EAT FSs for the same-version PE file. Although differences among different build systems exists, it is still feasible to apply the proposed software birthmarks to the executable files of different operating systems with a sufficient number of FSs in IATs and EATs.

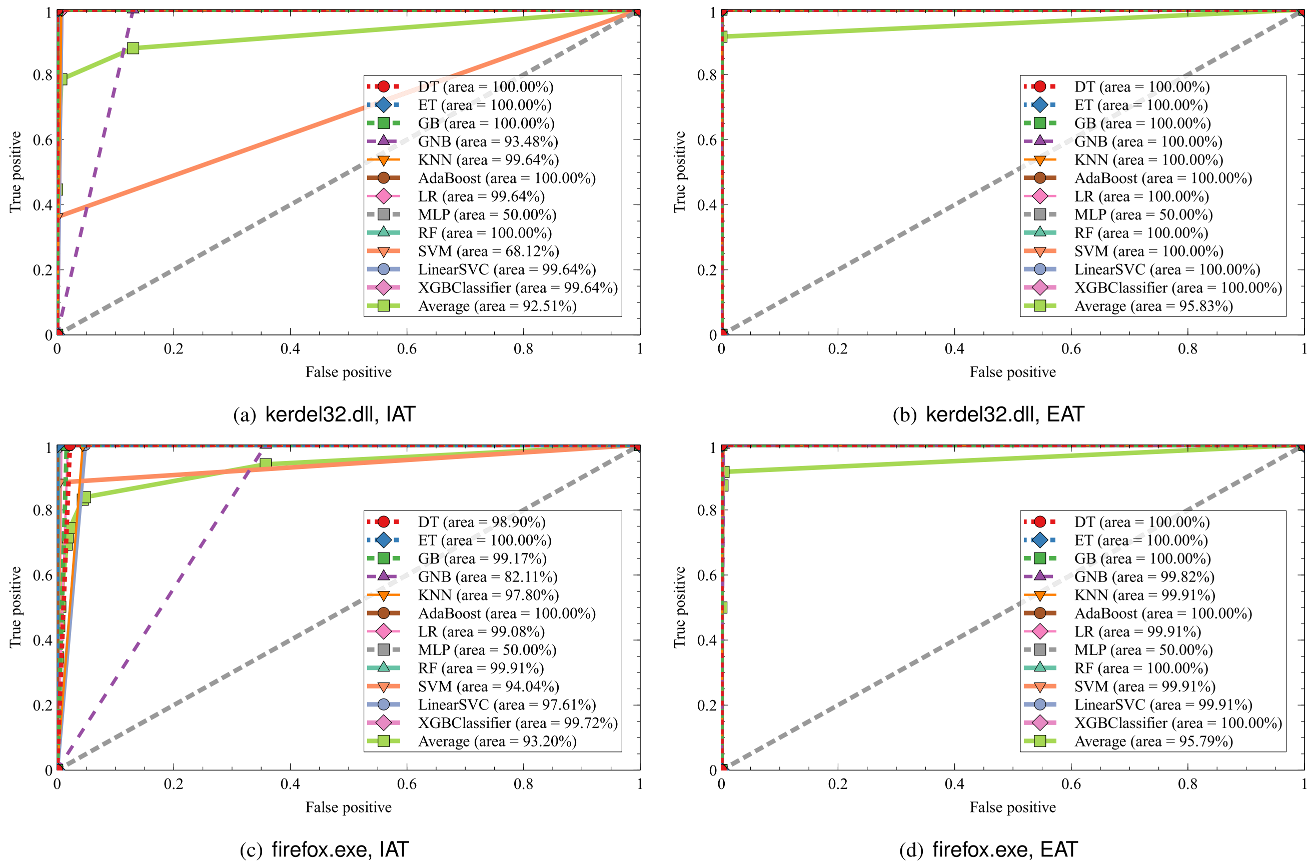


FIGURE 6. ROC performance for IAT and EAT of kernel32.dll and firefox.exe.

TABLE 6. Statistics of selected firefox versions for Mac.

