Impeding Behavior-based Malware Analysis via Replacement Attacks to Malware Specifications

Jiang Ming · Zhi Xin · Pengwei Lan · Dinghao Wu · Peng Liu · Bing Mao

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 also clearly stand out from other programs. Although the traditional malware specifications based on syntactic signatures are efficient, they can be easily defeated by various obfuscation techniques. Since the malicious behavior is often stable across similar malware instances, behavior-based specifications which capture real malicious characteristics during run time, have become more prevalent in anti-malware tasks, such as malware detection and malware clustering. This kind of specification is typically extracted from the system call dependence graph that a malware sample invokes. In this paper, we present *replacement attacks* to camouflage similar behaviors by poisoning behavior-based specifications. The key method of our attacks is to replace a system call dependence graph to its semantically equivalent variants so that the similar malware samples within one family turn out to be different. As a result, malware analysts have to put more efforts into reexamining the similar samples which may have been investigated before. We distil general attacking strategies by mining more than $5,200$ malware samples' behavior specifications and implement a compiler-level prototype to automate replacement attacks. Experiments on 960 real malware samples demonstrate the effectiveness of our approach to impede various behavior-based malware analysis tasks, such as similarity comparison and malware clustering. In the end, we also discuss possible countermeasures in order to strengthen existing malware defense.

Keywords: behavior based malware analysis, system call dependence graph, replacement attack, obfuscation

Jiang Ming
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: jum310@ist.psu.edu

Zhi Xin
Nanjing University, Nanjing 210093, China
E-mail: zxin@nju.edu.cn

Pengwei Lan
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: pul139@ist.psu.edu

Dinghao Wu
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: dwu@ist.psu.edu

Peng Liu
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: pliu@ist.psu.edu

Bing Mao
Nanjing University, Nanjing 210093, China
E-mail: maobing@nju.edu.cn

1 Introduction

Malware, or malicious software with harmful intents 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 the old code with new features or obfuscation methods [28]. With thousands of malware instances appearing every day, efficiently processing a large number of malware samples which exhibit similar behavior, has become increasingly important. A key step to improve efficiency is to define the 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 such general specifications. For example, every time a suspicious program is found in the wild, malware analysts can quickly determine whether it belongs to previously known families by matching their specifications.

In the malware arms race, the malicious code keeps evolving to evade detection. The classical syntactic specifications are insufficient to defeat various obfuscation techniques, such as polymorphism [26], binary packing [39], 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 represent the malicious behavior naturally, such as download and execution, replication and remote injection. The main interface for malware to interact with the operating system is through system calls[1]. The data flow dependencies among system calls are expressed as an acyclic graph, namely system call dependency graph (SCDG), where the nodes represent the system calls executed, and a directed edge indicates a data flow between two nodes. Typically, a dependency edge derives from the return value or the output argument of a previous system call. When a data source is passed to one of its succeeding system call's input-argument, a directed edge connecting these two nodes is created. Since such data flow dependencies are stable and hard to be reordered, SCDG has been broadly accepted as a reliable abstraction of malware behavior [15, 20], and widely employed in malware detection [6,25] and large scale malware clustering [7,35].

With quite a number of compelling applications, the malware specifications built on SCDG look 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 behavior-based malware analysis. We show that it is possible to *automatically conceal similar behavior specifications among malware variants by replacing an SCDG to its semantically equivalent ones.* As a result, similar mal-

ware variants show large distances and are assigned to different families. Eventually, malware analysts have to put more efforts into reanalyzing a large number of malware samples exhibiting similar functionalities. In this way, the effect of replacement attacks looks like the denial-of-service attack. To achieve this goal, we first mine two large data sets to identify popular system calls, OS objects, and their 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: 1) the transformation is semantics perseverving; 2) the new generating dependence relationships are so common that they cannot be easily recognized. After transformation, similar malware samples reveal a large distance when they are measured with widely used similarity metrics, such as graph edit distance [13] and Jaccard Index [11]. As a result, the further analyses that rely on SCDG (e.g., malware detection and clustering) are possibly misled.

To demonstrate the feasibility of replacement attacks, we have developed a compiler-level prototype, *API Replacer*, to automatically perform the transformation on top of LLVM framework [27] 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 clutters both SCDG structures and the higher level abstractions from SCDG. In addition, we demonstrate that the further analysis tasks such as malware similarity comparison and behavior-based malware clustering are misled as well. The cost of transformation is low, and the execution overhead after transformation is moderate. Note that our approach does not look for making malware analysis infeasible, but seeks to increase the costs of automatical malware defense solutions, which is practical in the malware arms race.

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 a 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 the replacement attacks by developing a compiler-level prototype to perform source

---

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

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 countermeasures 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 these methods rely on system call sequences or graphs and therefore they are the candidates of replacement attacks targets. 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 analysis, dynamic analysis is less vulnerable to various code obfuscation schemes [32]. Christodorescu et al. [15] introduce malware specifications built 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, the 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 [20,21,33], determining malware family in which the instances share similar functionalities [7,24,6,35], and detecting malicious behavior [8,25,30] by matching behavior-based malware specifications. However, none of the presented approaches is explicitly designed to be resilient to our replacement attacks.

