Replacement Attacks: Automatically Impeding Behavior-based Malware Specifications

Jiang Ming[1], Zhi Xin[2], Pengwei Lan[1], Dinghao Wu[1], Peng Liu[1], and Bing Mao[2]

[1] The Pennsylvania State University, University Park, PA 16802, U.S.A.
{jum310, pul139, dwu, pliu}@ist.psu.edu
[2] Nanjing University, Nanjing 210093, China
{zxin, maobing}@nju.edu.cn

Abstract. As the underground market of malware flourishes, there is an exponential increase in the number and diversity of malware. A crucial question in malware analysis research is how to define malware specifications or signatures that faithfully describe similar malicious intent and clearly stand out from other programs. It is evident that the classical syntactic signatures are insufficient to defeat state-of-the art malware. Behavior-based specifications which capture real malicious characteristics during runtime, have become more prevalent in anti-malware tasks, such as malware detection and malware clustering. This kind of specification is typically extracted from system call dependence graphs that a malware sample invokes. In this paper we present *replacement attacks* to poison behavior-based specifications by concealing similar behaviors among malware variants. The essence of the attacks is to replace a behavior specification to its semantically equivalent one, so that similar malware variants within one family turn out to be different. As a result, malware analysts have to put more efforts to re-analyze similar samples. We distill general attacking strategies by mining more than $5,000$ malware samples' behavior specifications and implement a compiler-level prototype to automate replacement attacks. Experiments on 960 real malware samples demonstrate effectiveness of our approach to impede multiple malware analyses based on behavior specifications, such as similarity comparison and malware clustering. In the end, we provide possible counter-measures to strengthen behavior-based malware analysis.

1 Introduction

Malware, or malicious software with harmful intent to compromise computer systems, is one of the major challenges to the Internet. Over the past years, the ecosystem of malware has evolved dramatically from “for-fun” activities to a profit-driven underground market [3], where malware developers sell their products and cyber-criminals can simply purchase access to tens of thousands of malware-infected hosts for nefarious purposes [1]. Normally malware developers do not write new code from scratch, but choose to update old code with new features or obfuscation methods [23]. With thousands of malware instances

appearing every day, efficiently processing large quantity of malware samples which exhibit similar behavior, has become increasingly important. A key step to improve efficiency is to define discriminative specifications or signatures that faithfully describe intrinsic malicious intents, so that malware samples with similar functionalities tend to share common specifications. Malware analysts benefit from general specifications. For example, every time a suspicious program is found in the wild, malware analysts can quickly determine whether it belongs to a previous known family by matching its specification.

As malware keeps evolving to evade detection, the classical syntactic specifications are insufficient to defeat various obfuscation techniques, such as polymorphism [21], binary packing [31] and self-modifying code [12]. In contrast, behavior-based specifications, which are generated during malware execution, are more resilient to static obfuscation methods and able to disclose the natural behavior of malware, such as replication, download and execution and remote injection. The main means for malware to interact with an operating system is through system calls[1]. The dataflow dependencies among system calls are expressed as an acyclic graph, namely system calls dependency graph (SCDG), where nodes represent system calls executed and a directed edge indicates a data flow between two nodes. Typically, the dependencies derive from the return value or the arguments computed by previous system calls. When a data source is passed to one of its succeeding native APIs, a directed edge connecting these two nodes is created. Since data flow dependencies are hard to be reordered, SCDG has been broadly accepted as a reliable abstraction of malware behavior [15, 18], and widely employed in malware detection [6, 20] and malware scalable clustering [7, 28].

With quite a number of compelling applications, SCDG looks promising. However, it is not impossible to circumvent. In order to inspire more state-of-the-art malware analysis techniques, we exploit the limitations of the current approaches and present *replacement attacks* against malware behavior specifications. We show that *it is possible to automatically conceal similar behavior specifications among malware variants by replacing a SCDG to its semantically equivalent one, so that similar malware variants show large distances and therefore are assigned to different families.* Eventually, malware analysts have to re-analyze large number of malware samples exhibiting similar functionalities. To achieve this goal, we first mine two large data sets to identify popular system calls and OS objects dependencies. We summarize two general attacking strategies to replace SCDG: 1) mutate a sequence of dependent system calls (sub-SCDG) to its equivalent ones, and 2) insert redundant data flow dependent system calls. Our approach ensures that the new generating dependence relationships are so common that they cannot be easily recognized. After transformation, similar malware samples reveal large distance when they are measured with widely used similarity metrics, such as graph edit distance [13] or Jaccard Index [11]. As a result, subsequent analyses (e.g., malware detection and clustering) are misled.

To demonstrate the feasibility of replacement attacks, we have developed a compiler-level prototype, *API Replacer*, to automatically perform transforma-

[1] The systems call in Windows NT is called as native API

tion on top of the LLVM framework [22] and Microsoft Visual Studio. Given a single malware source code, API Replacer is able to generate multiple malware binaries, and each one exhibits different behavior specifications. We evaluate our replacement attacks on a variety of real malware samples with different replacement ratio. Our experimental result shows that our approach successfully impede malware similarity comparison and state-of-the-art behavior-based malware clustering. The cost of transformation is low and the execution overhead after transformation is moderate.

In summary, we make the following contributions:

- We propose replacement attacks to camouflage similar behavior specifications among malware variants by replacing system call dependence graphs.
- We summarize the rules for equivalent replacements by mining large set of malware samples. The distilled attacking strategies tangle structure of system call dependency as well as behavior feature set without affecting semantics.
- We automate replacement attacks by developing a compiler-level prototype to perform source to binary transformation. The experimental results demonstrate our approach is effective.
- To the best of our knowledge, we are the first one to demonstrate the feasibility of automatically obfuscating behavior based malware clustering on real malware samples.