Versions	Files	IAT FSs	EAT FSs
10.0.x∼19.0.x	10	480	255
20.0.x∼29.0.x	10	505	521
30.0.x∼39.0.x	10	532	710
40.0.x∼49.0.x	10	807	1023
50.0.x∼59.0.x	10	951	1328
60.0.x∼69.0.x	10	976	1701
70.0.x∼79.0.x	10	1429	3363
80.0.x∼88.0.x	9	1333	2720
10.0.x∼88.0.x (All collected versions)	79	6874	11621

TABLE 7. Summarized ROC performance in Fig. 6.

Algorithms	kernel32.dll		firefox.exe		Average
	IAT	EAT	IAT	EAT	
DT	100%	100%	98.9%	100%	99.73%
ET	100%	100%	100%	100%	100%
GB	100%	100%	99.17%	100%	99.79%
GNB	93.48%	100%	82.11%	99.82%	93.85%
KNN	99.64%	100%	97.8%	99.91%	99.34%
AdaBoost	100%	100%	100%	100%	100%
LR	99.64%	100%	99.08%	99.91%	99.66%
MLP	50%	50%	50%	50%	50%
RF	100%	100%	99.91%	100%	99.98%
SVM	68.12%	100%	94.04%	99.91%	90.52%
SVM LinearSVC	99.64%	100%	97.61%	99.91%	99.29%
XGBClassifier	99.64%	93.75%	99.72%	100%	98.28%

B. PERFORMANCE OF VWSB

We use the receiver operating characteristic (ROC) curve to show the performance of the generated software birthmark. ROC has been used to visualize the performance of a classifier for selecting a suitable operating point or a decision threshold. The area under a ROC curve (AUC) is used to evaluate the performance of machine learning algorithms [63]. If the ROC curves of different algorithms do not intersect, at least one algorithm would outperform the others. However, if two ROC curves intersect, one algorithm outperforms the other for only some cost ratios.

Our experiments included two sets of target executable files. The training datasets were generated for kernel32.dll and firefox.exe, where each dataset includes 80% target and non-target executable files. For each training dataset, two software birthmarks were generated, one for IAT and the other for EAT.

We show the IAT and EAT results for kernel32.dll in Fig. 6(a) and 6(b), respectively. As shown in Fig. 6(a), five machine learning algorithms achieve 100% classification accuracy. The accuracy of another five algorithms, XGBClassifer, GNB, KNN, LR and LinearSVC, is within the range

