IEICE TRANS. INF. & SYST., VOL.E99–D, NO.7 JULY 2016

PAPER

API-Based Software Birthmarking Method Using Fuzzy Hashing

Donghoon LEE[†a)], Dongwoo KANG[†b)], Younsung CHOI[††c)], Jiye KIM[†d)], *Nonmembers*, *and* Dongho WON[†e)], *Member*

SUMMARY The software birthmarking technique has conventionally been studied in fields such as software piracy, code theft, and copyright infringement. The most recent API-based software birthmarking method (Han et al., 2014) extracts API call sequences in entire code sections of a program. Additionally, it is generated as a birthmark using a cryptographic hash function (MD5). It was reported that different application types can be categorized in a program through pre-filtering based on DLL/API numbers/names. However, similarity cannot be measured owing to the cryptographic hash function, occurrence of false negatives, and it is difficult to functionally categorize applications using only DLL/API numbers/names. In this paper, we propose an API-based software birthmarking method using fuzzy hashing. For the native code of a program, our software birthmarking technique extracts API call sequences in the segmented procedures and then generates them using a fuzzy hash function. Unlike the conventional cryptographic hash function, the fuzzy hash is used for the similarity measurement of data. Our method using a fuzzy hash function achieved a high reduction ratio (about 41% on average) more than an original birthmark that is generated with only the API call sequences. In our experiments, when threshold ε is 0.35, the results show that our method is an effective birthmarking system to measure similarities of the software. Moreover, our correlation analysis with top 50 API call frequencies proves that it is difficult to functionally categorize applications using only DLL/API numbers/names. Compared to prior work, our method significantly improves the properties of resilience and credibility.

key words: *software birthmark, birthmarking systems, software similarity, fuzzy hash, API-based sequences*

1. Introduction

A software birthmark is a technology that reflects the characteristics that are inherent to each software application. Software birthmarking techniques have conventionally been studied for detection of software piracy, code theft, and copyright infringement. As information and communication technologies have developed in recent years, malicious activities, such as piracy, code cracking, and code theft, have evolved and the distribution of illegal software has been accelerating. According to the 4th Annual State of Application Security Report published by ARXAN in 2015, approximately 77.9% of illegally shared media worldwide is software [1]. That software alliance analysis and IDC findings also reported that 45% of illegal software distributions are performed through online websites or P2P networks [2].

Software vendors have studied and applied such technologies such as software watermarking [3], [4], tamper-proofing [5], obfuscating [6], [7], and software birthmarking [8]–[11] to protect their intellectual property. Among these, software watermarking and birthmarking can handle copyright infringement and software code theft. Software watermarking techniques, which involve embedding copyright marks in source code to identify the software owner, have previously been proposed [4], [12]. These watermark technologies, however, are easily damaged by semantics-preserving obfuscation [13] or optimization of compilers.

A birthmark is a colored mark or mole that appears at birth. Similarly, a software birthmark technique represents a technology that reflects the inherent features of each software program. Therefore, such technologies are widely used in many recent applications, including digital forensics, malicious code detection, and detection of software copyright infringement and code theft [14], [15].

Software birthmarks have been conventionally generated from application program code. These can be classified into source code and compiled code. The compiled program code can be divided into native code (or unmanaged code, machine code) or managed code depending on the compiler and programming language used [16].

Managed code is a term coined by Microsoft [17]. If original source code, written in a programming language such as Java or C#, is compiled, the managed code generates an intermediate language (IL), which is interpreted and executed by a virtual machine (VM). ILs, such as Java bytecode, are operated on by a VM and can thus be independently executed on a given platform (i.e., a microprocessor, operating system, and so on).

In contrast, native code is compiled code that relies on a corresponding platform, such as the C and C++ family. For instance, if the execution environment is comprised of the Intel x86 Microprocessor and Windows operating system, then the native code has an Intel instruction set and portable execution (PE) format. For this reason, the approaches to software birthmarking can vary depending on the managed

Manuscript received September 18, 2015.
Manuscript revised March 18, 2016.
Manuscript publicized April 15, 2016.

[†]The authors are with College of Information and Communication Engineering, Sungkyunkwan University, Suwon, 440–746 Korea.
[††]The author is with Department of Cyber Security, Howon University, Gunsan, 440–746 Korea.
a) E-mail: dhlee@security.re.kr
b) E-mail: dwkang@security.re.kr
c) E-mail: yschoi@howon.ac.kr
d) E-mail: jykim@security.re.kr
e) E-mail: dhwon@security.re.kr (Corresponding author)

DOI: 10.1587/transinf.2015EDP7379

and native code. The software birthmarking approach proposed in this study involves native code.

Software birthmark technologies are primarily classified into dynamic birthmarks and static birthmarks [18]. These two approaches can be classified according to the environment in which a birthmark is generated. Static approaches use the static information of a program and do not require the program execution to extract the inherent characteristics, whereas dynamic approaches extract program characteristics by recording the actual behaviors during the running of the program for a given input. The positive and negative aspects of these two approaches are outlined as follows [19]:

- Dynamic birthmarks cannot feasibly cover all possible program paths and thus can only detect the theft of an entire program. Static birthmarks, on the other hand, cover all components of a program; therefore, theft of an entire program and of individual modules can both be detected.
- Dynamic birthmarks are highly dependent on the given input and runtime environments, whereas static ones are not.
- The extraction of a dynamic birthmark, which must collect information during the program execution, is more difficult to implement than that of a static one.
- Static birthmarks are extracted by static program analysis, which tends to over-approximate program properties. Hence, static birthmarks are less credible and less precise than dynamic ones.

Software birthmarks can be classified into three categories depending on the extraction targets: instruction-sequence-based, graph-based, and API-based birthmarks.

1.1 Instruction-Based Birthmarks

A program basically consists of data and instructions. In previous studies, an instruction sequence is frequently used as a birthmark because it reflects some program behaviors. A static n-gram-based birthmark extracted with Java bytecode/opcode was proposed by Myles and Collberg [9]. In their study, a set of all opcode n-grams of the methods in the class was extracted as a birthmark for the Java class. In addition, Lu et al. proposed a dynamic n-gram-based birthmark that is extracted from dynamic opcode sequences during program execution [20].

The opcode n-grams for these two birthmarks are computed by sliding a window of length n for continuous opcode sequences. In one of our previous studies, we proposed a method of generating a birthmark into categories by grouping the opcode of instruction sequences extracted from a program [21]. In that study, the opcode was classified into categories, such as data transfer, logical, and I/O, to improve the resilience; moreover, the continuously repeating opcode was reduced to increase efficiency. Unlike that partition method, Lim et al. considered using sequences of contiguous opcode that is partitioned based on its operand stack depth [22]. These three birthmark techniques consider the use of physical instruction sequences and are thus vulnerable to control-flow or program modifications [19].

1.2 Structure-Based Birthmarks

Software birthmarks have also been proposed in the graph structure of a program. Each function (alternatively, procedures on native code) of a program can be expressed as the dependence among statements in a function, the inheritance relationship between classes (such as acyclic graphs), and the control flow. Accordingly, a birthmark can be generated as an expression of a program graph [23]. Myles and Collberg proposed for Java applications a dynamic birthmark called the whole program path (WPP) [8]. To extract the WPP birthmark, dynamic traces of a program are compressed into a directed acyclic graph and then collected. However, comparing two graphs with millions of nodes may prove prohibitively expensive; moreover, it is unclear how this birthmark would perform on substantial traces of real programs [23].

1.3 API-Based Birthmarks

Several birthmarks exist that are based on the way a program uses standard libraries or system calls (henceforth collectively referred to as APIs); such a birthmark is both unique to that program and difficult for an attacker to forge [23]. We classify these birthmarks as API-based ones. Tamada et al. presented three algorithms for collecting birthmarks. These algorithms compute the birthmark from the sequence of method calls within a class, the inheritance path from the root class to a given class, and the types that a class employs, respectively, [24], [25]. In addition, Park et al. proposed a static API-call-based birthmark for software theft detection of Java applications [26]. Choi et al. additionally presented a static API birthmark for Windows execution files using a set of API calls identified as being static by a disassembler [27]. In addition to the above static birthmark-generation techniques, several dynamic API-based birthmarks have been proposed. Tamada et al. suggested a method of tracing the API calls of programs that are executed by particular input values [11]. Schuler et al. proposed a method of combining k-gram-based birthmarks and API-based birthmarks [10]. These researchers constructed a set of k-grams for API call sequences and proposed dynamic k-gram API-based birthmarking using an API call sequence that is well known to the program being executed with particular input values.

The most recent API-based software birthmark technique was proposed by Han et al. [28]. They generated a birthmark based on a program's API call sequence. Furthermore, by creating a birthmark database, their system can effectively detect software copyright infringement in the online service provider (OSP) or P2P environment. Such a birthmark extracts API call sequences of the whole program and then saves and compares them in the birthmark database by using a cryptographic hash function (MD5). They reported that different types of applications can be classified into categories (multimedia player, FTP client, text editor, etc.) through a prefiltering process, which is based on the numbers and names of DLLs and APIs in a program.

In this study, we intend to resolve the limitations of the software birthmarking technique proposed by Han et al. in [28]. Our proposed software birthmarking method extracts API call sequences in the segmented procedures for the native code of a program and then generates the sequences through a fuzzy hash function. In short, the contributions of our work can be summarized as follows:

- We improve the most recent API-based software birthmarking technique, which is proposed in [28]. Detailed descriptions of the shortcomings of that method are provided in Sect. 3.
- Our proposed birthmarking method is separated into procedures in a single program and is generated by creating API call sequences within each procedure.
- Each birthmark procedure is compressed into a fuzzy hash function for similarity measures.
- We prove the effectiveness of our method through experimental results.

The remainder of this paper is organized as follows. In Sect. 2, we explain relevant existing knowledge to elucidate our proposed method. The software birthmarking technique proposed in [28] is refined in this study. We describe that existing technique and its problems in Sect. 3. In Sect. 4, we provide a detailed description of our proposed method. In Sect. 5, we explain the implementation of the birthmarking system. In Sect. 6, we discuss the experiments conducted on the proposed system. Our discussion is provided in Sect. 7, and Sect. 8 presents our conclusions and future work.