Malware behavior analysis attacks As behavior-based malware analysis has become prevalent, some countermeasures have been proposed to evade this kind of analysis as well. Since collecting malware behavior is typically performed in a controlled sandbox environment, the lion's share of previous work focuses on runtime environment detection [14,18,34,37]. If a malware sample detects itself running in a sandbox rather than real physical machine, it will not carry out any malicious behaviors. Multiple ways have been proposed to defeat such environment-sensitive malware. Dinaburg et al. [17] utilize hardware virtualization to 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 indicate possible evasion attempts [22]. Our method does not detect sandbox and is valid in any runtime environment. Our replacement attacks share the similar goal with recent work by Biggio et al. [9,10] in that we all attempt to subvert malware clustering. However, our work is different from these previous work in two ways. First, our approach obfuscates data flow dependencies between system calls; while the behavior features that they attack do not contain data flow dependencies. As the data flow relationships between system calls or behavior features are highly resilient to random noise insertion, our attacking method is more challenging. Second, Biggio et al.'s work has a common "inverse feature-mapping" problem [10], that is, they directly manipulate malware behavior feature set instead of real malware code. Therefore, their attacks may not be feasible in practice. In contrast, to demonstrate the feasibility of our approach, we develop a compiler-level converter to perform the real replacement attacks.

System call obfuscation The original idea to obfuscate system call API can be traced back to *mimicry attack* [19,43], whose primary goal is to impede intrusion detection. Kim et al. [23] present a polymorphic attack to sequence-based software birthmarks. *Illusion* [42] 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. [29] present *shadow attacks* by partitioning a malware sample into multiple shadow processes and each shadow process reveals no-recognizable malware behavior. But it's still an open question to launch a multi-process sample covertly. Our proposed attacks are inspired by Xin et al. [45]'s approach to subverting 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 value. In contrast, our approach provides multiple attacking strategies. In addition, Xin et al. [45]'s attack code is pre-loaded as a dynamic library when a program starts running. However, it is quite easy to detect such library interruption. Our *API Replacer* embeds new system call dependencies at LLVM IL level and then compiles the modified IL to the binary code transparently. In this way, our approach has better stealth.

3 Replacement Attacks Design

In this section, we present the key design of replacement attacks. We start by introducing the background information about system call dependency graph (SCDG). Our approach attempts to obfuscate the structure of SCDG. Then we discuss the threat model of replacement attacks. The design of replacement attacks is inspired by two studies we performed. After that, we will introduce our replacement attacks arsenal with a motivating example.

3.1 Background

In spite of various obfuscation methods (e.g., metamorphism and polymorphism) that the malware authors adopt, malware samples within the same family tend to reveal similar malicious behavior [28]. Our goal in this paper is to separate similar malware variants by modifying the structure of SCDG, which is the most prevalent specification to represent malware behavior. 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[2]. In the original SCDG (as shown in Fig. 1 (a)), 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 → NtClose". Windows API "SetFilePointer" in the new code (as shown in Fig. 1 (b)) moves the file pointer and returns new position, which is quite similar to "lseek" system call in Unix. The return value of "SetFilePointer" 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).

---

[2] In the code, we use Windows API as a proxy for Windows native API. We will discuss this issue further at Section 4.

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", the native API "NtSetInformationFile" is invoked to change the position information of the file object represented by "hFile1". In this way, the new code still preserves the original data flow, while the structure of original SCDG changes significantly. Note that the file object in Fig. 1 (b) is updated with new position information. However, the file object is closed immediately, imposing no lasting side effect to the final state.

3.2 Threat Model

A typical threat model to apply replacement attacks is illustrated in Fig. 2. Before propagating malware samples, malware authors take the initial version of malware source code as input to *API Replacer*, our compiler-level transformation tool. The outputs are multiple malware variants in binary form. The generated variants share very similar malicious functionalities but exhibit different behavior specifications. Then cybercriminals spread these malware samples to the Internet or plant them in the vulnerable hosts to perform malicious purposes. Suppose these new malware variants, with other suspicious binaries are finally collected by anti-malware companies. To classify a large number of malware and select the samples that need further investigations, anti-malware companies utilize automated clustering tools to identify samples exhibiting similar behavior. These tools normally execute malware instances in a sandbox and collect runtime information (e.g., system alls and their arguments) to generate behavior specifications (e.g., SCDG), which will be normalized and then fed to clustering algorithm. As we mentioned in Section 2, current malware clustering tools are not designed to defeat replacement attacks explicitly. Therefore, the very similar malware mutations after replacement attacks are probably assigned to multiple clusters instead of one cluster. In that case, malware analysts have to waste excessive efforts to reanalyze these similar samples and the samples that require more attention may be left in the basket.