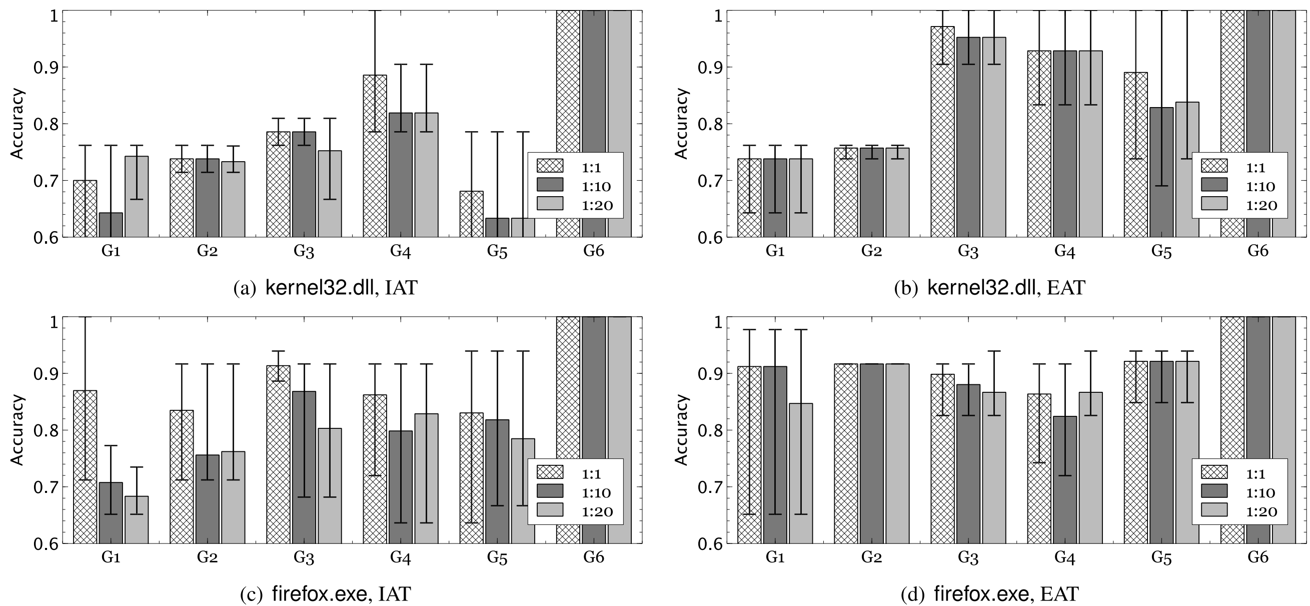


FIGURE 7. VWSB performance for files groups in Table 8.

between 93.48% and 99.64%. The accuracy of SVM is further degraded to 68.12%. However, MLP can only achieve 50% accuracy.

For EATs of kernel32.dll files, the performance of eleven machine learning algorithms achieve 100% classification accuracy, as shown in Fig. 6(b). However, MLP remains 50% accuracy in this experiment.

The IAT and EAT results for firefox.exe are shown in Fig. 6(c) and 6(d), respectively. Fig. 6(c) shows that two machine learning algorithms achieve 100% classification accuracy. The accuracy of another eight algorithms, XGB-Classifer, GB, KNN, LR, SVM, DT, RF and LinearSVC, is within the range between 94.04% and 99.91%. GNB can only achieve 82.11% accuracy, and the accuracy of MLP is further degraded to 50%.

For EATs of firefox.exe files, six machine learning algorithms achieve 100% accuracy, as shown in Fig. 6(d). The accuracy of another five algorithms, GNB, KNN, LR, SVM, LR and LinearSVC, is at least as high as 99.82%. However, MLP remains 50% classification accuracy in this experiment.

Table 7 summarizes the above results. MLP is the only algorithm which cannot achieve perfect classification accuracy in any scenarios. GNB and SVM have relatively low accuracy for IAT of firefox.exe and kernel32.dll, respectively. The other algorithms can achieve high accuracy of at least 97.61%. Two machine learning algorithms, ET and AdaBoost, have perfect accuracy in all scenarios. iIn summary, the proposed software birthmark can achieve high classification accuracy by using most machine learning algorithms. Moreover, the software birthmarks based on EAT have better performance than those based on IAT.

Next, we conduct two experiments to demonstrate the effectiveness of the proposed software birthmarks by varying the PE files for training and testing. In both experiments, we only use the best five machine learning algorithms in the previous experiments as shown in Table 7, namely DT, ET, GB, AdaBoost and RF, to better illustrate the performance difference caused by different training and testing datasets. We also increase the number of non-target files for training, where the ratios of target to non-target files vary from 1:1, 1:10 to 1:20. The first experiment uses different groups of PE files for training and uses randomly selected PE files of different versions for testing. This experiment demonstrates whether a software birthmark generated from specific-version PE files can be used to classify PE files of the other versions. The combinations of target PE files for training and testing are listed in Table 8, where each combination has a group identifier. There are fewer FSs in the corresponding GSFS because the training dataset is reduced. We denote the updated GSFS as GSFS' and show their sizes in Table 8. The feature string (FS) ratio for each combination is defined as the ratio of the size of GSFS' to that of the original GSFS.