2. Preliminaries

2.1 Software Birthmark

The software similarity problem has conventionally focused on code theft detection. It is used to determine if program $\mathcal{P}$ is a copy or derivative of program $\mathcal{Q}$. It is an extension of the definition in [11] and [18]. A workflow is shown in Fig. 1.

Definition 1: (**Software Birthmark**) Let $\mathbb{P}$ denote all sets of a program, where program $\mathcal{P}$ is given as $\mathcal{P} \in \mathbb{P}$. Let $\mathfrak{B}$ denote a function capable of extracting a set of program characteristics. If the following conditions are satisfied, the birthmark $\mathfrak{B}(\mathcal{P})$ of program $\mathcal{P}$ can be defined.

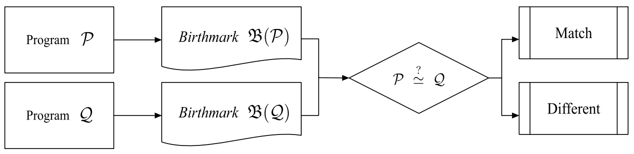

Program $\mathcal{P}$
Program $\mathcal{Q}$
Birthmark $\mathfrak{B}(\mathcal{P})$
Birthmark $\mathfrak{B}(\mathcal{Q})$
$\mathcal{P} \overset{?}{\simeq} \mathcal{Q}$
Match
Different

Fig. 1 Software similarity problem

- $\mathfrak{B}(\mathcal{P})$ is obtained only from $\mathcal{P}$ itself.
- Program $\mathcal{Q} \in \mathbb{P}$ is a copy of $\mathcal{P} \rightarrow \mathfrak{B}(\mathcal{P}) = \mathfrak{B}(\mathcal{Q})$.

As shown in Def 1, the software birthmark reflects the innate traits extracted from program $\mathcal{P}$ itself. Such a software birthmark is a technology designed to measure the similarity of two programs. If similarities of birthmarks extracted from two programs are matched, then the two programs can be considered identical or copied.

In order to evaluate the effectiveness of software birthmarks, researchers usually consider two properties: credibility and resilience [29].

Definition 2: (**Resilience**) Let us first assume there are two programs or program components $\mathcal{P}, \mathcal{Q} \in \mathbb{P}$. Then, let us say that $\mathfrak{B}(\mathcal{P}) \rightarrow \alpha$ and $\mathfrak{B}(\mathcal{Q}) \rightarrow \beta$ are the birthmark values extracted from programs $\mathcal{P}$ and $\mathcal{Q}$. Let $\mathbf{Sim}_p(\alpha, \beta) \rightarrow [0, 1]$ be a function that measures program similarity; the threshold is given as $0 < \varepsilon < 1$. If $\mathcal{P}$ and $\mathcal{Q}$ are similar to each other and $\mathbf{Sim}_p(\alpha, \beta) > 1 - \varepsilon$, then the birthmarking system is called resilient.

Definition 3: (**Credibility**) Let $\mathcal{P}$ and $\mathcal{Q}$ denote independently written programs. If the birthmarking system can distinguish between the two programs, it is deemed reliable and can be defined as follows:

$$\mathbf{Sim}_p(\alpha, \beta) \leqslant \varepsilon \tag{1}$$

Defs. 2 and 3 define the basic properties in measuring similarity between two birthmarks. Credibility defines the property in which comparison values of programs from the birthmarking system can be clearly sorted.

Generally, software birthmarking can be extracted with two approaches: original source code (including managed code) and compiled native code. One benefit of the extraction approach using original source code is its easy extraction of characteristics through intuitive searching of the flow of code written by a developer. However, it is not suitable for real-world applications because it is more common to perform it in the absence of original source code when detecting software copyright infringement or code theft. Another shortcoming is that code with the same flow can be differently analyzed because various factors, such as developer coding style, are reflected in this approach.

On the other hand, birthmarks can be extracted from native code only with machine code and thus do not require original source code. For its extraction, this native code that is dependent on the platform (microprocessor, operating system, and so on) is subject to the disassembling process and code flow analysis by an experienced professional.

Our challenge is to extract the software birthmark from native code. At the same time, our ultimate goal is to measure similarity between one program and other multiple programs or between two programs.

2.2 Context-Triggered Piecewise Hashing

The cryptographic hash algorithm produces a fixed output for input files. Such outputs are unique values, and the two files are considered identical if they have the same hash value. The traditional cryptographic hash function is generally used for data integrity checks in the field of information security.

A fuzzy hash can split input values into several pieces and then produce a single outcome in each piece by using the hash. In this way, similarity can be evaluated between two input values. Early fuzzy hash approaches use a method of comparing the hash values by splitting input values into fixed-size blocks by employing a block-based hash [30]. However, in these approaches, a different offset caused by the manipulated input values (insertions, deletions, editing) alters the components of the block and affects the overall outcome. To overcome this drawback, the context-triggered piecewise hash (CTPH) was proposed by Kormblum et al. [31]. CTPH recognizes an identifier capable of identifying the context of input values and uses a hash as an identifier for the split block.

For example, if a character is moved down by editing a page, the character contained in all pages is edited in the comparison process of two documents. This makes it difficult to determine the similarity between the two documents in the conventional method. CTPH, however, employs a method of dividing by paragraphs, not by pages, and thus similarity can be identified through only edited paragraphs while others remain unchanged.

Ssdeep is a tool that was also implemented by Kornblum [32]. It provides a function that produces fuzzy hash outcomes of input values and a function that computes similarity by comparing those outcomes. For CTPH outcomes in ssdeep, the processed outcomes of all blocks are produced by employing the spamsum algorithm that outputs the top six characters by Base64 encoding and by applying a corresponding block-scrolling hash and traditional hash. The algorithm is presented in Fig. 2.

The two signatures are recorded in the outcome: *signature*1 is the comparison of output block sizes, and *signature*2 is the computation of the blocks that are doubled in size.

The fuzzy hash has recently been used in measuring program similarity (i.e., malware) in the field of digital forensics. Roussev et al. compared the similarities of malware and different types of normal files, such as documents or images, by using ssdeep, and they confirmed its efficiency [33]. In 2008, more results of ssdeep were added by Virus Total for detecting malware [34]. Furthermore, in 2009, the FTK forensic tool also added the ssdeep function for its forensic investigations [35].

Despite the above advances, an issue remains with such a comparative approach. This was indicated in 2010 by the Carnegie Melon University computer emergency response team, who demonstrated that a comparison of similarity over the entire file suffers from low efficiency [36]. The built-in code in a program basically contains various constant values by default. This may change during recompiling even if it was derived from the same source code. Therefore, the hash value extracted from the entire file is not suitable for determining similarity. Our proposed birthmarking technique is broken into procedures in a program to apply the fuzzy hash function.

```
b = compute_initial_block_size( input )

done = FALSE
while ( done = FALSE ) {
      initialize_rolling_hash( r )
      initialize_traditional_hash( h1 )
      initialize_traditional_hash( h2 )
      signature1 = " "
      signature2 = " "

      for each byte d in input {
            update_rolling_hash( r, d )
            update_traditional_hash( h1, d )
            update_traditional_hash( h2, d )

            if ( get_rolling_hash( r ) mod b = b - 1 ) then {
                  signature1 += get_traditional_hash( h1 ) mod 64
                  initialize_traditional_hash( h1 )
            }

            if ( get_rolling_hash( r ) mod ( b * 2 ) = b * 2 - 1 ) then {
                  signature2 += get_traditional_hash( h2 ) mod 64
                  initialize_traditional_hash( h2 )
            }
      }

      if length( signature1 ) < S / 2 then
            b = b / 2
      else
            done = FALSE
}

signature = b + ":" + signature1 + ":" + signature2
```

Fig. 2 Pseudocode for the spamsum algorithm.

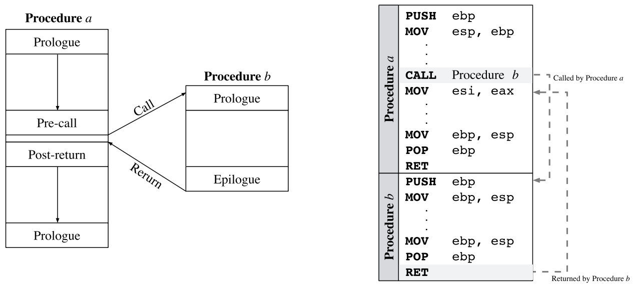
Definition 4: (**Fuzzy Hash**) Input value $\mathcal{K}$ is given, and let us say that $\mathcal{K} = \{k_1, k_2, \ldots, k_{n-1}, k_n\}$ has n size blocks. Let h denote the function capable of generating the fuzzy hash. When $h(\mathcal{K}) \rightarrow \mathcal{A}$ denotes a fuzzy hash value for the input value $\mathcal{K}$, it is noted that $\mathcal{A}$ satisfies the following conditions:

- When $\mathcal{A} = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$, a_i and k_i are in the bijection and a_i is represented by only k_i.
- After obtaining $h(\mathcal{K}) \rightarrow \mathcal{A}$, block $k_i \in \mathcal{K}$ must be $\mathcal{A} \simeq \mathcal{A}'$ even when $h(\mathcal{K}') \rightarrow \mathcal{A}'$ is obtained with different values that were changed by specific manipulation or editing.

We define the fuzzy hash function, h, if particular conditions are satisfied as described in Def. 4. With the application of the fuzzy hash function on the program characteristics, the similarities between programs can be measured on account of its properties. A detailed description is provided in Sect. 4.

2.3 Procedure

The procedure (subroutine, method, or function) is an important part of any computer system's architecture. A procedure is a group of instructions that usually performs one task. In this paper, the function and procedure are differently designated. A function is defined as something that can be semantically classified by human beings on original source code or managed code. A procedure is defined as something



```
PUSH  ebp
MOV   esp, ebp
  .
  .
  .
CALL  Procedure b
MOV   esi, eax
  .
  .
  .
MOV   ebp, esp
POP   ebp
RET
PUSH  ebp
MOV   ebp, esp
  .
  .
  .
MOV   ebp, esp
POP   ebp
RET
```

Fig. 3 A Standard procedure linkage

that can be classified by machines on native code.