3.3 Mining Two Large Data Sets

Since there are various expressions of malware behavior based on SCDG, we first mine two large and representative data sets of malware behavior specifications used for malware detection and clustering. The min-
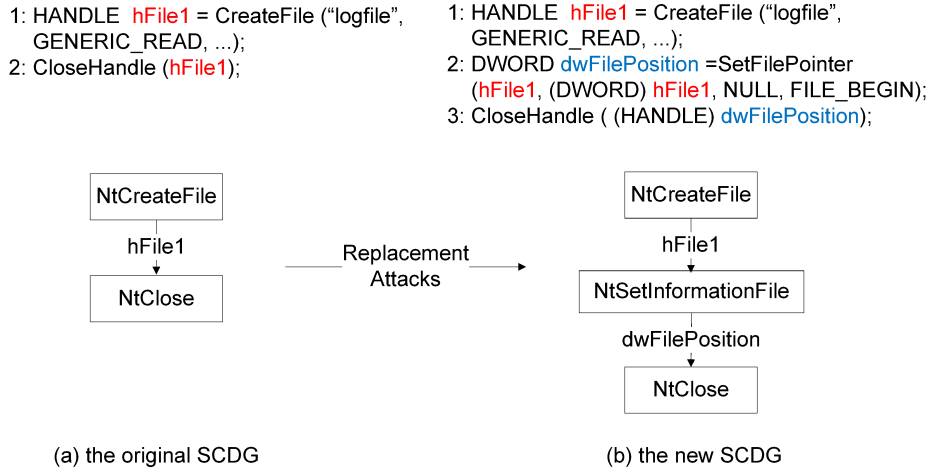
```
1: HANDLE  hFile1 = CreateFile ("logfile",
   GENERIC_READ, ...);
2: CloseHandle (hFile1);
```

```
1: HANDLE  hFile1 = CreateFile ("logfile",
   GENERIC_READ, ...);
2: DWORD dwFilePosition =SetFilePointer
   (hFile1, (DWORD) hFile1, NULL, FILE_BEGIN);
3: CloseHandle ( (HANDLE) dwFilePosition);
```


(a) the original SCDG

(b) the new SCDG

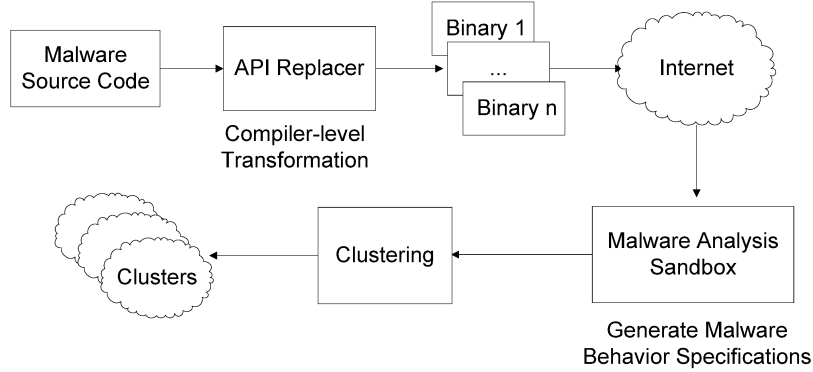**Fig. 1** An example of SCDG before and after replacement attacks.

Fig. 2 Illustration of replacement attacks threat model.

ing results are popular dependencies and common sub-SCDGs, which are the possible targets we may attack.

- BRS-data [6] is used by Babić et al. to evaluate malware detection via tree automata inference. BRS-data contains system call 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 the two-dimensional structure of 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) di-

rectly utilize rich structural information contained in SCDG [15,36,20], which is able to match behavioral patterns exactly but lacks scalability; 2) extract higher level abstractions from SCDG to fit for efficient large-scale malware analysis [7,8,38] at the cost of precision. The similarity of instances in 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]. Our approach is designed to attack the malware specifications like both BRS-data and BCHKK-data.

3.3.1 Popular Dependencies

We calculate the most popular native API dependencies from BRS-data, OS operations and dependencies from BCHKK-data. Table 1 lists eleven 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

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.

OS object type	OS operation
file	open, create, read, write, set_information, query_file, query_information, query_directory
registry	create, open, query_value, set_value
section	query, create, map, open, mem_read
process	create, open, query
thread	create, query, resume
OS object dependency	
file → file, registry → file, registry → registry, process → thread, section → file, file → section	

types, operations and dependencies from BCHKK-data. We believe as long as we diversify these popular dependencies and behavior features, the similarity among malware variants can drop significantly.

3.3.2 Common Sub-SCDGs

Although extracted from two different sources, the data in Table 1 and Table 2 reveal some common malicious functions, which are mapped to sub-SCDGs. The top three popular sub-SCDGs are corresponding to malware replication, registry modification for persistence and code remote injection. For example, the common dependencies about "NtMapViewOfSection" and the OS objects dependency between file and section, indicate that malware replications are commonly implemented via memory mapped file, which directly maps a malware sample into a memory area to speed up disk I/O. Besides, malware instances often configure Windows registry for persistence in order to run automatically when the compromised machine restarts, leading to frequent operations on Windows registry. "NtOpenProcess → NtWriteVirtualMemory" and "process → thread" are mainly introduced by creating a new thread in a remote process, the most common way to launch malware covertly in vulnerable hosts [41]. If we are able to implement these common functions through different ways, the corresponding sub-SCDGs are very likely changed as well.