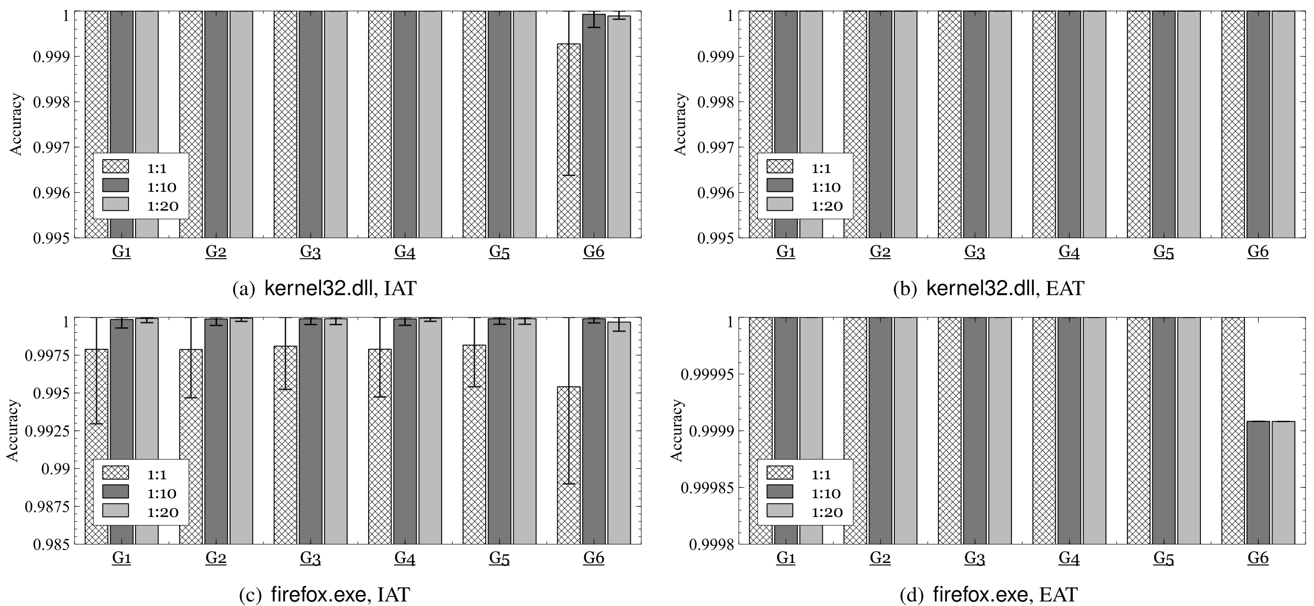
The classification accuracy for each file group is shown in Fig. 7, where Fig. 7(a) and 7(b) show results for kernel32.dll and Fig. 7(c) and 7(d) for firefox.exe. These results show that the accuracy for the first five groups is usually much lower than that for the last group, whose files for training include all target PE files. Fig. 7(a) and 7(b) show that for kernel32.dll, training based on the file groups of newer versions cannot achieve better accuracy than that of the earlier versions, even though these groups also have higher FS ratios. The results also suggest that EAT has stronger correlations among different-version kernel32.dll files than IAT. Fig. 7(c) and 7(d) show that although the accuracy degrades for different groups of firefox.exe files, the decrement is

TABLE 8. Files groups for specific-version training and cross-version testing.

PE File	Group	Versions		IAT			EAT		
		Training	Testing	|GSFS|	|GSFS'| / FS Ratio Training	|GSFS'| / FS Ratio Testing	|GSFS|	|GSFS'| / FS Ratio Training	|GSFS'| / FS Ratio Testing
kernel32.dll	G1	3.1x∼3.5x (5 files)	3.10.x∼10.0.2x (21 files)	2402	238/0.0991	637/0.2652	1808	566/0.3131	1119/0.6189
	G2	4.00.1x∼4.00.1x (5 files)			270/0.1124			675/0.3733	
	G3	5.00.2x∼5.50.5x (27 files)			385/0.1603			933/0.5160	
	G4	6.0.6x∼6.3.9x (45 files)			869/0.3618			1403/0.7760	
	G5	10.0.1x∼10.0.2x (56 files)			1225/0.5100			1610/0.8905	
	G6	3.10.x∼10.0.2x (138 files)			847/0.3526			1338/0.7400	
firefox.exe	G1	0.8.x∼52.9.x (142 files)	0.8.x∼77.0b9.x (66 files)	3953	1095/0.2770	507/0.1283	823	277/0.3366	116/0.1410
	G2	53.0.x∼58.0.2.x (94 files)			250/0.0632			47/0.0571	
	G3	59.0.x∼63.0.3.x (105 files)			249/0.0630			52/0.0632	
	G4	64.0.x∼68.7.0.x (95 files)			284/0.0718			53/0.0644	
	G5	69.0.x∼77.0b9.x (109 files)			335/0.0847			60/0.0729	
	G6	0.8.x∼77.0b9.x (545 files)			493/0.1247			112/0.1361	