Each procedure has a prologue sequence and an epilogue sequence. The former is a small line of code at the beginning of a procedure that prepares the registers and stacks to be used inside the procedure. Similarly, the epilogue procedure appears at the end of the procedure and restores the registers and stacks to their original conditions before the procedure is called. Figure 3 shows how the pieces of a standard procedure linkage fit together.

Definition 5: (**Procedure**) Given a program $\mathcal{P} \in \mathbb{P}$, let us assume that there is a universal set of procedures $\mathcal{P} = \{f_1, f_2, \ldots, f_{n-1}, f_n\}$, where n is the size of the procedures in a program $\mathcal{P}$. Also, let us assume that f_i is a specific i^{th} procedure of the sequence in $\mathcal{P}$, where i has $0 < i \leq n$. Then, f_i is defined as the procedure of program $\mathcal{P}$ when the following conditions are met.

- f_i is a set of insturctions that is called more than once when program $\mathcal{P}$ is executed.
- If f_i is called from f_j, then f_i must include the instructions (i.e., epilogue) that can be returned to f_j with certainty.
- When f_i is forwarding control to the other f_j, it is possible only through the call instruction, which is the prologue of f_j.
- f_i called by the call instruction must be a particular reference value or a set of instructions in the $\mathcal{P}$ (a property that also includes the API).

In this paper, according to Def. 5, the birthmark is extracted by separating procedures from the disassembled instruction sequences of a program. However, the procedures without the API call and the instructions beyond the procedure scope are excluded. This is because the accurate program characteristics must be reflected to measure similarity using the software birthmarking method herein proposed.

3. Han et al. Birthmark Review

As mentioned in Sect. 1, Han et al. proposed the most recent API-based software birthmark method [28]. It detects, blocks, and removes software that has been illegally copied and distributed by unauthorized users or malicious programmers on an OSP or P2P network. Their method covers files that are executable on the Microsoft Windows platform. Such Windows-executable files have conventionally been in portable executable (PE) format. The method is based on information that is extractable from the PE format of execution files, specifically the:

- Number of DLLs and their names
- Number of APIs and their names
- Sequences of API calls

First, the numbers of DLLs and APIs and their respective names are extracted from the import address table within the PE format. The sequences of API calls are extracted by analyzing each instruction from the code segment inside the PE format. Second, the API call sequences are converted into their representative values using a one-way cryptographic hash function. The extracted information is classified and saved by the category and software type, as shown in Fig. 4(a). The extraction process ends by saving this information in a birthmark database (as shown in Fig. 4(b)).

In this system, the uploaded program undergoes a four-step identification process to detect illegal programs that are available on OSP or P2P networks. A detailed description is as follows:

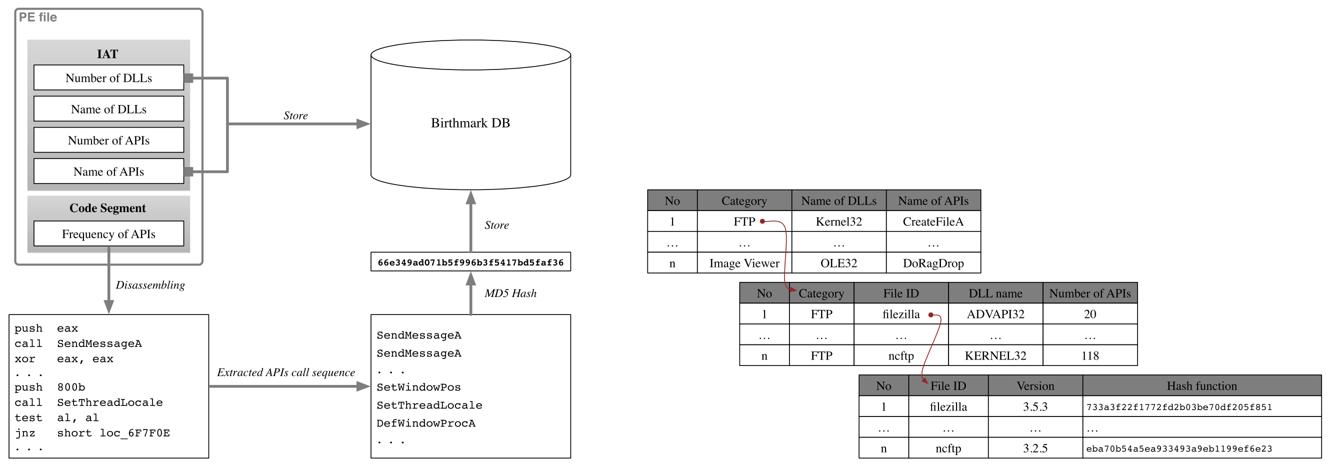
- **Step 1.** The uploaded program is sorted into one of the categories classified in the birthmark database. This step helps compare information in the corresponding category of the uploaded program without requiring a search of the whole, large birthmark database.

- **Step 2.** The uploaded program compares the number of APIs and the names of DLLs for the programs in the classified categories. If the uploaded program is not identified in this step, then the process advances to Step 3.

- **Step 3.** In the Han et al. approach, the code segments in the PE format of execution files are extracted; then, the API call sequences are saved in the birthmark database through an MD5 hash function. In this step, unidentified programs from Step 2 attempt another round of identification through a hash value comparison for the API sequences of the same category programs saved in the database. If this step fails again, then the process advances to Step 4.

- **Step 4.** Steps 2 and 3 identify only the programs in the categories classified in Step 1. However, if they are not identified until Step 3, then all programs in the birthmark database must be identified through an MD5 hash value comparison of API call sequences. This procedure is required on account of the possibility of incorrect categorization from Step 1.

The Han et al. birthmark method has three problems. First, it cannot achieve accurate category classification using the names of DLLs and numbers of APIs. APIs can be defined as a list of functions that operating systems provide



```
push  eax
call  SendMessageA
xor   eax, eax
. . .
push  800b
call  SetThreadLocale
test  al, al
jnz   short loc_6F7F0E
. . .
```

```
SendMessageA
SendMessageA
. . .
SetWindowPos
SetThreadLocale
DefWindowProcA
. . .
```

```
66e349ad071b5f996b3f5417bd5faf36
```

No	Category	Name of DLLs	Name of APIs
1	FTP	Kernel32	CreateFileA
…	…	…	…
n	Image Viewer	OLE32	DoRagDrop

No	Category	File ID	DLL name	Number of APIs
1	FTP	filezilla	ADVAPI32	20
…	…	…	…	…
n	FTP	ncftp	KERNEL32	118

No	File ID	Version	Hash function
1	filezilla	3.5.3	733a3f22f1772fd2b03be70df205f851
…	…	…	…
n	ncftp	3.2.5	eba70b54a5ea933493a9eb1199ef6e23

(a) Processes for birthmark extraction in the PE format

(b) The Schema of the birthmark database

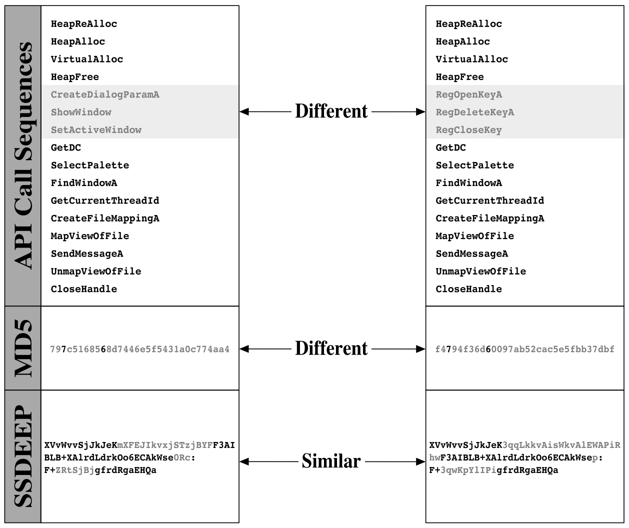
Fig. 4 Birthmarking system proposed by Han et al. [28]

to application developers. In particular, the Windows system offers APIs by loading the DLL-encapsulated modules in shared memory so they are available for a wide variety of applications. If DLLs are classified by function, several separate DLLs result that must remain in memory at all times. For this reason, Windows sets DLLs into specific classes to make them smaller in number and retained in memory. Therefore, the application properties cannot be classified by only a name, such as USER32.dll, KERNEL32.dll, or ADVAPI32.dll, which has not been clearly classified by function.

Furthermore, because multifunctional applications have recently become available in response to user demand, it has become difficult to determine characteristics of applications simply by the API call frequency alone. For example, recently available text editor programs have image editing functions and they save files by synchronizing with a cloud server. In this case, the APIs of the text editor program inherently contain image editing functions and networks—they are image-related APIs by default—and thus they can be similar to applications that perform such actual functions as described above.

Second, the API call sequences are extracted from the entire areas of a program. When program source code is compiled, the compiler generally reflects the function of the original source code as intact as possible. In particular, if the API call command is invoked from a specific function of the original source code, then it can also exist in the corresponding function area for the compiled program command sequences. Hence, even in the same source code, a different API call sequence may be generated if positional changes are made on the function area during compiling. This includes the possibility of a detour. The software birthmarking method proposed in this paper consists of API call sequences separated into procedures in a single program. Accordingly, it is not affected by the position of the function that can be modified during compiling.