3.4 Attacking Strategies

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

1. (R1) Potency: our replacement attacks should obfuscate the malware specifications like both BRS-data and BCHKK-data. In another word, replacement attacks have to invalidate various malware behavior similarity metrics, such as graph edit distance and Jaccard Index.
2. (R2) Semantics-preserving: new system calls and dependencies impose no side effect to the original data flow.
3. (R3) Stealth: the transformed SCDG should be as common as possible so that replacement attacks cannot be easily identified.

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).

Furthermore, 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, a popular malicious behavior—malware replication, can be implemented through two methods. The first one is utilizing memory-mapped file to speed up disk I/O, and the native APIs such as NtCreateSection, NtMapViewOfSection, and NtUnmapViewOfSection are involved. The second IO method is via the conventional file manipulating native APIs, such as NtCreateFile, NtReadFile, and NtSetInformationFile. The SCDGs under these two methods are completely different. Therefore, our second attacking strategy is to transform 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 3.3 is that popular dependencies can also be served as possible candidates to be embedded in an 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.

3.5 Replacement Attacks Arsenal

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

3.5.1 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 that use FileHandle as the medium. NtSetInformationFile is used to adjust the file pointer. By carefully crafting the input arguments, we can make "NtSetInformationFile" as a null operation. We have illustrated such example 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* → *NtSetInformationFile* dependency, where the output of "NtQueryInformationFile" ("FileInformation") is passed to "NtSetInformationFile". The two new dependencies also appear commonly.
4. The medium of "void *address" shown in Table 1 receives the address of a mapped memory. To handle this medium, we can insert "NtQueryVirtualMemory" or "NtReadVirtualMemory", which do not affect the memory mapped address.

3.5.2 Sub-SCDG Mutations

We present multiple implementation ways to achieve the top three popular malicious sub-tasks discussed in Section 3.3.2. Furthermore, we make sure that each implementation reveals a different sub-SCDG with the 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, the following reading and writing to the file are simply manipulating the mapped memory region, which produces OS objects dependencies between file and memory section. First, we can choose to map either the source or destination file to the memory section. Another implementation is explicitly through file I/O operations. For example, we can copy a file by calling "NtReadFile" and "NtWriteFile" instead of using memory as the medium.
2. Modify registry for persistence. Malware often add entries into the registry to remain active after rebooting. 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 modify.
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 [41].

3.6 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 ("hunatcha.exe') and then writing the memory content to the destination file ("ladygaga.mp3.exe"). Fig. 4(a) presents the feature set abstracted from Fig. 3(a) according to the definition of BCHKK-data [7]. The first three lines are operations (open, create, write, etc.) on OS objects (file, section). The fourth line is an OS dependency from section to the destination file.

As shown in Fig. 3(b), we first mutate the SCDG shown in Fig. 3(a) by switching the file mapped to the memory; that is, we explicitly map the destination file (not source file) into the memory. Then file copying is

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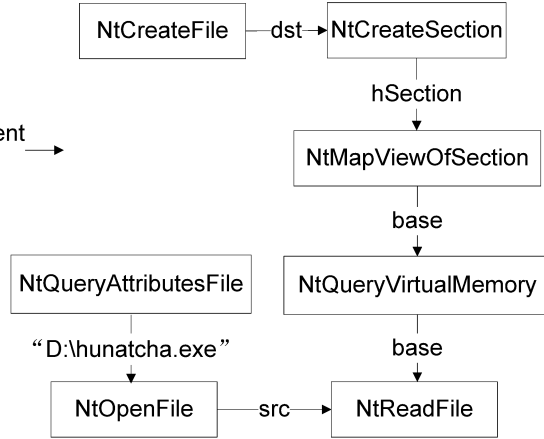5: NtWriteFile (dst, base, length (src), ···);
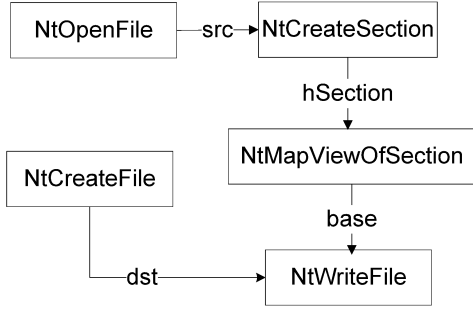
1: NtQueryAttributesFile ("D:\hunatcha.exe", ···);
2: HANDLE src = NtOpenFile ("D:\hunatcha.exe", ···);
3: HANDLE dst = NtCreateFile
    ("\My Shared Folder\ladygaga.mp3.exe", ···);
4: HANDLE hSection= NtCreateSection (···, dst);
5: void *base = NtMapViewOfSection (hSection, ···);
6: NtQueryVirtualMemory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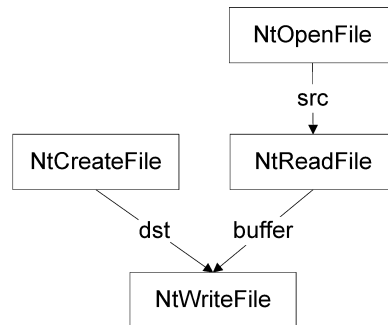7: *base = NtReadFile (src, length (src) , ···);