The rest of the paper is organized as follows. Section 2 introduces previous work on behavior based malware analysis. Section 3 describes in detail about how to generate replacement attacks rules with a case study. Section 4 highlights some of our implementation choices. We present the evaluation of our approach in Section 5. Possible counter-measures are discussed in Section 6 and we conclude the paper in Section 7.

2 Related Work

In this section we first present previous work on behavior based malware analysis, which is related to our work in that their methods rely on system call sequences or graphs that a malware sample invokes. Then, we introduce previous research on impeding malware dynamic analysis. In principle, our approach belongs to this category. At last we describe related work on system call API obfuscation, which is close in spirit to our approach.

Behavior based malware analysis Malware dynamic analysis techniques are characterized by analyzing the actual executing instructions of a program or the effects that this program brings to the operating system. Compared with static technique, dynamic analysis is less vulnerable to various code obfuscation [26]. Christodorescu et al. [15] introduce malware specifications on data-flow dependencies among system calls, which capture true relationships between system calls and are hard to be circumvented by random system call injection. Since

then, such malware specifications based on SCDG have been widely used in malware analysis tasks, such as extracting malware discriminative feature by mining the difference between malware behavior and benign program behavior [18], determining malware family in which instances share common functionalities [7, 6, 28], and detecting malicious behavior [8, 20, 25]. However, none of the presented approaches is explicitly designed to be resilient to our replacement attacks.

Anti-malware behavior analysis Some countermeasures have been proposed to evade behavior based malware analysis. Since malware behavior analysis is typically performed in a controlled sandbox environment, the lion's share of previous work focus on run time environment detection [14, 27]. If a malware sample detects itself running in a sandbox rather than real physical machine, it will not carry out any malicious behaviors. To defeat environment-sensitive malware, Dinaburg et al. [17] build a transparent analysis platform, which remains invisible to such sandbox environment check. Another direction relies on contrasting different executions of a malware sample when running in multiple sandboxes. The control flow deviations may indicate evasion attempts [19]. Our method does not detect sandbox and is valid in any run time environment. Our replacement attacks shares similar idea to subvert malware clustering with recent work [9, 10]. Our work is different from these previous works in that we attempt to obfuscate data flow dependencies between system calls, while the behavior features these works attack contain no data flow dependencies. As data relationships between behavior features are hard to be affected by random noise insertion, our attacking method is more challenging. Furthermore, these work evaluated their attacks by directly manipulating malware behavior feature set instead of malware code, which means their attacks may not be feasible in practice. In contrast, to demonstrate the feasibility of replacement attacks, we develop a compiler-level converter to transform malware source code to binary.

System call obfuscation The original idea to obfuscate system call API can be traced to *mimicry attack* against intrusion detection [35]. *Illusion* [34] allows user-level malware to invoke kernel operations without calling the corresponding system calls. To launch the *Illusion* attack, the attacker has to install a malicious kernel module, which is not practical in many real attacking scenarios. Ma et al. [24] present *shadow attacks* by partitioning a malware sample into multiple shadow precesses and each shadow process presents no-recognizable malware behavior. But it's still an open question to launch a multi-process malware sample covertly. Our proposed attack is inspired by Xin et al. [37]'s approach to subvert behavior based software birthmark. However, their attacking method is restricted to replacing a dependency edge with a new vertex and two new edges. As shown in Section 5.2, this simple attacking method only has limited effect on reducing Jaccard Index. In contrast, our approach provides multiple attacking strategies. In addition, Xin et al. [37]'s attack code is pre-loaded as a dynamic library when the program starts running. The drawback is it's quite easy to detect such library interruption. Our *API Replacer* embeds newly added system calls into the native code transparently, so that our approach has better stealth.

3 Replacement Attacks Design

3.1 Overveiw

In spite of various metamorphic or polymorphic obfuscation, malware samples within the same family tend to reveal similar malicious behavior [23]. Our goal in this paper is to separate similar malware variants by replacing SCDG, the most prevalent expression to represent malware behavior specifications. Fig. 1 shows an example of SCDG before/after replacement attacks. At the top of Fig. 1, we list pseudo code fragment written in MSVC for ease of understanding. In the original SCDG, the return value of "NtCreateFile" is a FileHandle ("hFile1"), denoting the new created file object. As hFile1 is passed to "NtClose", a data flow dependency connects "NtCreateFile $\rightarrow$ NtClose". Windows API "SetFile-Pointer" in the new code moves the file pointer and returns new position, which is quite similar to "lseek" system call in Unix. The return value of "SetFile-Pointer" is equal to moving distance plus the offset of starting point, which is 0 ("FILE_BEGIN") in this example. We exploit the fact that the data type of "hFile1" and the distance to move are both unsigned integers, and deliberately assign the distance to move with the same value of "hFile1" (line 2 in the new code). As a result, the return value of "SetFilePointer" ("dwFilePosition"), is equal to the "hFile1". Then "dwFilePosition" is passed to "NtClose" to close the file. When calling "SetFilePointer", native API "NtSetInformationFile" is invoked to change the position information of the file object represented by "h-File1". In this way, the new code still preserves the original data flow, while the SCDG changes significantly. Note that compared with the original code, the file object is updated with new position information. However, the file object is closed immediately, imposing no lasting side effect to the final state.

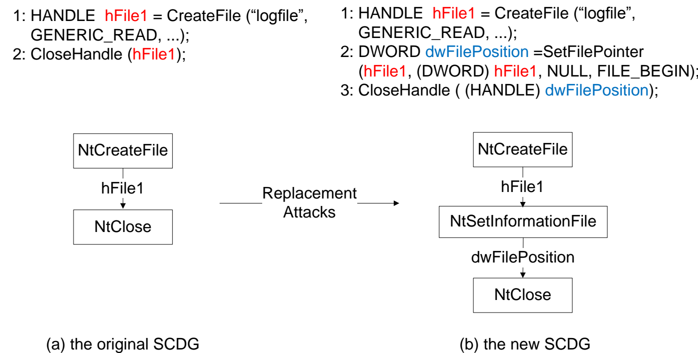


Fig. 1. An example of SCDG before and after replacement attacks

A typical scenario to apply replacement attacks is illustrated in Fig. 2. Taking malware source code as input to API Replacer, our compiler-level transformation tool, malware authors generate multiple binary mutations of the initial version.

