A Categorical Treatment of Malicious Behavioral Obfuscation

Romain Péchoux, Thanh Dinh Ta

A Categorical Treatment of Malicious Behavioral Obfuscation

Romain Péchoux[1] and Thanh Dinh Ta[1,2]

[1] Université de Lorraine - Inria Project Team CARTE, Loria
[2] INRIA Nancy - Grand Est

Abstract. This paper studies malicious behavioral obfuscation through the use of a new abstract model for process and kernel interactions based on monoidal categories. In this model, program observations are considered to be finite lists of system call invocations. In a first step, we show how malicious behaviors can be obfuscated by simulating the observations of benign programs. In a second step, we show how to generate such malicious behaviors through a technique called path replaying and we extend the class of captured malwares by using some algorithmic transformations on morphisms graphical representation. In a last step, we show that all the obfuscated versions we obtained can be used to detect well-known malwares in practice.

Keywords: Behavioral obfuscation, malware detection, monoidal category.

1 Introduction

A traditional technique used by malware writers to bypass malware detectors is program transformation. Basically, the attacker applies some transformations (e.g. useless code injection, change of function call order, code encryption, ...) on a given malware in order to build a new version having the same malicious behavior, i.e. semantically equivalent relatively to a particular formal semantics. This version may bypass a malware detector succeeding in detecting the original malware if the transformation is cleverly chosen. This risk is emphasized in [20] "*an important requirement of a robust malware detection is to handle obfuscation transformation*".

Currently, the works on *Code obfuscation* have been one of the leading research topic in the field of software protection [8]. By using code obfuscation, malwares can bypass code pattern-based detectors so that the detector's database has to be regularly updated in order to recognize obfuscated variants. As a consequence of Rice's theorem, verifying whether two programs are semantically equivalent is undecidable in general, which annihilates all hopes to write an ideal detector. Consequently, efficient detectors will have to handle code obfuscation in a convenient way while ensuring a good tractability. Most of recent researches have focused on *semantics-based* detection [6,20], where programs are described by abstractions independent from code transformations. Since the semantics of

the abstracted variants remains unchanged, the detection becomes more resilient to obfuscation.

In this paper, we will focus on *behavior-based* techniques [11,15] where programs are abstracted in terms of *observable behaviors*, that is interactions with the environment. Beside the works on detection (see [19] for an up-to-date overview), detection bypassing is also discussed academically in [16,23,21,10] and actively (but hardly accessible) in the underground. To our knowledge, there are only a few theoretical works on *behavioral obfuscation*. The lack of formalism and of general methods leads to some risks from the protection point of view: first, malwares deploying new attacks, that is attacks that were not practically handled before, might be omitted by current detectors. Second, the strength of behavior-based techniques might be overestimated, in particular if they have not a good resilience to code obfuscation.

As a illustrating example, consider the following sample, a variant of the trojan `Dropper.Win32.Dorgam` [3] whose malicious behavior consists in three consecutive stages:

- First, as illustrated in the listing of Figure 1, it unpacks two PE files whose paths are added into the registry value `AppInit_DLLs` so that they will be automatically loaded by the malicious codes downloaded later.
- Second, it creates the key `SOFTWARE\AD` and adds some entries as initialized values as illustrated by Figure 2.
- Third, it calls the function `URLDownloadToFile` of Internet Explorer (MSIE) to downloads other malicious codes from some addresses in the stored values.

```
NtCreateFile       (FileHdl=>0x00000734,RootHdl<=0x00000000,File<=\??\C:\WIND
OWS\system32\sys.sys)
NtWriteFile        (FileHdl<=0x00000734,BuffAddr<=0x0043DA2C,ByteNum<=11264)
NtFlushBuffersFile (FileHdl<=0x00000734)
NtClose            (Hdl<=0x00000734)
NtCreateFile       (FileHdl=>0x00000734,RootHdl<=0x00000000,File<=\??\C:\WINDO
WS\system32\intel.dll)
NtWriteFile        (FileHdl<=0x00000734,BuffAddr<=0x0041A22C,ByteNum<=145408)
NtFlushBuffersFile (FileHdl<=0x00000734)
NtClose            (Hdl<=0x00000734)
```

Fig. 1: File unpacking

Since the file unpacking at the first stage is general and the behaviors at the third stage are the same as those of the benign program MSIE, the only way for a behavior-based detector to detect the trojan is by examining its behaviors during the second stage in term of the syscall list[3]:

$$\texttt{NtOpenKey}, \texttt{NtSetValueKey}, \texttt{NtClose}, \texttt{NtOpenKey}, \ldots$$

[3] For readability, we omit the arguments in the syscall lists.

```
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)
NtSetValueKey (KeyHdl<=0x00000730,ValName<=ID,ValType<=REG_SZ,ValEntry<=2062)
NtClose        (Hdl<=0x00000730)
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)
NtSetValueKey (KeyHdl<=0x00000730,ValName<=URL,ValType<=REG_SZ,ValEntry<=
http://ad.***.com:82)
NtClose        (Hdl<=0x00000730)
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)
NtSetValueKey (KeyHdl<=0x00000730,ValName<=UPDATA,ValType<=REG_SZ,ValEntry<=
http://t.***.com:82/***)
NtClose        (Hdl<=0x00000730)
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)
NtSetValueKey (KeyHdl<=0x00000730,ValName<=LOCK,ValType<=REG_SZ,ValEntry<=
http://t.***.com?2062)
NtClose        (Hdl<=0x00000730)
......
```