(a) the original SCDG

(b) the new SCDG

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.
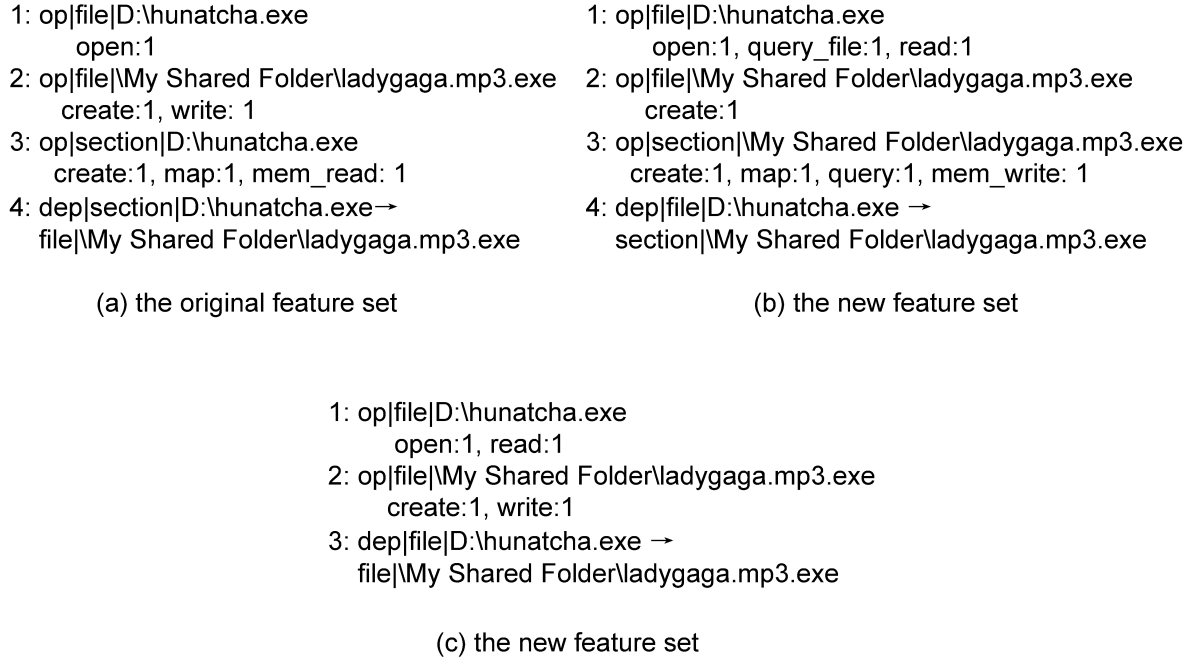
1: op|file|D:\hunatcha.exe
    open:1
2: op|file|\My Shared Folder\ladygaga.mp3.exe
    create:1, write: 1
3: op|section|D:\hunatcha.exe
    create:1, map:1, mem_read: 1
4: dep|section|D:\hunatcha.exe→
    file|\My Shared Folder\ladygaga.mp3.exe

(a) the original feature set

1: op|file|D:\hunatcha.exe
    open:1, query_file:1, read:1
2: op|file|\My Shared Folder\ladygaga.mp3.exe
    create:1
3: op|section|\My Shared Folder\ladygaga.mp3.exe
    create:1, map:1, query:1, mem_write: 1
4: dep|file|D:\hunatcha.exe →
    section|\My Shared Folder\ladygaga.mp3.exe

(b) the new feature set

1: op|file|D:\hunatcha.exe
    open:1, read:1
2: op|file|\My Shared Folder\ladygaga.mp3.exe
    create:1, write:1
3: dep|file|D:\hunatcha.exe →
    file|\My Shared Folder\ladygaga.mp3.exe

(c) the new feature set

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

Table 3 Similarity metrics for the variants a, b and c.

	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

achieved by reading the content of source file to the allocated memory region. At the same time, we also insert redundant data flow dependent system calls to create new dependencies and decouple original dependencies. At line 1 and line 6 at the top of Fig. 3(b), we add NtQueryAttributesFile and NtQueryVirtualMemory, resulting in two redundant dependencies. As a result, 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 copy the sample only through disk file I/O. In this way, no memory section appears both in SCDG and feature set. Table 3 shows the two similarity metrics for these 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. As shown in table 3, the large graph edit distance and small Jaccard Index value after replacement attacks[3] indicate that the similarities of malware variants drop substantially.