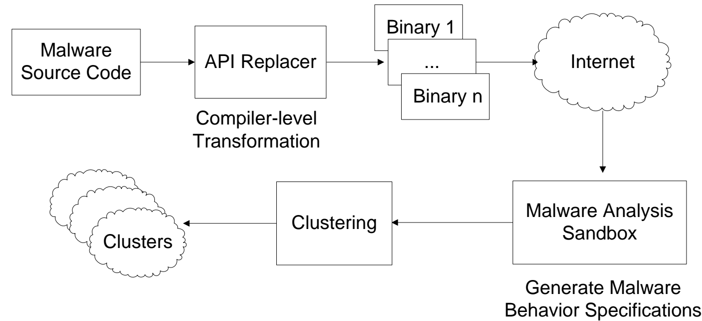


Fig. 2. Illustration of replacement attacks

Each mutation shares similar malicious functionalities, but exhibits different behavior specifications. Then cyber-criminals spread these malware samples to the Internet or plant them in the live vulnerable hosts. Suppose these transformed malware samples, with other suspicious binaries are finally collected by anti-malware companies. To process large number of malware samples, anti-malware companies utilize automated clustering tools to identify samples with similar behavior. These tools execute malware instances in a sandbox and collect run time information to generate behavior specifications, which will be normalized and then fed to clustering algorithm. As we mentioned in Section 2, current malware clustering tools are not designed to explicitly resist replacement attacks, therefore similar malware mutations after replacement attacks are probably assigned to different clusters. In that case, malware analysts have to waste excessive efforts to re-analyze these similar samples.

3.2 Mining Two Large Data Sets

Since there are various expressions of malware behavior based on SCDG, to find out the possible targets we may attack, we first mine two large data sets of malware behavior specifications used for malware detection and clustering.

- BRS-data [6] is used by Babić et al. to evaluate malware detection with tree automata inference. BRS-data contains system calls dependency graphs generated for 2631 malware samples and covers a large variety of malware, such as trojan, backdoor, worm, and virus.
- BCHKK-data [7] is used for evaluating malware clustering technique proposed by Bayer et al. BCHKK-data includes behavior profiles extracted from 2658 malware samples, and more than 75% samples are the variants of Allaple worm. Note that SCDG is not amenable to scalable clustering techniques, which usually operate on numerical vectorial feature set. Bayer et al. converted system call dependencies to a set of features in terms of operations (create, read, write, map, etc.) on OS objects (file, registry, process, section, thread, etc.) and dependencies between OS objects.

These two data sets reflect two typical applications of SCDG to represent malware specifications: 1) directly utilize rich structural information contained in SCDG [15, 29, 18], which is able to match behavioral patterns exactly but lacks of scalability; 2) extract higher level abstractions from SCDG to fit for efficient large-scale malware analysis [7, 8, 30] at the cost of precision. The similarity of BRS-data is normally measured by graph edit distance or graph isomorphism [13], while the similarity metrics of BCHKK-data is calculated by Jaccard Index [11].

Popular dependencies We calculate popular native API dependencies from BRS-data and OS operations and dependencies from BCHKK-data. Table 1 lists 11 popular native API dependencies out of BRS-data, which are mainly related to the operations on Windows registry, memory and file system. The second column is the medium data flow types passed between system calls. Most of the medium types are handles, which stands for various OS objects such as file, registry, section (memory-mapped file), process, etc. Table 2 presents popular OS object types, operations and dependencies from BCHKK-data. We believe as long as we diversify these popular dependencies and behavior features, the similarity among malware mutations can drop significantly.

Common sub-SCDGs Although extracted from different sources, these data reveal some common malicious functions, which are mapped to sub-SCDGs. The top 3 popular sub-SCDGs are corresponding to malware replication, registry modification for persistence and code remote injection. For example, the several frequent dependencies regarding "NtMapViewOfSection" and OS objects dependency between file and section, indicate malware writers commonly utilize memory mapped file to facilitate file manipulation. Malware often configure Windows registry for persistence in order to run automatically when machine starts, leading to frequent operations on Windows registry. "NtOpenProcess $\rightarrow$ NtWriteVirtualMemory" and "process $\rightarrow$ thread" are mainly introduced by creating a new thread in a remote process, the most common way to launch malware covertly in vulnerable hosts [33]. If we implement these common functions through different ways, the corresponding sub-SCDGs can be changed drastically as well.

3.3 Attacking Strategies

In this section we elaborate how to construct replacement attacks strategies. We propose 3 requirements that our attacking strategies have to meet:

1. (R1) Our replacement attacks should invalidate various malware behavior similarity metrics, such as graph edit distance and Jaccard Index.
2. (R2) New system calls and dependencies impose no side effect to original data flow.
3. (R3) Transformed SCDG should be as common as possible.