Lastly, the use of the cryptographic hash function is not



```
HeapReAlloc
HeapAlloc
VirtualAlloc
HeapFree
CreateDialogParamA
ShowWindow
SetActiveWindow
GetDC
SelectPalette
FindWindowA
GetCurrentThreadId
CreateFileMappingA
MapViewOfFile
SendMessageA
UnmapViewOfFile
CloseHandle
```

```
797c51685 68d7446e5f5431a0c774aa4
```

```
XVvWvvSjJkJeKmXFEJIkvxjSTzjBYFF3AI
BLB+XAlrdLdrkOo6ECAkWse0Rc:
F+ZRtSjBjgfrdRgaEHQa
```

```
HeapReAlloc
HeapAlloc
VirtualAlloc
HeapFree
RegOpenKeyA
RegDeleteKeyA
RegCloseKey
GetDC
SelectPalette
FindWindowA
GetCurrentThreadId
CreateFileMappingA
MapViewOfFile
SendMessageA
UnmapViewOfFile
CloseHandle
```

```
f4794f36d60097ab52cac5e5fbb37dbf
```

```
XVvWvvSjJkJeK3qqLkkvAisWkvAlEWAPiR
hwF3AIBLB+XAlrdLdrkOo6ECAkWsep:
F+3qwKpYlIPigfrdRgaEHQa
```

Fig. 5 Comparison of a cryptographic hash and fuzzy hash for API call sequences

suitable for determining the similarity of program characteristics. Software birthmarks have conventionally been studied to measure similarity between programs. Therefore, a software birthmarking method must be designed to measure similarity between programs.

As mentioned, the Han et al. birthmarking approach uses a cryptographic hash function on the API call sequence. The cryptographic hash function is a one-way function that is largely used in data integrity checking because it produces totally different outputs, even with a minor change in the original source data [37]. For instance, when a pair of API call sequences that are similar to each other is given, as shown in Fig. 5, a cryptographic hash function, such as MD5, produces completely different output. In contrast, a fuzzy hash function, such as CTPH, as described in Sect. 2, maintains the similarity of the original data. Therefore, it is more appropriate to use the fuzzy hash function rather than the cryptographic hash function in evaluating the similarity of two data sets.

The birthmarking method proposed in this paper enhances the approach of Han et al. by resolving its limitations. The proposed method is detailed in Sect. 4.

4. Proposed Method

4.1 Feature Extraction

A software birthmark can be defined as an inherent property of a program. Our proposed software birthmarking method is based on procedures that contain API calls in the distributed programs with native code. In other words, these characteristics are extracted from a single program and are described as follows:

- A set of procedures that includes the API call.
- Sequences of API calls in each procedure.
- DLL and API information embedded in a target program.

In [28], the API call sequences are extracted from the entire code section in a program, and a birthmark is generated via an MD5 hash function. A program fundamentally consists of functions or procedures. Therefore, it is inappropriate to generate sequences over the entire code section in a program. A cryptographic hash function is also unsuitable for the similarity measure. This is a motivation for the software birthmark because it produces completely different output even with a minor change in the original data. In Sect. 5, relevant experiments that confirm the above-mentioned findings are described.

In the our proposed method, the instructions in a single program that are first disassembled to extract the program characteristics are classified in the procedure unit. Next, the sequences of the API call commands, which are among the call commands in each function, are generated. Finally, the API call sequences of a function generate their values by the fuzzy hash function and then save the values in the database.

As described earlier, applying the fuzzy hash function on all of the data for the data similarity measure is not recommended [36]. Accordingly, our proposed technique properly applies the fuzzy hash function on the procedures that can be classified into regular units within a program.

4.2 Birthmark Generation

The proposed method is based on the API call sequence of the segmented procedures in a single program. In our method, a procedure must be satisfied with all conditions of Def. 5 and the following Eq. (2):

$$f_i = \{API_1, API_2, \ldots, API_m \mid m > 0\}, \tag{2}$$

where m is a size of the API call sequence in a procedure, $f_i \in \mathcal{P}$. Then, the birthmarking procedure is defined as below.

Definition 6: (**Birthmarking Procedure**) Given program $\mathcal{P}$, let us assume that there is a procedure $f_i \in \mathcal{P}$, where i is a specific i^{th} procedure of sequence of $\mathcal{P}$. Also, let $\mathfrak{B}_f$ be the function that generates a birthmark for the procedures. Then, the procedure birthmark $\mathfrak{B}_f(f_i)$ is defined as follows:

$$\mathfrak{B}_f(f_i) = h\left(\bigcup_{k=1}^{m} API_k\right), \tag{3}$$

where h is fuzzy hash function that was defined as Def. 4 in the previous section. A birthmarking procedure individually extracted from a single program generates values through the fuzzy hash function for the similarity measure. In [28], the birthmarking method was that generates values through the MD5 cryptographic hash function for the API call sequences extracted from the entire area of the program. This method, as indicated in Sect. 3, has several problems.

To overcome these drawbacks, the fuzzy hash function h is applied on m API call sequences $\{API_1, API_2, \ldots, API_m\}$ extracted from each procedure in this study (as shown in Eq. (3)). Next, we define the birthmarking program as Def. 7.

Definition 7: (**Birthmarking Program**) Let us assume that there is a universal set of procedures, $\mathcal{P} = \{f_1, f_2, \ldots, f_n\}$, which has a procedure size of n in a given a program $\mathcal{P}$. Program birthmark function $\mathfrak{B}_p$ is defined as a $\mathfrak{B}_f(f_i)$ list with $\mid \mathcal{P} \mid = \mid \mathfrak{B}_p(\mathcal{P}) \mid \rightarrow n$ length by birthmark procedure $\mathfrak{B}_f$. This can be defined as follows:

$$\mathfrak{B}_p(\mathcal{P}) = \left\{\bigcup_{i=1}^{n} \mathfrak{B}_f(f_i)\right\} \tag{4}$$

4.3 Similarity Measure for Fuzzy Hashing Using the Weighted Edit Distance

The fuzzy hash function used in this paper is CTPH [31]. The CTPH spamsum algorithm uses the weighted version of the edit distance formula developed for the USENET newsreader [38]. In this version, each insersion or deletion is weighted as a difference of one; however, each change is weighted as three and each swap (i.e., the correct characters but in the reverse order) is weighted as five. For the sake of clarity, this weighted edit distance is defined as $e(s_1, s_2)$ when two signatures, s_1 and s_2, are given, as illustrated in Eqs. (6) and (7). In these equations, i is the number of insertions, d is the number of deletions, c is the number of changes, and w is the number of swaps.

$$e = i + d + 3c + 5w \tag{5}$$

$$c + w \leq min(l_1, l_2) \tag{6}$$

$$i + d = |l_1 - l_2| \tag{7}$$

where l_1 and l_2 are the lengths of s_1 and s_2, respectively. The edit distance is then rescaled from $0 - 64$ to $0 - 100$ and inverted so that zero represents no homology, and 100 indicates almost identical files. The final match score, M, for strings of length l_1 and l_2 can be computed using Eq. (8).

$$M = 100 - \left(\frac{100S\, e(s_1, s_2)}{64(l_1 + l_2)} \right), \tag{8}$$

where S is represented by rescale constant. It rescaled and inverted the degree of file similarity from $0 - S$ to $0 - 100$. For example, when $S = 64$, 0 is rescaled to 100, which indicated identical files, and $S = 64$ is rescaled to 0, which indicated no homology. Note that when $S = 64$, which is the default, that the S and 64 terms are cancelled.

The match score represents a conservative weighted percentage of the extent to which s_1 and s_2 are ordered in the homologous sequences [31]. That is, it denotes the measure of how many of the bits of these two signatures are identical and in the same order.

Finally, in Eq. (9), match score M is defined as the similarity measure function $\mathbf{Sim}_f$ for the proposed birthmarking procedure. Let us assume that $\mathfrak{B}_f(f_1) \to \alpha$ and $\mathfrak{B}_f(f_2) \to \beta$ are obtained from the two given procedures, f_1 and f_2, respectively. This enables the measuring of similarity of the two birthmarking procedures through similarity measure function $\mathbf{Sim}_f$.

$$\mathbf{Sim}_f(\alpha,\beta) = 100 - \left(\frac{100S\, e(\alpha,\beta)}{64(\mid \alpha \mid + \mid \beta \mid)} \right) \tag{9}$$

4.4 Program Similarity Measuring

Our ultimate goal is to measure similarity between two programs. To achieve this, a list of birthmarking procedures using a fuzzy hash is generated through a set of procedures that generate API calls in a single program. With this list, similarity is measured between the two programs.

The similarity measure is known as the dice coefficient or Sorensen index. It is used for information retrieval. By expanding this index, the similarity measure function between two programs is defined.

Definition 8: (**Program Similarity**)

Let us assume there are two programs or program components $\mathcal{P}, Q \in \mathbb{P}$. Also, let us say that $\mathfrak{B}_f(f_i \in \mathcal{P}) \to \alpha_i$ and $\mathfrak{B}_f(f_j \in Q) \to \beta_j$ are the birthmark values extracted by the birthmarking procedure function $\mathfrak{B}_f$. Then, program similarity $\mathbf{Sim}_p(\mathcal{P}, Q)$ is defined as:

$$\mathbf{Sim}_p(\mathcal{P}, Q) = \frac{2 \times \sum_{i=1}^{n} max\left(\bigcup_{j=1}^{k} \mathbf{Sim}_f(\alpha_i, \beta_j) \right)}{n + k} \tag{10}$$

where n and k represent the number of procedures that contain API calls for programs $\mathcal{P}$ and Q, respectively. α_i and β_j are the values generated through $\mathfrak{B}_f$ for each procedure extracted from the two programs $\mathcal{P}$ and Q.

5. Implementation

In this section, the implementation of the proposed birthmarking method is described. The method is comprised of five steps. The birthmark is extracted through four steps after initial program $\mathcal{P}$ input and is then saved in the database. From that point, similarity is measured between the procedures or other programs as a whole in two steps. The overall workflow is presented in Fig. 6.

Most existing studies evaluate similarity for two programs, whereas the proposed method evaluates similarity in five steps and enables the automation of the comparison of multiple programs. It can achieve this because it evaluates similarity with proceduralized birthmarks of all programs saved in the database. Moreover, birthmarks are saved separately in the procedure unit. Therefore, similarity can be measured in part rather than over the entire program. Unlike previous studies in which the entire program is compared, the proposed method can detect the copied code in a procedure. Furthermore, the birthmark information downscaled through the fuzzy hash is more effective than the conventional way of performing sequence-comparison processing.

- **Step 1.** A code area is extracted from the input program, $\mathcal{P}$, where the code area refers to a set of code interpreted and executed by the microprocessor. The executable files have particular formats according to the given operation system. For instance, Windows has a PE format, whereas Unix/Linux has formats such as ELF or COFF.