---

[3] The similarity metrics and threshold will be discussed in detail in Section 5.2.

4 Implementation

To automate the attacking strategies (discussed 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 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. 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 to change the original SCDG. Section 3.4 describes these steps in details. After that, our pass updates the changes of the call graph. Algorithm 1 lists each step of API Replacer's transformation pass.

The transformed LLVM IR code is passed to LLC to emit object code. Then the object code is 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. The major imple-
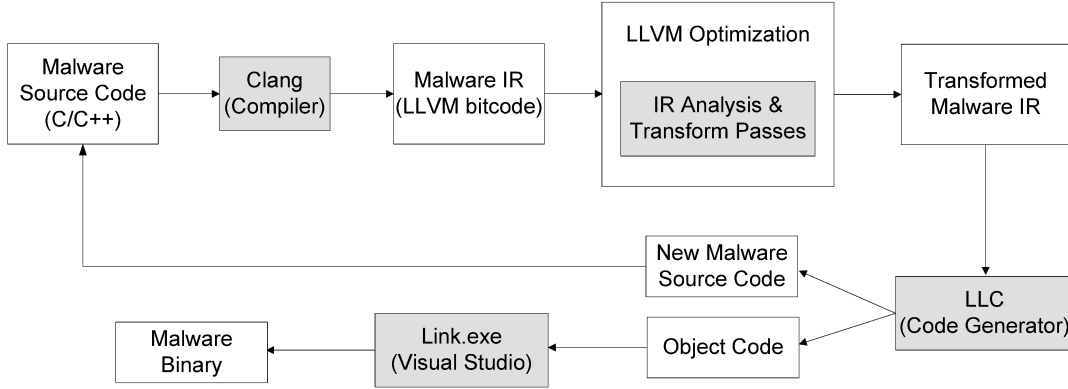
Fig. 5 The architecture of API Replacer.

mentation choice we made is using Windows API as a proxy for Windows native API. The reason is Windows native API are not comprehensively documented; while Windows API is well described in MSDN.[4] According to the mapping between Windows API and native API [40], we are able to manipulate Windows API directly.

Algorithm 1 API Replacement 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

---

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 perform comparison experiment to demonstrate that replacement attacks outperforms two other attacking methods. At last, We 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[5]. These malware samples are chosen for two reasons: 1) they do not contain any trigger-based behavior [44] or runtime environment checking condition [22];

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

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[6], 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 four mutations respectively for each testing malware sample. 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.
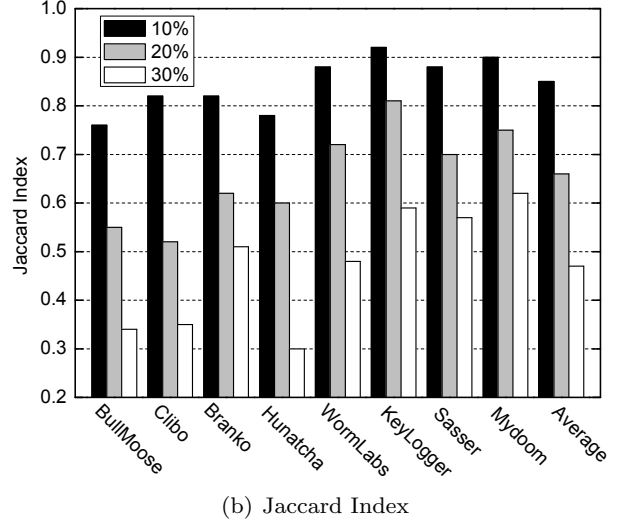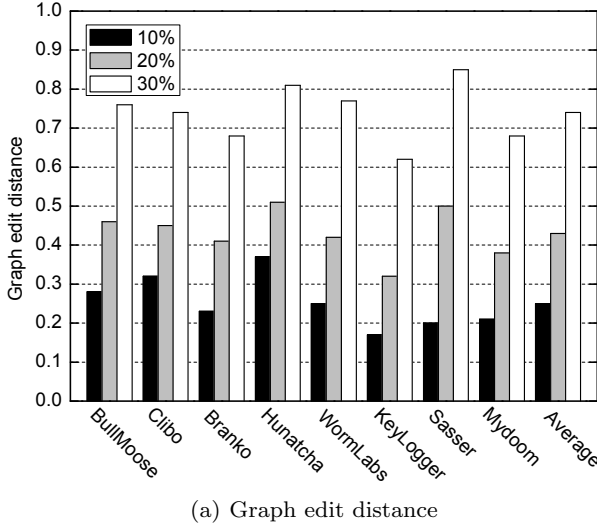
---

[4] http://msdn.microsoft.com/
[5] http://vxheaven.org/src.php
[6] http://www.cuckoosandbox.org/

(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 [35, 36], where they set the similarity threshold as 0.3. Graph distance above the threshold means two malware samples are different. Taking the sample with 0% replacement ratio as the baseline, Fig. 6(a) shows the graph edit distance after replacement attacks. The graph edit distance increases steadily as the amount of replaced system calls raises. Please note that when we enforce only 20% replacement, all the distances go beyond the threshold of 0.3. When the replacement ratio is set as 30%, the graph edit distance is as high as 0.74 on average. This experiment demonstrates that our replacement attacks can 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

Table 5 Similarity metrics of inter-family comparisons.