Table 1. Popular windows native API dependencies

Dependencies	Data flow types	Ratio (%)
NtMapViewOfSection → NtProtectVirtualMemory	void *address	22.4
NtOpenKey → NtQueryValueKey	KeyHandle	19.4
NtCreateSection → NtMapViewOfSection	SectionHandle	9.6
NtMapViewOfSection → NtUnmapViewOfSection	void *address	8.9
NtOpenSection → NtMapViewOfSection	SectionHandle	6.3
NtCreateFile → NtReadFile	FileHandle	5.4
NtCreateSection → NtQuerySection	SectionHandle	4.8
NtOpenKey → NtQueryKey	KeyHandle	4.6
NtCreateFile → NtQueryInformationFile	FileHandle	4.2
NtOpenFile → NtSetInformationFile	FileHandle	4.1
NtOpenProcess → NtWriteVirtualMemory	ProcessHandle	3.8

Table 2. Popular OS object types, operations and dependencies

<table>
<tr><th>OS object type</th><th>OS operation</th></tr>
<tr><td>file</td><td>open, create, read, write, query_information,
query_directory, set_information, query_file</td></tr>
<tr><td>registry</td><td>create, open, query_value, set_value</td></tr>
<tr><td>section</td><td>query, create, map, open, mem_read</td></tr>
<tr><td>process</td><td>create, open, query</td></tr>
<tr><td>thread</td><td>create, query, resume</td></tr>
<tr><th colspan="2">OS object dependency</th></tr>
<tr><td colspan="2">file → file, registry → file, registry → registry,
process → thread, section → file, file → section</td></tr>
</table>

We meet our design requirement R1 by two attacking methods. The first one is embedding redundant data flow dependent system calls to replace original popular dependencies. As a result, new vertices and dependencies are created (see example in Fig. 1). At the same time, we make sure data types and values of original dependencies are preserved (satisfy R2). Further more, we observe that malicious functionalities can be developed with different technical methods, making it possible for SCDG mutations without undermining the intended purpose. For example, malware replication can be implemented through either memory-mapped file or file I/O; multiple ways exist to modify registry for the purpose of persistence. Therefore our second attacking strategy is transforming a sub-SCDG to its semantically equivalent mutations (satisfy R2). As a result, the original dependencies probably do not exist anymore. A by-product of our mining result in Section 3.2 is that popular dependencies can also be served as possible candidates to be embedded in a SCDG, so that the new SCDG doesn't look unusual (satisfy R3). Note that these two attacking methods can seamlessly weave together to amplify each other's effect.

3.4 Replacement Attacks Arsenal

In this section we present the details of our replacement attacks arsenal. According to our attacking strategies, we classify them into 2 categories:

Inserting redundant dependencies We summarize attacks belong to this category based on the medium data flow types listed in Table 1.

1. "NtSetInformationFile" attack. This attack can replace the dependencies with FileHandle as medium, which has been illustrated in Fig. 1.
2. "NtDuplicateObject" attack. "NtDuplicateObject" returns a duplicated object handle, which refers to the same object as the original handle.
3. "NtQuery*" attack. There are several windows native APIs for querying information of kernel objects, such as "NtQueryAttributesFile", "NtQueryKey", "NtQueryInformationProcess" and "NtQueryInformationFile". All of these query APIs take certain object handle as one of input argument and output object information. No any modification is introduced to the kernel objects. Hence "NtQuery*" native APIs are good candidates for our replacement attacks. For example, we could insert "NtQueryInformationFile" into a popular *NtCreateFile* $\rightarrow$ *NtSetInformationFile* dependency, where the output of "NtQueryInformationFile" ("FileInformation") is passed to "NtSetInformationFile". The two new dependencies also appear frequently.
4. The medium of "void *address" shown in Table 1 receives address of a mapped memory. To handle this medium, we can insert "NtQueryVirtualMemory" or "NtReadVirtualMemory", which do not affect the mapped memory address.

Sub-SCDG mutations We present multiple implementation ways to achieve 3 common malicious sub tasks we observed in Section 3.2, and what's more, we make sure that each implementation reveals different sub-SCDG with others.

1. Replication. When malware authors call Windows API "CopyFile" to replicate malware sample from source to target file, it is actually achieved through memory mapped file. When a process maps a file into its virtual address space, reading and writing to the file is simply manipulating the mapped memory region, which produces OS objects dependencies between file and section. First we can choose to map either source or destination file to memory section. Another implementation is only through file I/O operations. For example, we can copy a file by calling "NtReadFile" and "NtWriteFile" instead of using memory as medium.
2. Modify registry for persistence. Malware often add entries into the registry to remain active in the event of a reboot. There are multiple registry keys that can be configured to load malware at startup. The reference [4] lists 23 registry keys are accessed during system start. We leverage these multiple choices to randomly pick up available registry keys to update.

3. Code remote injection. Malicious code can be injected into another running process so that the process could execute the malware unwittingly. To achieve this functionality, we can either inject the malicious code directly into a remote process, or put the code into a DLL and force the remote process to load it [33].

3.5 Case Study

For a better understanding of our replacement attacks, we provide a real case to mutate the replication behavior of `Worm.Win32.Hunatcha`. Fig. 3(a) shows a native API sequence fragment we collected from the initial version and the corresponding SCDG. The malware sample replicates the file "hunatcha.exe" to "ladygaga.mp3.exe" by first memory-mapping the source file and then writing the memory content to the destination file. Fig. 4(a) presents the feature set abstracted from Fig. 3(a) , following the definition of BCHKK-data [7]. The first 3 lines are operations (open, create, write, etc.) on OS objects (file, section). The fourth line is an OS dependency from section to destination file.