- **Step 2.** The code area extracted in Step 1 is disassembled. This disassembly process is needed to identify the meaning of the instruction sequences and to separate the procedures for extraction from which the API call sequences. The distorm3 program is herein used to perform reliable and automated disassembly. The basic structure of the procedures described in Sect. 2 is recognized for Step 3, which involves the separation of procedures. In addition, reference procedures for the call instructions are recognized through a recursive descent disassembly method, which focuses on the control flow of the disassembly processing method.

- **Step 3.** The disassembled instruction sequences of the program are separated into procedures. The sequences that fail to generate API calls in each procedure are excluded. The outcome from this step is a set of the procedure-intrinsic names and the API call sequences generated in it. The proposed system provides the disassembly outcomes of similar procedures for user review. In this way, the disassembly outcome can be selectively saved.

- **Step 4.** The final extraction step is to generate the hash values through the fuzzy hash function for the API call sequences of the separated procedures. They are then saved in a database. The database can be comprised of a table with information on the execution files, and a table with the procedure list connected to it with information on the execution files.

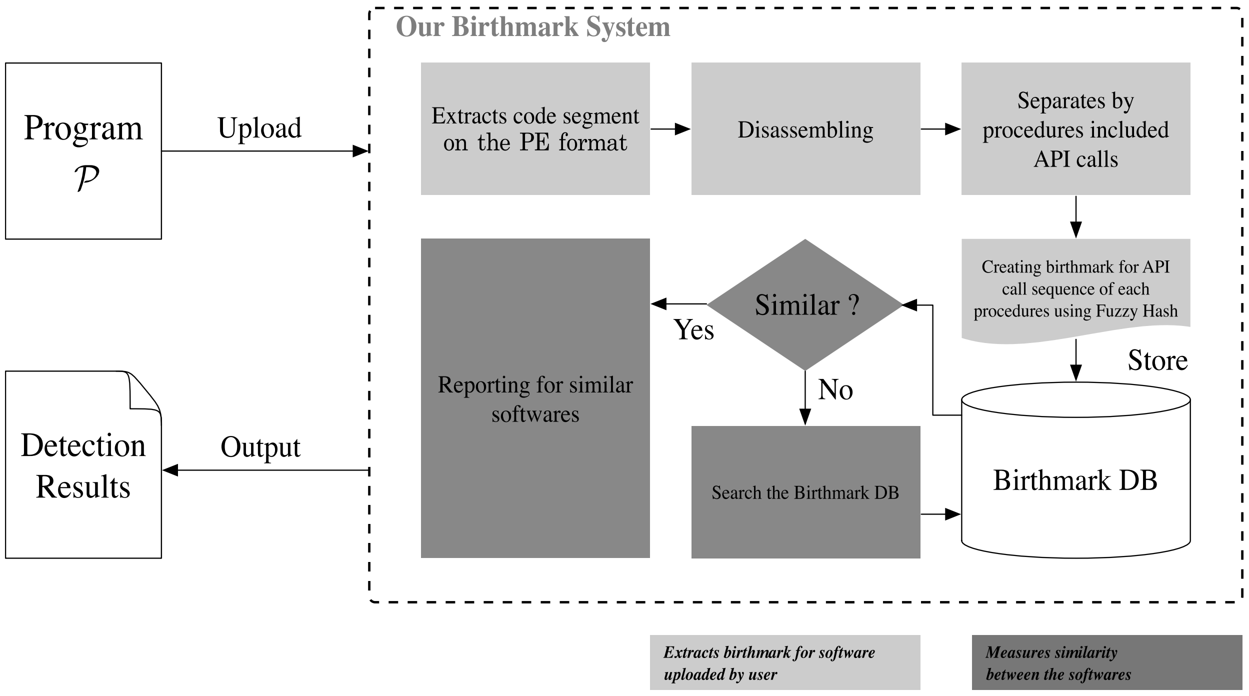

Our Birthmark System
Program $\mathcal{P}$
Upload
Extracts code segment on the PE format
Disassembling
Separates by procedures included API calls
Creating birthmark for API call sequence of each procedures using Fuzzy Hash
Store
Similar ?
Yes
No
Reporting for similar softwares
Search the Birthmark DB
Birthmark DB
Detection Results
Output
Extracts birthmark for software uploaded by user
Measures similarity between the softwares

Fig. 6 Proposed birthmak method workflow

In the method of Han et al., the numbers of DLLs and APIs are saved for category classification. However, the experiments conducted in this study prove that it is practically impossible to classify categories by identifying the characteristics of applications with the numbers of DLLs and APIs, which may cause false-positives. Moreover, recently available applications include numerous multifunctions. For this reason, it is highly likely that the categories will not be clearly classified but will rather be processed at a great expense. Therefore, the proposed system focuses only on the similarity of a program itself.

5.1 Similar Relation Assessment

Definition 9: (**Similarity Measure**) If Q is derived from the same source as $\mathcal{P}$ (namely, different versions of the same program), where $\mathcal{P}$ and Q are given, we can let $\mathcal{P}$ and Q be similar to each other and write them as $\mathcal{P} \simeq Q$. If $\mathcal{P} \simeq Q$, then the two values α and β obtained by $\mathfrak{B}_p(\mathcal{P}) \rightarrow \alpha$ and $\mathfrak{B}_p(Q) \rightarrow \beta$ can also be defined as $\alpha \simeq \beta$. In this manner, the similarity measure function $\mathbf{Sim}_p(\alpha, \beta)$ defined in this paper can be defined as follows:

$$\mathbf{Sim}_p(\alpha,\beta) = \begin{cases} > 1-\varepsilon & \alpha \simeq \beta \\ \leqslant \varepsilon & \alpha \not\simeq \beta \\ \text{otherwise} & \text{inconclusive} \end{cases} \tag{11}$$

In Sect. 6, we determined the threshold as being $\varepsilon = 0.35$. From the threshold, a similarity range of $[0.0, 0.35]$ is classified as independent, $(0.35, 0.65]$ as inconclusive, and $(0.65, 1]$ as that of similar programs.

Schuler et al. employed such a classification scheme with a threshold of 0.2, such that the similarity range $[0, 0.2]$ is classified as independent, $(0.2, 0.8)$ as inconclusive, and $[0.8, 1]$ as copies. In [29], the authors classify programs with a slightly different scheme, wherein the similarity range $[0, 1 - 2\varepsilon)$ is classified as independent, $[1 - 2\varepsilon, 1 - \varepsilon)$ as inconclusive, and $[1 - \varepsilon, 1]$ as similar with to $\varepsilon = 0.2$. Accordingly, the similarity range $[0, 0.4)$ is classified as independent, $[0.4, 0.8]$ as inconclusive, and $(0.8, 1]$ as copies. However, Schuler and Myles did not explain the threshold determination.

Generally, the previous works were evaluated with each other threshold ε. Our decision to set the threshold to $\varepsilon = 0.35$ is related to the our experimental results in the properties of resilience and credibility. In Sect. 6, our evaluations performed with Windows applications show that is satisfied with $\varepsilon = 0.35$ as the properties.

6. Evaluation

Our experimental environment was comprised of the 32-bit Windows 7 operating system, an Intel Core i7 2.6-GHz processor, and 16Gb of RAM. The system was embodied by Python, and the packages of `pefile`, `distorm3`, and `ssdeep` were used. Furthermore, IDA Pro 6.1 of Hex-Rays was used for the verification of the disassembly.

Our two-part assessment was comprised of a resilience evaluation and credibility evaluation. First, birthmark resilience was examined and the problems of the Han et al. approach were investigated. Second, credibility was evaluated. The results demonstrated the $\leqslant \varepsilon$ relationship between independent programs. In addition, a correlation coefficient analysis was conducted using API call frequency to examine the problems of the Han et al. application category classifi-

Table 1 Similarity measure for `putty` version-specific information using the software birthmarking method proposed by Han et al.

Ver.	`0.52`	`0.53`	`0.54`	`0.55`	`0.56`	`0.57`	`0.58`	`0.59`	`0.60`	`0.61`	`0.62`
`0.52`	1.0	0.44	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`0.53`	0.44	1.0	0.41	0.41	0.44	0.0	0.0	0.0	0.0	0.0	0.0
`0.54`	0.0	0.41	1.0	0.93	0.82	0.0	0.0	0.0	0.0	0.0	0.0
`0.55`	0.0	0.41	0.93	1.0	0.91	0.0	0.0	0.0	0.0	0.0	0.0
`0.56`	0.0	0.44	0.82	0.91	1.0	0.0	0.0	0.0	0.0	0.0	0.0
`0.57`	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
`0.58`	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.6	0.57	0.0	0.0
`0.59`	0.0	0.0	0.0	0.0	0.0	0.0	0.6	1.0	0.96	0.0	0.0
`0.60`	0.0	0.0	0.0	0.0	0.0	0.0	0.57	0.96	1.0	0.0	0.0
`0.61`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
`0.62`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0

Table 2 Similarity measure for `putty` version-specific information using our proposed software birthmarking method

Ver.	`0.52`	`0.53`	`0.54`	`0.55`	`0.56`	`0.57`	`0.58`	`0.59`	`0.60`	`0.61`	`0.62`
`0.52`	1.0	0.94	0.82	0.82	0.82	0.71	0.69	0.67	0.67	0.66	0.66
`0.53`	0.94	1.0	0.83	0.82	0.81	0.72	0.71	0.68	0.68	0.66	0.66
`0.54`	0.82	0.83	1.0	0.99	0.97	0.87	0.84	0.76	0.76	0.72	0.72
`0.55`	0.82	0.82	0.99	1.0	0.98	0.86	0.82	0.79	0.76	0.75	0.75
`0.56`	0.82	0.81	0.97	0.98	1.0	0.88	0.85	0.8	0.79	0.76	0.76
`0.57`	0.71	0.72	0.87	0.86	0.88	1.0	0.95	0.89	0.88	0.83	0.83
`0.58`	0.69	0.71	0.84	0.82	0.85	0.95	1.0	0.91	0.91	0.86	0.86
`0.59`	0.67	0.68	0.76	0.79	0.8	0.89	0.91	1.0	0.99	0.91	0.91
`0.60`	0.67	0.68	0.76	0.76	0.79	0.88	0.91	0.99	1.0	0.91	0.91
`0.61`	0.66	0.66	0.72	0.75	0.76	0.83	0.86	0.91	0.91	1.0	1.0
`0.62`	0.66	0.66	0.72	0.75	0.76	0.83	0.86	0.91	0.91	1.0	1.0

cation method.