TABLE 9. Files groups for cross-version training and specific-version testing.

PE File	Group	Versions		IAT			EAT		
		Training	Testing	|GSFS|	|GSFS'| / FS Ratio Training	|GSFS'| / FS Ratio Testing	|GSFS|	|GSFS'| / FS Ratio Training	|GSFS'| / FS Ratio Testing
kernel32.dll	G1	3.10.x∼10.0.2x (21 files)	3.1x∼3.5x (5 files)	2402	637/0.2652	238/0.0991	1808	1119/0.6189	566/0.3131
	G2		4.00.1x∼4.00.1x (5 files)			270/0.1124			675/0.3733
	G3		5.00.2x∼5.50.5x (27 files)			385/0.1603			933/0.5160
	G4		6.0.6x∼6.3.9x (45 files)			869/0.3618			1403/0.7760
	G5		10.0.1x∼10.0.2x (56 files)			1225/0.5100			1610/0.8905
	G6		3.10.x∼10.0.2x (138 files)			847/0.3526			1338/0.7400
firefox.exe	G1	0.8.x∼77.0b9.x (66 files)	0.8.x∼52.9.x (142 files)	3953	507/0.1283	1095/0.2770	823	116/0.1409	277/0.3366
	G2		53.0.x∼58.0.2.x (94 files)			250/0.0632			47/0.0571
	G3		59.0.x∼63.0.3.x (105 files)			249/0.0630			52/0.0632
	G4		64.0.x∼68.7.0.x (95 files)			284/0.0718			53/0.0644
	G5		69.0.x∼77.0b9.x (109 files)			335/0.0847			60/0.0729
	G6		0.8.x∼77.0b9.x (545 files)			493/0.1247			112/0.1361



(a) kernel32.dll, IAT (b) kernel32.dll, EAT (c) firefox.exe, IAT (d) firefox.exe, EAT

FIGURE 8. VWSB performance for files groups in Table 9.

less obvious as compared to the results of kernel32.dll. The results of firefox.exe also show that the firefox.exe files have higher similarity in their IATs and EATs than kernel32.dll. In addition, the EATs of firefox.exe provide better accuracy than IATs. For both kernel32.dll and firefox.exe, the ratios of non-target files in the training dataset may lead

TABLE 10. VWSB accuracy results in Fig. 8.

	kernel32.dll, IAT									kernel32.dll, EAT								
	1:1			1:10			1:20			1:1			1:10			1:20		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
G1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G3	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G4	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G5	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G6	0.9993	0.9964	1.0000	0.9999	0.9996	1.0000	0.9999	0.9998	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	firefox.exe, IAT									firefox.exe, EAT								
	1:1			1:10			1:20			1:1			1:10			1:20		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
G1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G3	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G4	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G5	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
G6	0.9993	0.9964	1.0000	0.9999	0.9996	1.0000	0.9999	0.9998	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

to oscillatory accuracy. The oscillation is caused by the similarity between limited FSs of specific-version target files and non-target files. By including all target PE files of different versions, the oscillation can be eliminated to achieve perfect accuracy.