Table 3. Similarity metrics of 3 mutations

	a vs. a	a vs. b	a vs. c	b vs. c
Graph edit distance	0.0	0.71	0.60	0.71
Jaccard Index	1.0	0.14	0.33	0.27

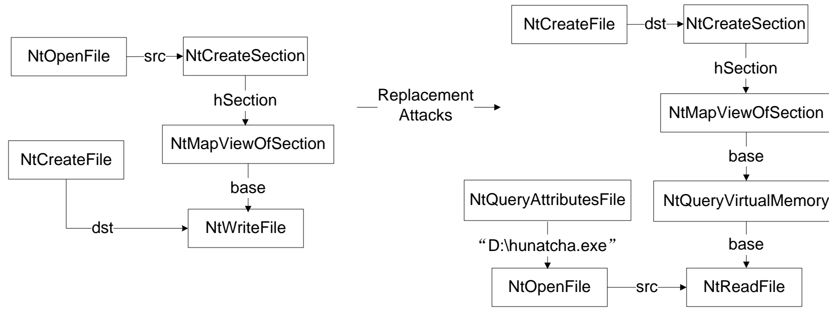
As shown in Fig. 3(b), we first mutate the generated SCDG by switching the file mapped to the memory, that is, we explicitly map the destination file (not source file) into the memory, so that file copying is achieved by reading content of source file to the mapped memory region. At the same time, we also insert redundant data flow dependent system calls to create new dependencies and decouple original dependencies. Therefore the structure of resulting SCDG and feature set (shown in Fig. 4(b)) are changed significantly. Fig. 3(c) presents another round attack. Instead of utilizing memory mapped file, we directly copy file through file I/O. Therefore no memory section appears in SCDG and feature set. Table 3 shows the two similarity metrics for these 3 mutations. The calculation of these two metrics is introduced in Section 5.2. The graph edit distance value of 0.0 or Jaccard Index value of 1.0 indicates that two behaviors are identical. The large graph edit distance or small Jaccard Index value means that after our replacement attacks, the similarity of malware variants drops substantially.

4 Implementation

To automate the attacking strategies we distill in Section 3, we have implemented a prototype tool, *API Replacer*, on top of LLVM and Microsoft Visual Studio 2012. Given an initial version of malware source code, API Replacer is able to

```
1: HANDLE src = NtOpenFile ("D:\hunatcha.exe", ⋯);
2: HANDLE dst = NtCreateFile
   ("\My Shared Folder\ladygaga.mp3.exe", ⋯);
3: HANDLE hSection= NtCreateSection(⋯, src);
4: void *base = NtMapViewOfSection (hSection, ⋯);
5: NtWriteFile (dst, base, length (src), ⋯ );
```

```
1: NtQueryAttributesFile ("D:\hunatcha.exe", ⋯);
2: HANDLE src = NtOpenFile ("D:\hunatcha.exe", ⋯);
3: HANDLE dst = NtCreateFile
   ("\My Shared Folder\ladygaga.mp3.exe", ⋯);
4: HANDLE hSection= NtCreateSection (⋯, dst);
5: void *base = NtMapViewOfSection (hSection, ⋯); 6:
NtQueryVirtualMemory (⋯, base, ⋯);
7: *base = NtReadFile (src, length (src) , ⋯);
```

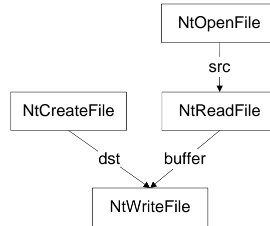


(a) the original SCDG

(b) the new SCDG

Replacement
Attacks

```
1: HANDLE src = NtOpenFile ("D:\hunatcha.exe", ⋯);
2: HANDLE dst = NtCreateFile
   ("\My Shared Folder\ladygaga.mp3.exe", ⋯);
3: void *buffer = NtReadFile (src, length (src) , ⋯ )
4: NtWriteFile (dst, buffer , length (src) ⋯ );
```



(c) the new SCDG

Fig. 3. System calls dependence graph (SCDG) of replication before and after replacement attacks

automatically generate multiple versions of malware binaries, which share similar malicious functionalities but exhibit different malware specifications. Fig. 5 describes the architecture of API Replacer. It takes malware source code as input and first generates LLVM IR through the Clang compiler. Then the IR code is manipulated by our transformation pass to fulfill replacement attacks.

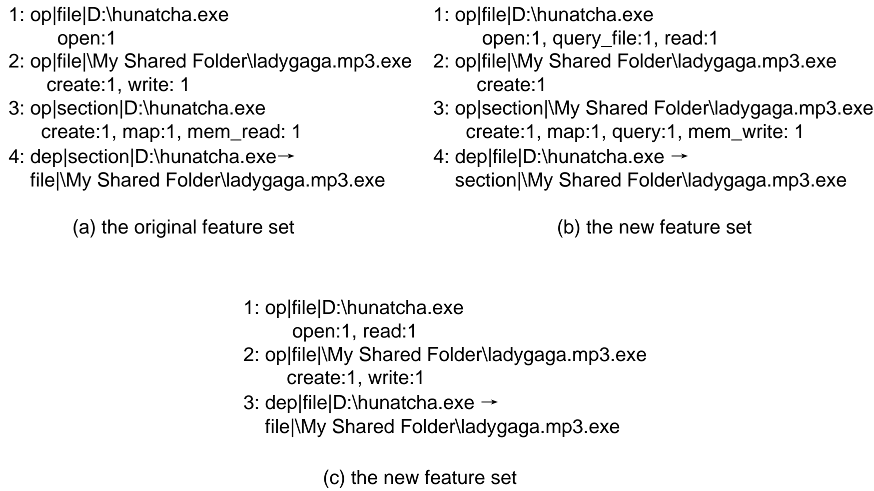


Fig. 4. Feature set of replication before and after replacement attacks

Afterwards the new transformed code are passed to LLC to emit object code, which are given to Visual Studio's link.exe to generate an executable binary. Moreover, new malware IR can be converted back to source code by LLC for another round of transformation. We follow the instructions in [2] to integrate LLVM system with Visual Studio. More specifically, our transformation pass inherits "CallGraphSCCPass" provided by LLVM to traverse the call graph and identify candidate system calls to attack. Our pass utilizes data flow analysis of LLVM to find out dependencies among system calls. Then two attacking strategies are performed in order to change the original SCDG. Section 3.3 describes these steps in details. After that, our pass updates the changes of call graph. Algorithm 1 lists each step of API Replacer's transformation pass.