	10%	20%	30%	inter-family
Graph edit distance	0.25	0.43	0.74	0.78
Jaccard Index	0.85	0.66	0.47	0.36

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 significant 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, a 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 Jaccard Index values are below the similarity threshold of 0.7, with a value of 0.47 on average.

Our attacks vs. inter-family comparisons We conduct inter-family comparisons for the malware samples listed in Table 4. Since all these malware samples have different malicious functionalities, the similarity metrics values of inter-family comparisons represent the approximate upper bound on the degree of the obfuscation attainable by our attacks. Table 5 shows the similarity metrics of inter-family comparisons on average, indicating that the similarities between inter-family samples are quite small. Besides, we also present the average similarity metrics under different replacement ratio settings. It is obvious that with the replacement ratio setting of 30%, our attacking effect is quite close to the optimal result.

Our attacks vs. other approaches Furthermore, we compared our attacks with two other attacking approaches:
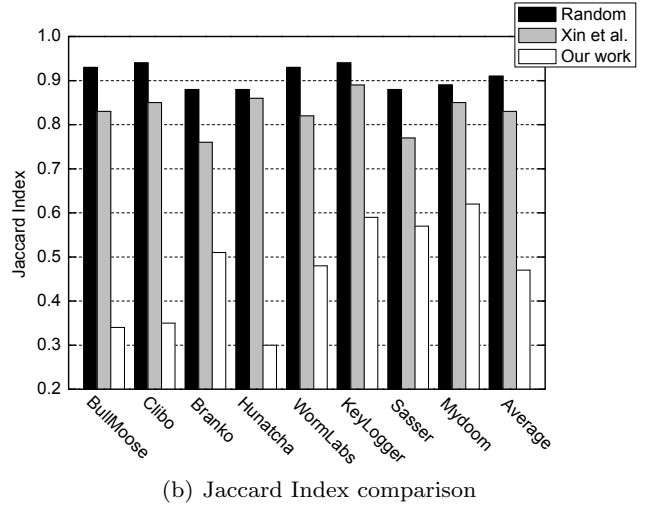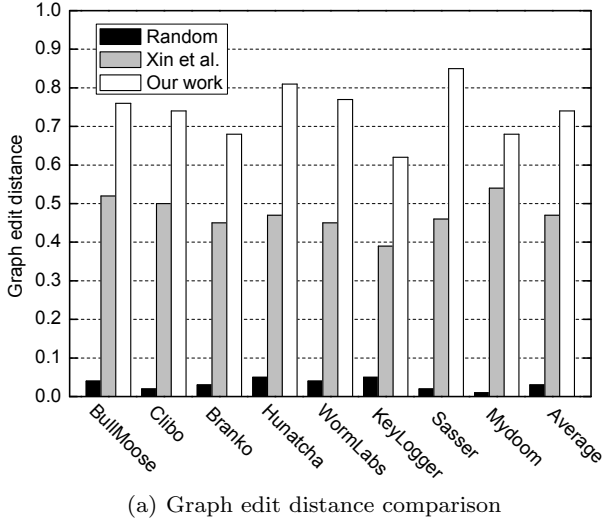
(a) Graph edit distance comparison

(b) Jaccard Index comparison

Fig. 7 Our attacks vs. other approaches.

system call random insertion [23] and Xin et al.'s approach [45]. The comparison results are presented in Fig. 7. The approach of system call random insertion ("Random" bar) adds redundant system calls without considering data flow dependencies; while Xin et al.'s method ("Xin et al." bar) 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 small graph edit distance and large Jaccard Index value show that SCDG is resilient to the attack of system call random insertion, as maintaining the data flow dependencies between system calls can easily remove random system calls. As shown in Fig. 7(a), Xin et al.'s approach is good at cluttering the structure of SCDG (the distance is > 0.3), but our attacks still outperform their approach by a factor of 1.6x on average. The Jaccard Index value in Fig. 7(b) indicates that Xin et al.'s attack only has a marginal effect on the behavior feature set such as BCHKK-data [7]. We attribute this to the limitation of Xin et al.'s approach design, which is unable to introduce additional new OS objects or 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 the same as in [7]. As mentioned in Section 5.1, malware samples in our initial dataset belong to eight different families. To enlarge the dataset for our malware clustering evaluation, we generate five datasets:

– Dataset 0: We apply various polymorphism obfuscation and packing [39] 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 ∼ 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 ∼ 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. Assume we have s malware samles $S_1, S_2, ..., S_s$ in total. A reference clustering which has the ground truth is $M = M_1, M_2, ..., M_m$ with m clusters. The clustering we adopted