The second experiment uses another set of file groups shown in Table 9 for training and testing. In this experiment, we include two target PE files of each major version in the training dataset and show the accuracy for PE files of specific versions in Fig. 8. The selected files of different major versions can provide sufficient FSs to identify PE files to achieve superior accuracy, even though the FS ratio is only 0.1283 for firefox.exe. Moreover, different numbers of non-target files in a training dataset only result in subtle influence. We conclude that the features extracted from target PE files of different versions are crucial for the accuracy of the proposed software birthmarks. The detailed results are listed in Table 10.

VI. CONCLUSION

In this work, we propose a scheme, version-wide software birthmark, to detect software credibility without relying on PKI. Unlike the previous algorithms of software birthmarks designed for detecting software theft and piracy, our scheme generates one birthmark based on the different-version executable files of a program by using machine learning algorithms. The birthmark is a binary-classification model classifying whether an executable file is a different-version executable file of the same program. We implement this scheme for Windows portable executable file format. Our implementation extracts function names from import and export address tables and transforms these function names into a matrix of features, where the transformation is based on string lengths and occurrences. The matrix is then used for training selected machine learning algorithms. In our experiments, two notable programs, kernel32.dll and firefox.exe, are used to evaluate the performance of our scheme. Although these executable files are quite different with respect to the number and appearance of feature strings, the results show that most machine learning algorithms have at least 99% accuracy for EAT of the tested executable files. Two algorithms, ET and AdaBoost, can achieve 100% accuracy of classification for both IAT and EAT. We also demonstrate that a training dataset including different-version target files has a positive impact on the classification accuracy.

The version-wide software birthmark is not embedded in the PE files, thus software identification based on birthmarks can be performed remotely. The software birthmarks can be hidden or updated by software publishers. It is thus difficult for malicious programmers to create a forgery executable file without collecting different-version executable files, even when they employ the same feature extraction method. In summary, we believe that the proposed version-wide software birthmark could be an effective approach for validating the credibility of a program, especially the executable files with suspicious certificates. In the future work, we attempt to extend VWSB by extracting features for stripped binary codes.

REFERENCES

[1] Microsoft. (2017). *Introduction to Code Signing*. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms537361(v=vs.85).aspx

[2] S. Khandelwal. (2017). *The Rise of Super-Stealthy Digitally Signed Malware-Thanks to the Dark Web*. [Online]. Available: https://thehackernews.com/2017/11/malware-digital-certificate.html

[3] D. Papp, B. Kócsó, T. Holczer, L. Buttyán, and B. Bencsáth, "ROSCO: Repository of signed code," in *Proc. Virus Bull. Conf.*, 2015, pp. 1–7.

[4] E. Lawrence. (2011). *Everything You Need to Know About Authenticode Code Signing*. [Online]. Available: https://blogs.msdn.microsoft.com/ieinternals/2011/03/22/everything-you-need-to-know-about-authenticode-code-signing/

[5] National Security Agency. (2019). *Enforce Signed Software Execution Policies*. [Online]. Available: https://media.defense.gov/2019/Sep/09/2002180334/-1/-1/0/Enforce%20Signed%20Software%20Execution%20Policies%20-%20Copy.pdf

[6] McAfee. (Apr. 2021). *McAfee Labs Threats Report*. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-apr-2021.pdf

[7] D. Kim, B. J. Kwon, K. Kozák, C. Gates, and T. Dumitraș, "The broken shield: Measuring revocation effectiveness in the windows code-signing PKI," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 851–868.

[8] P. R. Kasselman, "A fast attack on the MD4 hash function," in *Proc. South Afr. Symp. Commun. Signal Process. (COMSIG)*, 1997, pp. 147–150.