6.1 Version-Specific Applications

Han et al. extracted a birthmark using a cryptographic hash function for the API call sequences over the entire area of a program. Our first experiment was intended to prove that their birthmarking system is resilient by using the sets of similar programs. To this end, similarity was measured for different versions (v0.52–0.62) of putty, a well-known remote terminal access program (see Table 1). Each compiler version was identified using the PEiD tool.

The original source code was modified each time that ten programs were respectively updated to the most recent version. Hence, we formulated a hypothesis. First, version-specific putty programs have some similarities. Second, with version updates, the given similarity to lower versions is weakened. The resulting difference shows that the largest similarity is observed between the oldest and newest versions. Lastly, the similarity to itself must always indicate a 100% perfect match.

Our first experiment was performed in two ways. One method concerned a birthmark generated with the fuzzy hash function by extracting the API call sequence from the entire program, as proposed by Han et al. The other method concerned a birthmark generated with the fuzzy hash function by extracting API call sequences from each procedure in a program, as herein proposed.

Table 1 presents the results of similarity measures performed by generating a birthmark through fuzzy hash function *h* in the entire area of putty version-specific programs. Fields with more than 40% similarity are highlighted. The

Table 3 `putty` version-specifics infromation

Ver	**Size(KB)**	**Num. Proc.**	**Num. Proc. (API Calls)**	**Compiler (Microsoft Visual C++)**
`0.52`	324	668	536	6.0
`0.53`	348	705	569	6.0
`0.54`	364	835	687	6.0
`0.55`	368	840	691	6.0
`0.56`	372	856	704	6.0
`0.57`	372	828	676	7.0
`0.58`	412	900	735	7.0
`0.59`	444	967	794	7.0
`0.60`	444	966	792	7.0
`0.61`	472	1015	832	7.0
`0.62`	472	1015	832	7.0

issues on the method of Han et al. can be found in the experimental results in Table 1. As shown, all similarity measurement results are 0.0, except for some versions. However, the only valid hash value is 1.0 because the MD5 hash function was applied in the Han et al. approach.

Furthermore, generating the API call sequences over the entire area of a program may generate false-negatives, even if programs are similar to each other. This depends on the positional changes in procedures during program compiling, regardless of program characteristics. From the results reported in Table 1, it is proved that the Han et al. approach does not satisfy resiliency, which is a key property of software birthmarking.

The experimental results of the proposed our method are presented in Table 2. The results show that all versions of putty are more than 66% similar. Moreover, the results from the hypothesis-driven examination show that similarity is weakened as the difference between versions becomes

Table 4 Various applications information for similarity measuring of birthmark

Category	Applications	Ver.	File Size (KB)	Num. Proc.	Num. Proc. (API Calls)	Size of Original Birthmark (KB)	Size of Birthmark applied Fuzzy Hashing (KB)
Text Editor	Sublime Text 2	2.0.2	3830	22153	2260	49	19
	Sublime Text 3	1.0.1	3981	30127	612	20	9
	notepad	6.1	176	135	84	9	2
	notepad++	6.7.4	2447	8637	855	84	29
	HxD	1.7.7	1643	3893	687	33	14
	UltraEdit32	22.0.0.51	14463	33685	7534	400	158
Emulator	DeamonToolsLite	4.49.1	3941	8361	2549	112	57
	CDSpace8	8.0.12	9803	13522	2683	132	61
Zip	Alzip	9.66	3037	14201	3251	189	76
	7zip	15.05b	256	2372	605	11	5
	7zipFM	15.05b	476	4827	1226	25	12
	PeaZip	5.7.0	5543	7434	422	17	7
Player	GOMPlayer	2.2.73	9182	50834	6404	248	110
	GOM Audio	2.0.10	4440	6243	1870	101	43
	KMPlayer	3.9.1	11862	28631	3837	117	48
	WinAmp	5.666	2273	2938	1656	149	50
Terminal	putty	0.65	512	1272	534	25	10
	SecureCRT	7.3.4	3619	23437	4422	191	82

greater. It is herein proved that the largest difference is observed between the oldest and newest versions with a similarity of approximately 66%, where 0.56 and 0.57 versions went through changes in their compilers.

To prove the properties of resilience (Def. 2) and credibility (Def. 3), we need to decide the threshold ε. In previous works, the threshold value was suitably decided by the experimental results of each method. Our first experimental result with the `putty` version-specific was satisfied in a property of resilience if $\varepsilon = 0.35$, and all versions of the `putty` program showed that it is similar in Table 2. Therefore, we can decide a threshold $\varepsilon = 0.35$. Once again, in order to satisfy the credibility property, we will describe the $\varepsilon = 0.35$ with our second evaluations in Sect. 6.2.

Our detailed results of resilience property are presented in Fig. 7. The graph shown in Fig. 7 is based on data from Table 2 and 1. For better expression of the graphs, a specific value was mapped to reveal the version-specific differences. For instance, as a reference, the 0.55 version has 0, the 0.56 version has +1, and the 0.57 version has +2. Accordingly, higher versions were cumulatively mapped at +1 each. Conversely, lower versions were mapped by deducting −1; for example, 0.54 and 0.53 were mapped at −1 and −2, respectively. The resulting values were used as x-coordinates. The y-coordinates were used as the measurements of similarity. The arithmetic mean of this reconfigured data was computed and expressed in a black curve. Each point indicates a similarity value for each version. By using this graph, we prove that the similarity decreases as a version is farther from itself ($x = 0$). We additionally confirm that the proposed system satisfies the resiliency.

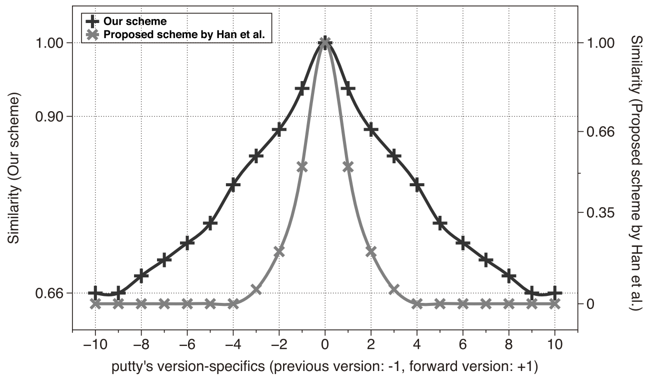


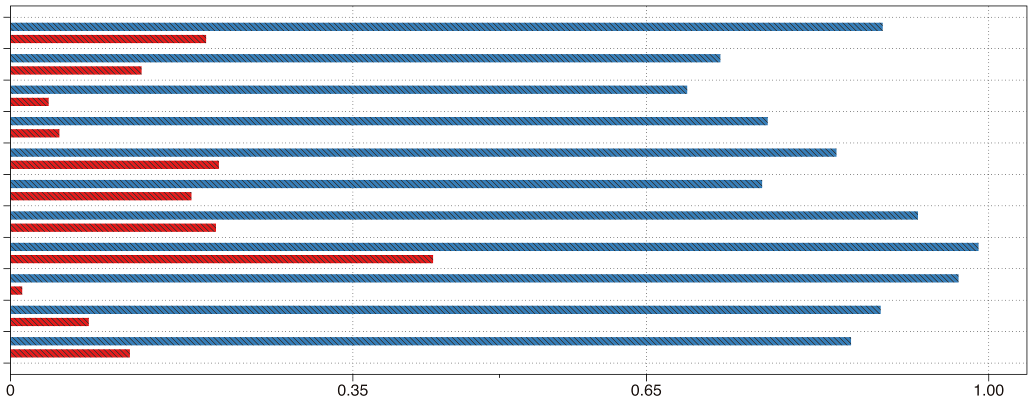
Fig. 7 Comparison between the procedure-based API sequence and entire API sequence

6.2 Evaluation of Other Applications

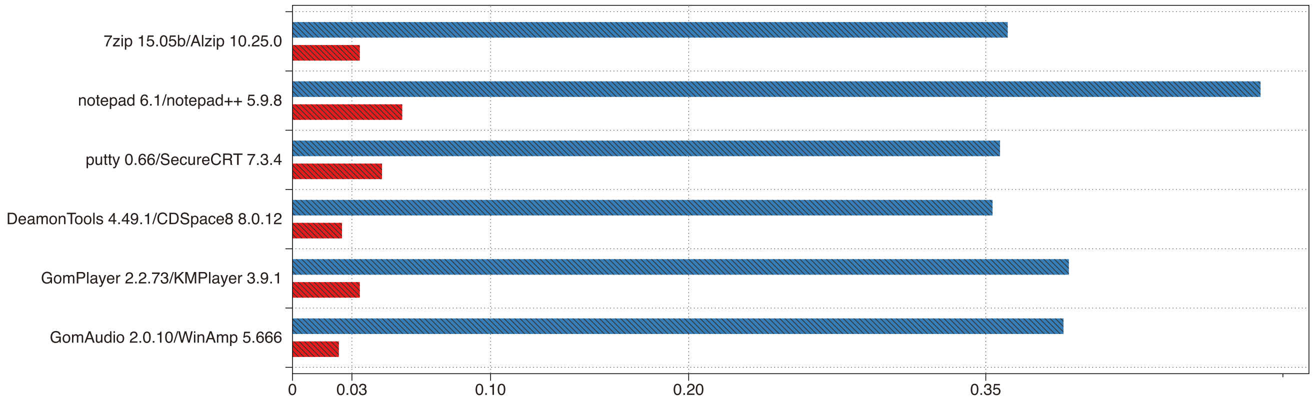
Our second experiment was performed on applications that were independently implemented. As shown in Table 4, the sample applications were selected by classifying them into particular categories. Some applications with two different versions were selected for the similarity measure. All applications had the PE format that were executable in Windows, and they were compiled for a 32-bit operating system.