The major implementation choice we made is using Windows APIs as a proxy for Windows native APIs. The reason is Windows native APIs are not comprehensively documented, while Windows APIs is well described in MSDN.[2] According to the mapping between Windows APIs and native APIs [32], we are able to manipulate Windows APIs directly.

Algorithm 1 API Replacer's algorithm

1: Traverse call graph
2: Identify candidate system calls and their dependencies
3: Mutate a sequence of dependent system calls to their equivalent ones
4: Insert redundant data flow dependent system calls
5: Update new call graph

[2] http://msdn.microsoft.com/

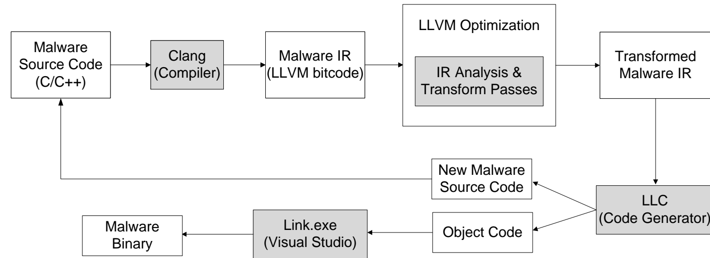


Fig. 5. The architecture of API Replacer

5 Evaluation

In this section, we apply API Replacer to transform real malware samples and evaluate the effectiveness of our approach to impede malware similarity metrics calculation and behavior-based malware clustering. We also test with 5 SPEC CPU2006 benchmarks to evaluate performance slowdown imposed by replacement attacks.

5.1 Experiment Setup

We transform malware source code collected from VX Heavens[3]. These malware samples are chosen for two reasons: 1) they do not contain any trigger-based behavior [36] or runtime environment checking condition [19]; 2) they have different malicious functionalities. In this way, we ensure that each sample fully exhibits its specific malicious intent during runtime execution and each sample presents different behavior specifications. Malware samples under experiment are executed in a malware dynamic analysis system, Cuckoo Sandbox[4], to collect windows native API calls traces. We first filter out isolated nodes which have no dependencies with others. Then we compute SCDG for each sample following the data flow dependencies between native APIs. Statistics for lines of code and SCDG are shown in Table 4.

5.2 Subverting Malware Behavior Similarity Metrics

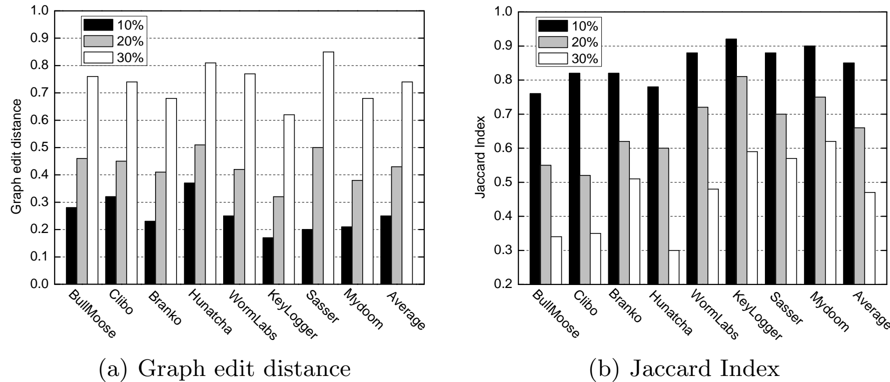
In this experiment, we evaluate replacement attacks with two representative similarity metrics, namely graph edit distance and Jaccard Index. The former is used to measure the similarity of SCDG structure; while the latter represents the similarity of behavior feature set, a higher level abstraction extracted from SCDG. We first set the ratio of replaced system calls as 0%, 10%, 20%, and 30% and then generate 4 mutations respectively for each testing malware sample.

[3] http://vxheaven.org/src.php
[4] http://www.cuckoosandbox.org/

Table 4. Test set statistics

Sample	Type	LoC #	SCDG	
			Node #	Edge #
BullMoose	Trojan	30	602	360
Clibo	Trojan	90	698	342
Branko	Worm	270	590	332
Hunatcha	Worm	340	756	408
WormLabs	Worm	420	895	506
KeyLogger	Trojan	460	811	439
Sasser	Worm	950	1860	1044
Mydoom	Worm	3276	9342	5418

Then we run these mutations in Cuckoo Sandbox to collect SCDGs in order to compute graph edit distance. After that, we convert SCDGs to feature sets to calculate their Jaccard Index.



(a) Graph edit distance (b) Jaccard Index

Fig. 6. Graph edit distance and Jaccard Index after replacement attacks

Graph edit distance We measure the similarity of SCDG G_1 and SCDG G_2 via graph edit distance [13], which is defined as

$$d(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{max(|G_1|, |G_2|)}$$

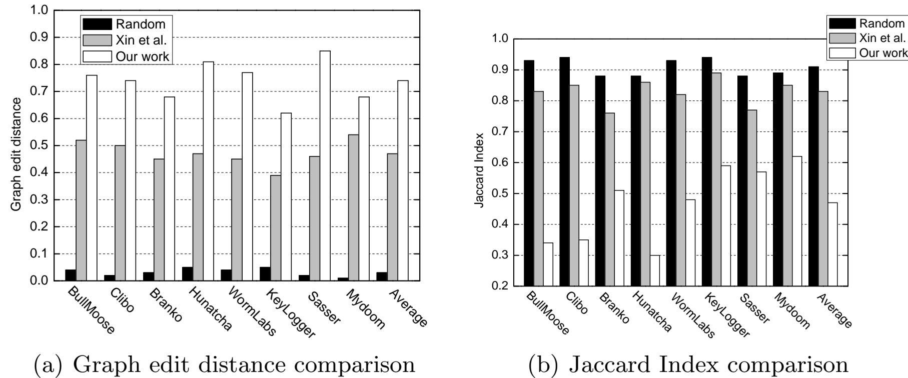
$MCS(G_1, G_2)$ is the maximal common subgraph and $|G|$ is the number of nodes in a graph. The value of the distance varies from 0.0 to 1.0. Distance value 0.0 denotes that two graphs are identical. Park et al. employed the graph edit distance for malware classification and clustering [28, 29], where they set similarity threshold as 0.3. Graph distance above the threshold means two malware samples are different. Taken the sample with 0% replacement ratio as the baseline,