[9] T. Xie and D. Feng, "How to find weak input differences for MD5 collision attacks," in *Proc. IACR Cryptol. ePrint Arch.*, 2009, p. 223.
[10] F. Chabaud and A. Joux, "Differential collisions in SHA-0," in *Proc. Annu. Int. Cryptol. Conf.*, 1998, pp. 56–71.
[11] W. Xiaoyun, "Finding collisions in the full SHA-1," in *Proc. CRYPTO*, 2005, pp. 17–36.
[12] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse. (2017). *Announcing the First SHA1 Collision*. [Online]. Available: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
[13] E. Bursztein. *How we Create the First SHA-1 Collision and What it Means for Hash Security*. Accessed: Jul. 26, 2017. [Online]. Available: https://www.blackhat.com/us-17/briefings/schedule/index.html#how-we-created-the-first-sha-1-collision-and-what-it-means-for-hash-security-7693
[14] T. Nipravsky. (2016). *Certificate Bypass: Hiding and Executing Malware From a Digitally Signed Executable*. Blackhat USA. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Nipravsky-Certificate-Bypass-Hiding-And-Executing-Malware-From-A-Digitally-Signed-Executable-wp.pdf
[15] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Proc. Int. Conf. Inf. Secur.*, 2004, pp. 404–415.
[16] S. Nazir, S. Shahzad, and N. Mukhtar, "Software birthmark design and estimation: A systematic literature review," *Arabian J. Sci. Eng.*, vol. 44, no. 4, pp. 3905–3927, 2019.
[17] A. Tucker, R. Morelli, and C. De Silva, *Software Development: An Open Source Approach*. Boca Raton, FL, USA: CRC Press, 2011.
[18] U. Haritha and V. L. Naidu, "A survey on windows component loading vulnerabilities," *Int. J. Adv. Res. Comput. Eng. Technol.*, vol. 2, no. 5, pp. 1780–1783, 2013.
[19] Microsoft. (2008). *Windows Authenticode Portable Executable Signature Format (Version 1.0. March 21, 2008)*. [Online]. Available: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx
[20] A. Alsaid and C. J. Mitchell, "Installing fake root keys in a PC," in *Proc. Eur. Public Key Infrastruct. Workshop*, 2005, pp. 227–239.
[21] J. Henno, H. Jaakkola, and J. Mäkelä, "Using multiplayer games to create secure communication," in *Proc. 8th Workshop Softw. Qual. Anal., Monit., Improvement, Appl.*, 2019, pp. 1–13.
[22] D. Kim, B. J. Kwon, and T. Dumitraş, "Certified malware: Measuring breaches of trust in the windows code-signing PKI," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1435–1448.
[23] TrendMicro. (2018). *Understanding Code Signing Abuse in Malware Campaigns*. [Online]. Available: https://www.trendmicro.com/en_us/research/18/d/understanding-code-signing-abuse-in-malware-campaigns.html
[24] D. Kim, "Understanding of adversary behavior and security threats in public key infrastructures," Ph.D. dissertation, Dept. Comput. Sci., Univ. Maryland, College Park, MD, USA, 2020.
[25] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 1–38, Jun. 2021.
[26] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Kam1n0: MapReduce-based assembly clone search for reverse engineering," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 461–470.
[27] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 88–98.
[28] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 155–166.
[29] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[30] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "FOSSIL: A resilient and efficient system for identifying FOSS functions in malware binaries," *ACM Trans. Privacy Secur.*, vol. 21, no. 2, pp. 1–34, Feb. 2018.
[31] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2018, pp. 114–138.
[32] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, Nov. 2018.
[33] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2018, pp. 896–899.
[34] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*. [Online]. Available: http://arxiv.org/abs/1808.04706
[35] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 309–329.
[36] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 376–387.
[37] D. Grover, *The Protection of Computer Software: Its Technology and Applications*. Cambridge, U.K.: Cambridge Univ. Press, 1992.
[38] S. Cesare and Y. Xiang, *Software Similarity and Classification*. London, U.K.: Springer-Verlag, 2012.
[39] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, 1988.
[40] K. Guan, S. Nazir, X. Kong, and S. U. Rehman, "Software birthmark usability for source code transformation using machine learning algorithms," *Sci. Program.*, vol. 2021, pp. 1–7, Feb. 2021.
[41] D. Kim, J. Moon, S.-J. Cho, J. Choi, M. Park, S. Han, and L. Chung, "A birthmark-based method for intellectual software asset management," in *Proc. 8th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2014, pp. 1–6.
[42] S. Vemparala, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamo, "Malware detection using dynamic birthmarks," in *Proc. ACM Int. Workshop Secur. Privacy analytics*, 2016, pp. 41–46.
[43] S. Nazir, S. Shahzad, S. A. Khan, N. B. Alias, and S. Anwar, "A novel rules based approach for estimating software birthmark," *Sci. World J.*, vol. 2015, Apr. 2015, Art. no. 579390.
[44] S. Nazir, S. Shahzad, I. Zada, and H. Khan, "Evaluation of software birthmarks using fuzzy analytic hierarchy process," in *Proc. 4th Int. Multi-Topic Conf.*, 2015, pp. 171–175.
[45] S. Nazir, S. Shahzad, R. B. Atan, and H. Farman, "Estimation of software features based birthmark," *Cluster Comput.*, vol. 21, no. 1, pp. 333–346, Mar. 2018.
[46] J. Choi, Y. Han, S.-J. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung, "A static birthmark for ms windows applications using import address table," in *Proc. 7th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput.*, 2013, pp. 129–134.
[47] X. Xie, F. Liu, B. Lu, and L. Chen, "A software birthmark based on weighted K-gram," in *Proc. IEEE Int. Conf. Intell. Comput. Intell. Syst.*, vol. 1, Oct. 2010, pp. 400–405.
[48] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2005, pp. 314–318.
[49] Y. Kim, J. Moon, D. Kim, Y. Jeong, S.-J. Cho, M. Park, and S. Han, "A static birthmark of windows binary executables based on strings," in *Proc. 7th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput.*, 2013, pp. 734–738.
[50] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-I. Matsumoto, "Design and evaluation of dynamic software birthmarks based on API calls," Info. Sci., Nara Inst. Sci. Technol., Kansai Science City, Japan, Tech. Rep. NAIST-IS-TR2007011, pp. 0919–9527, 2007.
[51] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2007, pp. 274–283.
[52] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *Proc. Eur. Conf. Mach. Learn.*, 1998, pp. 137–142.
[53] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.
[54] L. Bernauer, E. J. Han, and S. Y. Sohn, "Term discrimination for text search tasks derived from negative binomial distribution," *Inf. Process. Manage.*, vol. 54, no. 3, pp. 370–379, May 2018.