In Table 4, we also measured the reduction ratio between sizes of the original birthmark (including only the API call sequences) and birthmark applied fuzzy hashing. The reduction ratio is about 41% on average. This means that the size of the birthmark is actually more efficient at using fuzzy hashing than the original birthmark. Also, it can be decreased in a storage overhead on the database.

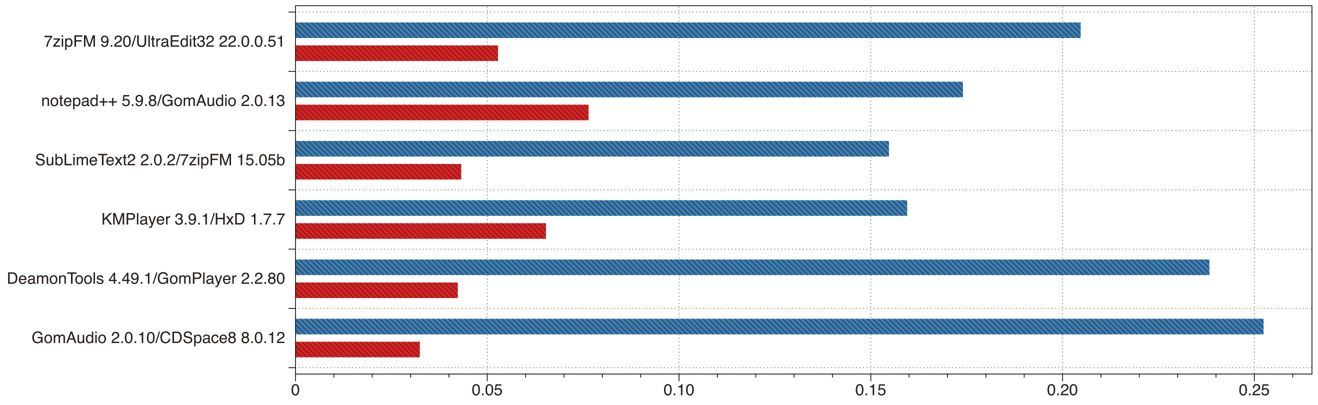
To compare with the method proposed by Han et al.,



(a) Similarities between the same applications with different versions.



(b) Similarities between the applications in same category.



(c) Similarities between applications in different categories.

Fig. 8 Several cases of similarities between same-category and different-category applications

we experimented with various Windows applications. In Fig. 8, the blue bar and red bar represent the similarities of the our API-based birthmark using fuzzy hashing and the method proposed by [28], respectively. Among these, Fig. 8(a) shows a graph that is compared by same applications with different versions.

By deciding the threshold $\varepsilon = 0.35$ in Sect. 6.1, $1 - \varepsilon$ is 0.65. Since all the blue bars have higher values than $1 - \varepsilon$, our method also satisfy the resilience property. In Fig. 8, the results were performed with the same applications and identical approach, using the API-based approach. Thus, the other method, proposed by Han et al., also must have a similar value as our method. Nonetheless, their results were not fully satisfied in term of the resilience property, and all the values are distinguished from our values. Such an issue is appeared by each different extraction method.

Fundamentally, the organizational unit of a native code (i.e. compiled from original source code) is the procedure,

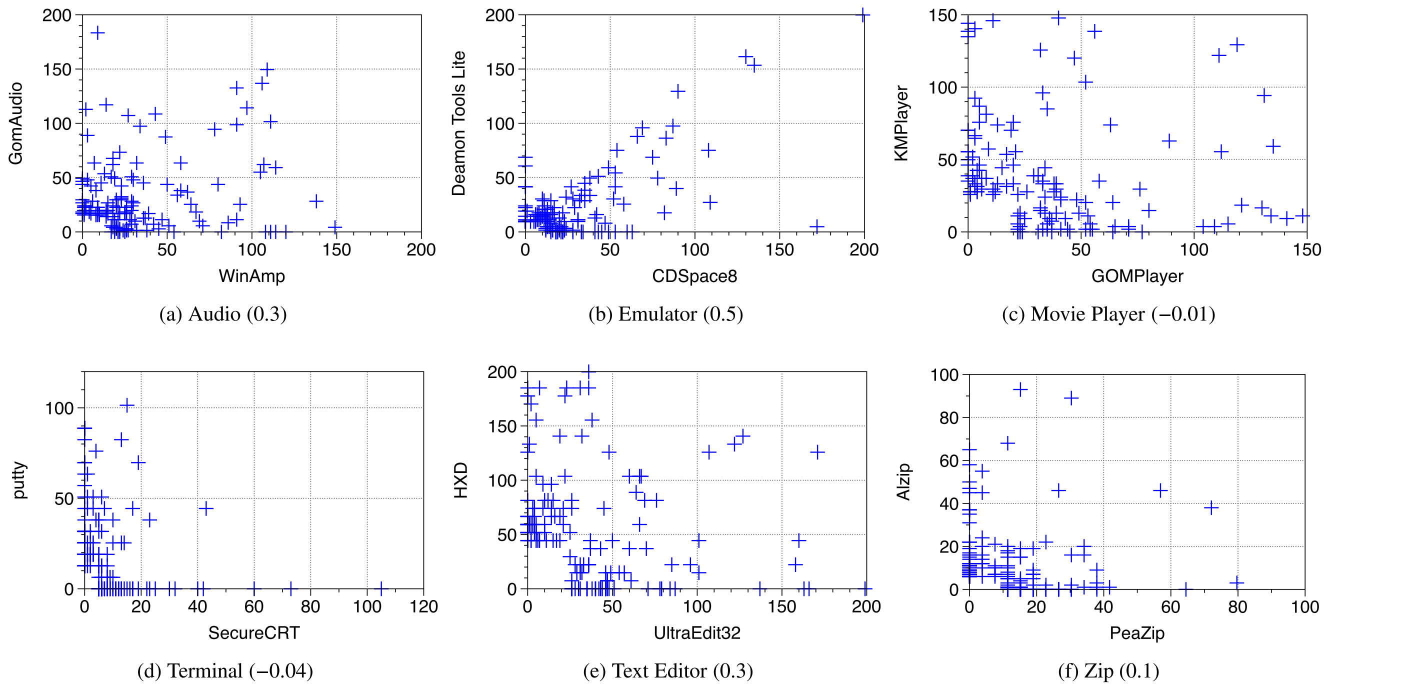


Fig. 9 Correlation analysis of the top 50 API call frequency between same-category programs

also called the function. We extracted the birthmark from the API call sequence out of precisely segmented procedures. By contrast, the order of the API call sequence that is extracted from the entire code section can be changed by the procedure position, when recompiling with same original source code. The distinction between these methods, which is involved in the characteristics of a program application, was clearly revealed from our experiments.

In Fig. 8(b) and Fig. 8(c), to prove the credible property, we also compared the applications in the same categories or different category. Even if two applications are in the same category, the similarity is not always high, because the applications have different numbers of procedures and API calls. However, our method fully satisfied the inconclusive, which is $\varepsilon < \alpha \leq 1 - \varepsilon$, where α is $\mathbf{Sim}_p(\mathfrak{B}_p(\mathcal{P}), \mathfrak{B}_p(\mathcal{Q})) \rightarrow \alpha$.

In this paper, we would like to clearly elucidate this point. Applications of the same category satisfied "inconclusive" in the experiment. However, this does not mean that all applications in the same category had the same outcome. What we would like to make clear about this issue is that even in applications that belong to the same category may have a high probability of different API call sequences due to functional characteristics of each application despite similarity between applications; therefore, the applications can be perceived as completely independent of one another. For example, let us assume that there are two music player programs and the first program has simply a playback function of mp3 files only and the second program can not only playback high quality sound music files (such as flac or dsd) format but also has a visual function. Then, even if the above two programs belong to the same category according to our method, birthmark similarity is considerably low. The reason for this result is because the type of similar applications is not measured but similarity is measured by focusing on reflecting unique functional characteristics due to the birthmark characteristics.

Nonetheless, evaluation results on similarity using their method, which are compared in Fig. 8(b), are significantly low. As shown in Fig. 8(c), their experiments results using applications of totally different categories had also significantly low similarity. The overall reason for the low similarity result of their measurement is due to the fact that the API call sequence is extracted from the entire code section as mentioned in the above. When a location of procedure in the API calls is modified within a program due to reasons such as recompiling, their method is likely to determine the programs as independent programs even if the programs are similar to one another. Thus, our experiment results showed that their measurement values are maintained significantly lower than ours overall. Furthermore, even if threshold ε is set to 0.4 in accordance with their experiment result shown as Table 1, any results in Fig. 8 cannot satisfy the attributes such as credibility and resilience.

6.3 Correlation Analysis for Pre-Filtering with DLLs/APIs

Han et al. performed pre-filtering with the numbers of DLLs and APIs of the given program. Windows enables a variety of applications to be available by loading APIs to a shared memory in a DLL-encapsulated module. As mentioned earlier, the functional classification of DLLs requires a large number of separated DLLs to remain in memory all the time. For this reason, Windows reduces the number of DLLs by separating them into particular classes to make them remain in memory. Hence, as also mentioned earlier, the application characteristics cannot be sorted only with names that are not clearly classified by function. Moreover, because many multifunctional applications are currently available, it has becomes difficult to determine the characteristics of applications simply by the API call frequency alone.

This problem has already been addressed in Sect. 3. To investigate this issue, a correlation analysis was conducted between programs in the same category using the top 50 API call frequencies, as shown in Fig. 9. Each graph indicates the correlation coefficient regarding the API call occurrence frequencies of the two programs in each category for the sample programs. The correlation coefficient value for each category was rated very low, which indicates that the numbers of DLLs and APIs and their names were not sufficient for category classification as proposed by Han et al. To achieve the category classification, a stochastic approach, such as machine learning, is required. Therefore, related research will be conducted in our future work.