Fig. 6(a) shows the graph edit distance after replacement attacks. Basically the graph edit distance increases steadily as the amount of replaced system calls raises. Please note that when we only enforce 20% replacement, all the distances are beyond the threshold of 0.3. This experiment demonstrates that our replacement attacks change the structure of SCDG significantly.

Jaccard Index Assume behavior feature set of malware sample a and b are F_a and F_b, Jaccard Index is defined as

$$J(a,b) = \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$$

Bayer et al. [7] identified two similar malware feature sets by checking whether their Jaccard Index is ≥ 0.7. Similar with the setting of Fig. 6(a), Fig. 6(b) presents the result of Jaccard Index after replacement attacks. We can draw a similar conclusion that Jaccard Index reduces as replacement ratio increases. However, the decline rate of Jaccard Index is not as large as the rising rate of graph edit distance. We attribute this to a better fault tolerance of large scale feature set. For example, Mydoom in our testing set has more the 1000 features. Consequently, small portion of system calls replacement imposes less effect on Jaccard Index. In spite of this, when the replacement ratio is increased to 30%, all of the the Jaccard Index value are below the similarity threshold of 0.7.



(a) Graph edit distance comparison

(b) Jaccard Index comparison

Fig. 7. Our attacks vs. other approaches

Our attacks vs. other approaches Furthermore, we compared our attacks with two other attacking approaches, that is system call random insertion ("Random" bar) and Xin et al.'s approach [37], which obfuscates SCDG by replacing a dependency edge with a new vertex and two new edges. The ratio of new system calls insertion, replaced edges and replaced system calls are all set as 30%. The comparison results are presented in Fig. 7. The quite small graph edit distance

and large Jaccard Index value show that SCDG is resilient to the attack of system call random insertion, which does not consider data flow dependencies. As shown in Fig. 7(a), although Xin et al.'s approach is able to subvert the structure of SCDG (the distance is > 0.3), our attacks outperform their approach by a factor of 1.6x on average. Moreover, Fig. 7(b) indicates that Xin et al.'s attacking only has a marginal effect on the behavior feature set such as BCHKK-data [7]. The reason is Xin et al.'s approach neither introduces new OS objects nor brings new dependencies between OS objects.

5.3 Against Behavior-based Clustering

In this section, we demonstrate that replacement attacks are able to impede behavior-based malware clustering approach. We choose the clustering approach proposed by Bayer et al. [7], which is a state-of-the-art clustering system for malware behavior. Bayer et al.'s approach contains two major steps: 1) employ locality sensitive hashing (LSH) to find approximate near-neighbors of feature sets; 2) perform single-linkage hierarchical clustering.

We use the LSH code from [5] in our experiment. To fairly evaluate the clustering approach, we stick to a similar setup. The Jaccard Index threshold and LSH parameters, are all exactly the same as in [7]. As mentioned in Section 5.1, malware samples in our initial dataset belong to 8 different families. To enlarge the dataset for our malware clustering evaluation, we generate 5 datasets:

– Dataset 0: We apply various polymorphism obfuscation and packing [31] on our initial samples. For each family, we generate 30 variants. All mutations in each group are only different in terms of static properties. The samples within the same family exhibit quite similar behavior.
– Dataset $1 \sim 3$: We set system call replacement ratio as 10%, 20% and 30% respectively and then produce 30 variants for every family under each replacement ratio setting. Each dataset includes 240 instances.
– Dataset 4: We mix all samples within Dataset $0 \sim 3$ to this dataset, which comprises 960 malware samples in total.

We perform LSH-based single-linkage hierarchical clustering on each dataset. The quality of the clustering results is measured by two metrics: *precision* and *recall*. The goal of *precision* is to measure how well a clustering algorithm assigns malware samples with different behavior to different clusters, while *recall* indicates how well a clustering algorithm puts malware with the same behavior into the same cluster. The naive clustering method that creates only one cluster comprising all samples has the highest recall (1.0), but the worst precision. On the contrary, the method sets up a clustering for each sample achieves the highest precision (1.0) but with low recall number. An optimal clustering method should provide both high precision and recall at the same time. Please refer to [7] for detailed information.

Table 5 summarizes our results. Since the samples in Dataset 0 are only different in terms of static features, the clustering result has the optimal precision and recall. Because 6 samples crashed after applying virtualization obfuscators [16],

Table 5. Quality of the clustering

Dataset	0	1	2	3	4
Samples #	240	240	240	240	960
Cluster #	8	12	35	110	208
Precision	1.000	0.981	0.978	0.965	0.973
Recall	0.975	0.933	0.483	0.121	0.529

the recall value is slightly smaller than 1.0. The results of Dataset 1 $\sim$ 3 show the trend that the recall value falls as system call replacement ratio raises. For example, under the replacement ratio of 30%, on average only about 2 samples are clustered into each family. A small recall value implies that more clusters are created than expected. Dataset 4 simulates a real scenario we mentioned in Section 3.1: malware samples after replacement attacks, mixed with other suspicious binaries, are finally collected for clustering. The low recall value demonstrates that our approach is effective in practice.