[55] D. Davis and B. J. B. Mark, "Untangling parametric schemata: Enhancing collaboration through modular programming," in *Proc. 14th Int. Conf. Comput. Aided Architectural Design*, 2011, pp. 55–68.
[56] Wikipedia. (2021). *Software Versioning*. [Online]. Available: https://en.wikipedia.org/wiki/Software_versioning
[57] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005.
[58] X. Meng and B. P. Miller, "Binary code is not easy," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 24–35.
[59] Y. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 925–943, Apr. 2015.
[60] R. H. W. Pinheiro, G. D. C. Cavalcanti, R. F. Correa, and T. I. Ren, "A global-ranking local feature selection method for text categorization," *Expert Syst. Appl.*, vol. 39, no. 17, pp. 12851–12857, 2012.
[61] Z. S. Harris, "Distributional structure," *Word*, vol. 10, nos. 2–3, pp. 146–162, 1954.
[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[63] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, 1997.

CHIH-KO CHUNG received the bachelor's degree from the Department of Management Information System, National Pingtung University of Science and Technology, in 1998. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, National Chung Hsing University. He has been serving for Trend Micro Cybersecurity Solutions department, as a Senior Architect, since 2000. His research interests include cyber security and machine learning.

PI-CHUNG WANG (Member, IEEE) received the M.S. and Ph.D. degrees in computer science and information engineering from National Chiao Tung University, Taiwan, in 1997 and 2001, respectively. From 2002 to 2006, he was with the Telecommunication Laboratories, Chunghwa Telecom. He joined the Department of Computer Science and Engineering (CSE), National Chung Hsing University (NCHU), in 2006, where he has been a Professor of CSE, since 2014. His research interests include packet classification and mobile edge computing.