7. Discussion and Future Works

In this paper, we proposed a birthmarking system using fuzzy hashing for API call sequences. Our proposed method improves the most recent API-based software birthmarking method, which was proposed by Han et al. [28]. Software birthmarks have conventionally been studied to satisfy two properties, resiliency and credibility. In the experiments conducted, resiliency was evaluated by comparing different versions of the putty program, while the problems of the Han et al. approach were identified. Based on the results, we conclude that our proposed birthmarking system is superior to that of Han et al. In addition, we conducted a credibility evaluation experiment in which 18 applications were selected from each category. This involved similarity measures, including different versions of the same program. The results suggest that an exact distinction is possible.

In this paper, we highlighted problems associated with category classification of the Han et al. approach. A correlation analysis was performed across programs of the same category with the top 50 APIs and their frequencies in each program's API call frequency. In Fig. 9, the results showed that all programs had low correlations. Therefore, we conclude that the functional category classification cannot be achieved with the Han et al. approach. Instead, a stochastic approach, such as machine learning, is needed and will be proposed in our subsequent paper.

In recent years, owing to technological advances, various types of computer-related crimes have been increasing. Specialized digital forensic technologies have therefore been in high demand to collect and analyze evidence. Our proposed birthmarking method can determine similarities between software programs. Determination of similarity measures between software programs can be utilized in detecting rapidly evolving malware variants. Antivirus programs conventionally relying on signatures and thus cannot successfully address unknown or variant malware. In contrast, the birthmarking technique can be used in malware detection as well as in detecting software piracy, code theft, and copyright infringement. Therefore, our future research will extend the present paper and develop a birthmark technique generated by the combination of a program's other inherent information as well as API calls. We will then apply the technique to unknown and variant malware detection.

8. Conclusion

Thesoftware birthmark is a technology that reflects the in herent characters of application programs. It has conventionally been studied in fields such as software piracy, code theft, and copyright infringement. This technique can be classified into three approaches: instruction-based, structure-based, and API-based approaches. The API-based approach can reflect well on the characteristics of software because the API is used by all applications that must perform certain functions (such as the I/O, network, and GUI) of the operating system. In this paper, we proposed an API-based software birthmarking method using fuzzy hashing. Our proposed software birthmarking technique extracts API call sequences in the segmented procedures for the native code of a program and then generates them using a fuzzy hash function. The fuzzy hash, unlike the conventional cryptographic hash function, is used for the similarity measurement of data. Our method using fuzzy hash function achieved a high reduction ratio (about 41% on average) more than the original birthmark that is generated only with the API call sequences. In our experiments, when threshold ε is 0.35, the results showed that our method is an effective birthmarking system to measure similarities of the software. The threshold $\varepsilon = 0.35$ was obtained from the experimental results: a similarity measure for an application version-specific and the various applications in the same category or different category. Moreover, our correlation analysis with top 50 API call frequencies proved that it is difficult to functionally categorize applications using only DLL/API numbers/names. Compared to prior work, our method significantly improved the properties of resilience and credibility. Our future work will address unknown or variant malware detection as an extension of this proposed birthmarking technique.

References

[1] ARXAN, "A look inside the universe of pirated software and digital assets," tech. rep., 2015 4th Annual State of Application Security Report, 2015.
[2] J.F. Gantz, et al., "The dangerous world of counterfeit and pirated software," tech. rep., IDC White Paper, March 2013.
[3] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp.311–324, ACM, 1999.
[4] W. Zhu, C. Thomborson, and F.Y. Wang, "A survey of software watermarking," in Intelligence and Security Informatics, pp.454–458, Springer, 2005.

[5] D. Aucsmith, "Tamper resistant software: An implementation," Information Hiding, pp.317–333, Springer, 1996.

[6] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[7] C.S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," Software Engineering, IEEE Transactions on, vol.28, no.8, pp.735–746, 2002.

[8] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in Information security, pp.404–415, Springer, 2004.

[9] G. Myles and C. Collberg, "K-gram based software birthmarks," Proc. 2005 ACM symposium on Applied computing, pp.314–318, ACM, 2005.

[10] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," Proc. twenty-second IEEE/ACM international conference on Automated software engineering, pp.274–283, ACM, 2007.

[11] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.i. Matsumoto, "Dynamic software birthmarks to detect the theft of windows applications," International Symposium on Future Software Technology, 2004.

[12] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," ACM Sigplan Notices, pp.107–118, ACM, 2004.

[13] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–a tool for software protection research," IEEE security & privacy, no.4, pp.40–49, 2003.

[14] S.L. Garfinkel, "Digital forensics research: The next 10 years," digital investigation, vol.7, pp.S64–S73, 2010.

[15] G. Palmer et al., "A road map for digital forensic research," First Digital Forensic Research Workshop, Utica, New York, pp.27–30, 2001.

[16] K. Gregory, "Managed, unmanaged, native: What kind of code is this?," 4 2003.

[17] B. Abrams, "What is managed code?," blog, Microsoft Windows Dev Center, Jan. 2004.

[18] S. Cesare and Y. Xiang, Software similarity and classification, Springer Science & Business Media, 2012.

[19] H.i. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of java programs through analysis of the control flow information," Information and Software Technology, vol.51, no.9, pp.1338–1350, 2009.

[20] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A software birthmark based on dynamic opcode n-gram," Semantic Computing, 2007. ICSC 2007. International Conference on, pp.37–44, IEEE, 2007.

[21] D. Lee, Y. Choi, J. Jung, J. Kim, and D. Won, "An efficient categorization of the instructions based on binary excutables for dynamic software birthmark," Int. J. Information and Education Technology, vol.5, no.8, pp.571–576, 2015.

[22] H.-I. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of java applications via a static birthmark based on weighted stack patterns," IEICE Trans. Inf. & Syst., vol.E91-D, no.9, pp.2323–2332, 2008.

[23] J. Nagra and C. Collberg, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, Pearson Education, 2009.

[24] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Detecting the theft of programs using birthmarks," Information Science Technical Report NAIST-IS-TR2003014 ISSN, 0919–9527, 2003.

[25] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.i. Matsumoto, "Design and evaluation of dynamic software birthmarks based on api calls," Info. Science Technical Report NAIST-IS-TR2007011, ISSN, pp.0919–9527, 2007.

[26] H. Park, S. Choi, H.i. Lim, and T. Han, "Detecting java theft based on static api trace birthmark," in Advances in Information and Computer Security, pp.121–135, Springer, 2008.

[27] S. Choi, H. Park, H.i. Lim, and T. Han, "A static api birthmark for windows binary executables," Journal of Systems and Software, vol.82, no.5, pp.862–873, 2009.

[28] Y. Han, J. Choi, S.j. Cho, H. Yoo, J. Woo, Y. Nah, and M. Park, "A New Detection Scheme of Software Copyright Infringement using Software Birthmark on Windows Systems," COMPUTER SCIENCE AND INFORMATION SYSTEMS, vol.11, no.3, SI, pp.1055–1069, Aug. 2014.

[29] G. Myles, "Software theft detection through program identification," 2006.

[30] N. Harbour, Dcfldd, 2002.

[31] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," Digital investigation, vol.3, pp.91–97, 2006.

[32] J. Kornblum, "Fuzzy hashing and ssdeep," 2006.

[33] V. Roussev, G.G. Richard, and L. Marziale, "Multi-resolution similarity hashing," digital investigation, vol.4, pp.105–113, 2007.

[34] jcanto, "Extra metadata field: ssdeep," 2008.

[35] D. Hurlbut, "Fuzzy hashing for digital forensic investigators," AccessData, 2009.

[36] D. French, "Fuzzy hashing against different types of malware," tech. rep., 2010 CERT Research Report, Software Engineering Institute, Sept. 2010.

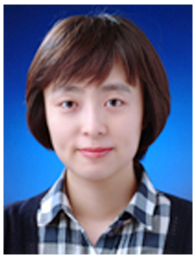
[37] R. Rivest, "The md5 message-digest algorithm," 1992.

[38] T. Andrew, "Spamsum readme," 2011.

Donghoon Lee received the B.S. degree in Computer Science from National Institute for Lifelong Education (NILE), Korea, in 2009 and the M.S. degree in Information Security Engineering from Sungkyunkwan University, Korea, in 2011. He is currently undertaking a Ph.D. course on Electrical and Computer Engineering in Sungkyunkwan University. He also worked as a security developer in EGLOO SECURITY and NEXON COMPANY between 2010 and 2013. His current research interest is in the area of software security, cryptography, authentication protocol, and network security.

Dongwoo Kang was born in Seoul, Korea on January 26, 1993. He received the B.S. degree in Electrical and Computer Engineering from Sungkyunkwan University, Korea, in 2015. He is currently pursuing M.S. degreein Electrical and Computer Engineering at Sungkyunkwan University. His current research interest includes cryptography, malware, and authentication or key management protocols.

Younsung Choi received the B.S. degree in Electrical and Computer Engineering from Sungkyunkwan University, Korea, in 2006 and the M.S. degree in Electrical and Computer Engineering from Sungkyunkwan University, Korea, in 2007. He is currently undertaking a Ph.D. course on Electrical and Computer Engineering in Sungkyunkwan University. His current research interest is in the area of digital forensic, cyber crime, cryptography, authentication protocol, and network security.



Jiye Kim received the B.S. degree in Information Engineering from Sungkyunkwan University, Korea, in 1999 and the M.S. degree in Computer Science Education from Ehwa University, Korea, in 2007. He is currentlyundertaking a Ph.D. course on Electrical and Computer Engineering in Sungkyunkwan University. His current research interest is in the area of cryptography, authentication protocol, and sensor security.

Dongho Won received his B.E., M.E., and Ph.D. from Sungkyunkwan University in 1976, 1978, and 1988, respectively. After working at ETRI (Electronics and Telecommunications Research Institute) from 1978 to 1980, he joined Sungkyunkwan University in 1982, where he is currently Professor of the School of Information and Communication Engineering. In the year 2002, he served as the President of KIISC (Korea Institute of Information Security and Cryptology). He was the Program Committee Chairman of the 8th International Conference on Information Security and Cryptology (ICISC 2005). His research interests are on cryptology and informationsecurity.