5.4 Performance

Since switching between kernel and user mode is inherently expensive, the redundant system calls introduced by replacement attacks will no doubt impact runtime performance. We measure runtime performance after applying replacement attacks on 5 SPEC CPU2006 benchmarks, including `bzip2`, `libquantum`, `omnetpp`, `astar` and `xalancbmk`. Our testbed is a laptop with a 2.30GHz Intel(R) Core i5 CPU and 8GB of memory, running on the operating system of Windows 7. On average, testing programs have a slowdown of 1.33 times (normalized to the runtime without transformation) when the system call replacement ratio is 30%. Considering the significant effect under this replacement ratio, the performance tradeoff is worthy.

6 Discussion

Limitations Currently the compatibility with Visual Studio and LLVM tool chain is not perfect. For example, C++ standard library and Windows Platform SDK are not fully supported by clang, which prevent us from testing more complicated malware. The attacking strategies we summarized in Section 3.3, especially the sub-SCDG mutation rules are limited. Implementing the same functionality through diverse ways need comprehensive domain knowledge. We plan to extend our replacement attacks arsenal in future work.

Possible ways to defeat We suggest possible ways to defend against replacement attacks. As one of our attacking strategies is to insert redundant dependencies, the size of SCDG could be enlarged. An analyzer is able to detect such change by comparing new SCDG with the original one. However, without more close investigation (usually involving tedious work), analyzer cannot easily differentiate whether the size change of SCDG comes from incremental updates or our

attacks. Another countermeasure is to normalize the behavior graph mutations. For example, the multiple semantically equivalent graph patterns of malware replication can be unified as a canonical form before clustering. The effort in this direction is Martignoni et al.'s work [25]. They designed a layered architecture to detect alternative events that deliver the same high-level functionality. However, admitted by the authors, the layered hierarchy is generated manually and tested only with 7 malware samples. A general and automated behavior graph normalization is still missing. Moreover, high-level malware behavior abstractions may overlook subtle distinctions among malware samples. Therefore, the higher-level of behavior abstractions are probably valid in distinguishing malware from benign program, but are incompetent to differentiate malware variants. Another way is to perform more fine-grained data flow analysis. For example, If the data passed in two sequential dependencies are not changed, the medium system call is probably a redundant native API such as *NtSetInformationFile* and *NtDuplicateObject*. However, this approach cannot defeat sub-SCDG mutations, which may completely change the structure of sub-SCDG.

7 Conclusion

Behavior-based malware specifications have been broadly employed in malware detection and clustering. In this paper we study the vulnerability of current behavior based malware analysis and propose replacement attacks to impede malware behavior specifications. We distill general attacking strategies by mining large malware behavior data sets and develop a compiler level prototype to demonstrate their feasibilities. Our evaluation on real malware samples shows that the transformed malware could evade malware similarity comparison and impede behavior-based clustering. We expect our study can cultivate further research to improve resistance to this potential threat.

Acknowledgements

We are very grateful to Paolo Milani Comparetti and Christopher Kruegel for providing access to the BCHKK-data dataset. This research was supported in part by the NSF Grant CNS-1223710, CCF-1320605 and ARO W911NF-13-1-0421 (MURI).

References

1. Cybercriminals sell access to tens of thousands of malware-infected Russian hosts. `http://www.webroot.com/blog/2013/09/23/`, last reviewed, 10/03/2014.
2. Getting started with the llvm system using Microsoft Visual Studio. `http://llvm.org/docs/GettingStartedVS.html`, last reviewed, 10/03/2014.
3. Malicious software and its underground economy. `https://www.coursera.org/course/malsoftware`, last reviewed, 10/03/2014.
4. Windows registry persistence, part 2: The run keys and search-order. `http://blog.cylance.com`, last reviewed, 10/03/2014.

5. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1), Jan. 2008.
6. D. Babić, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *Proceedings of the 23rd Int. Conference on Computer Aided Verification (CAV'11)*, 2011.
7. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'09)*, 2009.
8. U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, 2010.
9. B. Biggio, I. Pillai, S. Rota Bulò, D. Ariu, M. Pelillo, and F. Roli. Is data clustering in adversarial settings secure? In *Proceedings of the 6th ACM Workshop on Artificial Intelligence and Security (AISec'13)*, 2013.
10. B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Rol. Poisoning behavioral malware clustering. In *Proceedings of the 7th ACM Workshop on Artificial Intelligence and Security (AISec'14)*, 2014.
11. A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International Conference on World Wide Web*, 1997.
12. D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, 2006.
13. H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3-4):255–259, 1998.
14. X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, 2008.
15. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE' 07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2007.
16. K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.
17. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'08)*, 2008.
18. M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
19. M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the Workshop on Virtual Machine Security (VMSec'09)*, 2009.
20. C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zho, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
21. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID'05)*, 2005.

22. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, 2004.
23. M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.
24. W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu. Shadow attacks: Automatically evading system-call-behavior based malware detection. *Computer Virology*, 8(1-2):1–13, 2012.
25. L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.
26. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.
27. R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT'09)*, 2009.
28. Y. Park and D. Reeves. Deriving common malware behavior through graph clustering. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, 2011.
29. Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the 6th Annual Workshop on Cyber Security and Information Intelligence Research*, 2010.
30. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 2011.
31. K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1), 2013.
32. M. Russinovich. Inside the native api. `http://netcode.cz/img/83/nativeapi.html`, last reviewed, 10/03/2014.
33. M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, February 2012.
34. A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'11)*, 2011.
35. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, 2002.
36. Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Proceedings of the 2011 European Symposium on Research in Computer Security (ESORICS'11)*, 2011.
37. Z. Xin, H. Chen, X. C. Wang, P. Liu, S. Zhu, and B. Mao. Replacement attacks on behavior based software birthmark. In *Proceedings of the 14th Information Security Conference (ISC'11)*, 2011.