Table 6 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

is $N = N_1, N_2, ..., N_n$ with n clusters. For each $N_i, i \in n$, precision value P_i is defined as:

$$P_i = max(|N_i \cap M_1|, |N_i \cap M_2|, ..., |N_i \cap M_m|)$$

The overall precision is calcuated as:

$$P = \frac{P_1 + P_2 + ... + P_n}{s}$$

For each $M_j, j \in m$, recall value R_j is defined as:

$$R_j = max(|N_1 \cap M_j|, |N_2 \cap M_j|, ..., |N_n \cap M_j|)$$

The overall recall is calcuated as:

$$R = \frac{R_1 + R_2 + ... + R_m}{s}$$

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 a small 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 6 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 six samples crashed after applying virtualization obfuscators [16], 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 two samples are clustered into each family. A small recall value implies the similar samples after replacement attacks are distributed into different families, and therefore 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 dividing similar samples into different clusters, which imposes an extra workload to security analysts.
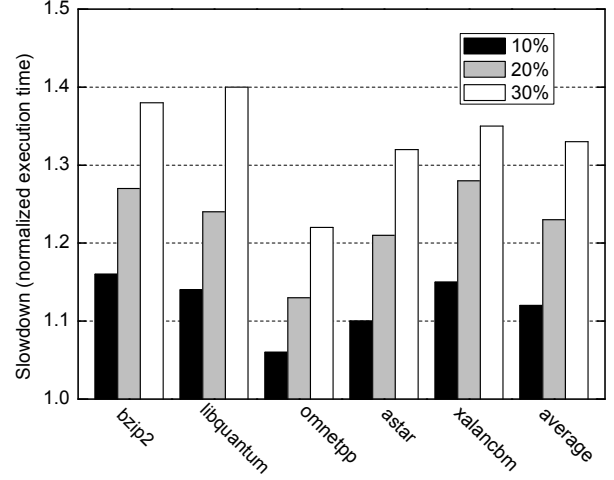
Fig. 8 Slowdown on 5 SPEC CPU2006 programs.

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. As shown in Fig. 8, runtime overhead raises when the replacement ratio increases. 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%. As we have demonstrated in Section 5.2, under this replacement ratio, our attacking effect is quit close to the optimal result. Considering such significant effect, we believe the performance tradeoff is worthy.

6 Discussion

In this section, we discuss the limitations of our approach and future work. In order to strengthen current malware defense approaches, we also provide possible ways to defeat our replacement attacks.

6.1 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. We demonstrate the feasibility of

replacement attacks on Windows platform, but our idea to obfuscate SCDG can be applied in other platforms as well. The attacking strategies we summarized in Section 3.4, especially the sub-SCDG mutation rules are limited. Implementing the same functionality through multiple ways need comprehensive domain knowledge. One direction is to automatically learn equivalent sub-SCDGs with data mining techniques. We plan to extend our replacement attacks arsenal in future.

6.2 Countermeasures

We suggest three possible ways to defend against replacement attacks and discuss their pros and cons. 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), it is hard to differentiate whether the size change of SCDG comes from incremental updates or our attacks. The second 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 a sub-SCDG.

The third 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 [30]. 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 seven malware samples. A general and automated behavior graph normalization is still missing. Moreover, the high-level malware behavior abstractions may overlook subtle distinctions among malware variants. Therefore, the higher-level of behavior abstractions are probably valid in distinguishing malware from benign programs, but are insufficient to differentiate malware variants.

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 obfuscate malware behavior specifications. We distil 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.

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. P. Ferrie. Attacks on virtual machines. In *Proceedings of the Association of Anti-Virus Asia Researchers Conference*, 2007.

19. S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*, 2008.

20. 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.

21. G. Jacob, R. Hund, C. Kruegel, and T. Holz. Jackstraws: Picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX conference on Security*, 2011.

22. 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.

23. H. Kim, W. M. Khoo, and P. Lio. Polymorphic attacks against sequence-based software birthmarks. In *Proceedings of the 2nd Software Security and Protection Workshop (SSP'12)*, 2012.

24. J. Kinable and O. Kostakis. Malware classification based on call graph clustering. In *Journal in Computer Virology Volume 7, Number 4 (2011)*, 2011.

25. 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.

26. 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.

27. 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.

28. 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.

29. 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.

30. 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.

31. J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao. Replacement attacks: Automatically impeding behavior-based malware specifications. In *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS 2015)*, June 2015.

32. 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.

33. R. Paleari, L. Martignoni, E. Passerini, D. Davidson, M. Fredrikson, J. Giffin, and S. Jha. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX Security Symposium*, September 2010.

34. 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.

35. 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.

36. 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.

37. T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Proceedings of the 10th International Conference on Information Security (ISC'07)*, 2007.

38. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 2011.

39. K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1), 2013.

40. M. Russinovich. Inside the native api. `http://netcode.cz/img/83/nativeapi.html`, last reviewed, 10/03/2014.

41. M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, February 2012.

42. 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.

43. 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.

44. 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.

45. 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.