Fig. 2: Registry initializing

corresponding to the consecutive syscalls of Figure 2 in order to detect this trojan. However, the `NtOpenKey` syscall associated to each `NtSetValueKey` syscall is verbose and can be replaced by a single syscall. Moreover, the key handler can be obtained by duplicating a key handler located in another process, so the call `NtOpenKey` is not mandatory. Consequently, the following syscall lists are equivalent behaviors that the trojan could arbitrarily select in order to perform its malicious task:

$$\texttt{NtOpenKey}, \texttt{NtSetValueKey}, \texttt{NtSetValueKey}, \ldots$$
$$\texttt{NtDuplicateObject}, \texttt{NtSetValueKey}, \texttt{NtSetValueKey}, \ldots$$

The remainder of the paper will be devoted to modestly explain how such lists can be both generated and detected for restricted but challenging behaviors. For that purpose, our main contribution is to construct a formal framework explaining how semantics-preserving *behavioral transformations* may evolve. The underlying mathematical abstraction is the notion of monoidal category that we use to model syscall interactions and internal computations of a given process with respect to the kernel. In a first step, it allows us to formally define the behaviors in term of syscall observations. In a second step, it allows us to define a non-trivial subclass of behaviorally obfuscated programs on which detection becomes decidable. In a last step, we show that, apart from purely theoretical results, our model also leads to some encouraging experimental results since the aforementioned decidability result allows us to recognize distinct versions of malwares from the real-world quite efficiently.

This work was inspired by ideas of R. Milner in [18] where the importance of *effects* that influence each participant in interactions is emphasized. In fact, the kernel and process interaction semantics may be thought of as effects that an execution path has on the process and the kernel, respectively. The current work is an application of such ideas to a more specific context.

Outline In Section 2, we introduce a new abstract and simple model based on monoidal categories, that only requires some basic behavioral properties, and introduce the corresponding notions of observable behaviors. We provide several practical examples to illustrate that, though theoretically oriented, this model is very close to practical considerations. In Section 3, we present the main principles of behavioral obfuscation and some semantics-preserving transformations with respect to the introduced model. In Section 4, we introduce a practical implementation of our model and conclude by discussing related works and further developments.

2 Behavior modeling

We assume the reader to be familiar with category theory (see [4], for an introduction) and, in particular, with the concept of monoidal categories [17] introducing a *tensor product* operator $\otimes$ to represent concurrent computations. As usual, $m, n, \ldots$ will denote *objects* and $s, r, \ldots$ will denote *morphisms* mapping a *source* object, noted $source(s)$, to a *target* object, noted $target(s) = n$, and will be represented by either $m \xrightarrow{s} n$ or $s : m \longrightarrow n$. Let also 1_m denote the identity morphism, for each object m, and let $\circ$ be the associative *composition*.

Morphism (resp. object) *terms* are terms built from basic morphisms (resp. objects) as variables, composition and tensor product. E.g. $(s_1 \circ s_2) \otimes s_3$ is a morphism term and $m_1 \otimes (m_2 \otimes m_3)$ is an object term.

2.1 Syscall interaction modeling

From a practical viewpoint, the computations and interactions between processes and the kernel can be divided in two main sorts, the *system calls* interactions and the process or kernel *internal computations*.

System calls are implemented by the trap mechanism where there is a mandatory control passing from the process (caller) to the kernel (callee). A syscall affects to and is affected by both process and kernel data in their respective memory spaces. Throughout the paper, we will distinguish *syscall names* (e.g. `NtCreateFile`) from *syscall invocations* (e.g. `NtCreateFile(h,...)`). The former are just names while the later compute functions and will be the main concern of our study.

Internal computations are operations inside the process or kernel memory spaces. There is no control passing and they only affect to and are affected by data of the caller memory.

We will abstract this practical viewpoint by a categorical model where computations and interactions (i.e. both syscalls and internal computations) will be represented by morphisms on the appropriate objects. For that purpose, objects will consist in formal representations of physical memories.

Definition 1 (Memory space). *Let Addr be a fixed set of memory adresses. A* memory state *(or value) s is a mapping from a subset of memory addresses*

$B \subseteq Addr$ to memory bits in $\{0,1\}$. The domain of s is defined by $dom(s) = B$. A memory space *m is the set of all memory states corresponding to some fixed domain $B \subseteq Addr$, i.e. $m = \{s \mid dom(s) = B\}$. The domain of m is defined by $dom(m) = B$ and the codomain $codom(m)$ is the set of all binary words of length $\#B$. Given two memory spaces m and n, we write $m \subseteq n$ if $dom(m) \subseteq dom(n)$. In what follows, we will use the notation m^i, $i \in \{k, p\}$, to denote that m is either a* kernel *or a* process *memory space. Given two memory spaces m and n of disjoint domains, $m \cup n$ denotes their disjoint union.*

We now introduce the notion of *interaction category* in order to abstract syscall invocations and internal computations.

Definition 2 (Interaction category). *Let m^p, m^k be memory spaces satisfying $m^p \cap m^k = \emptyset$. The interaction category $\mathcal{C}\langle m^p, m^k\rangle$ is a category defined by:*

- *The set of objects is freely generated from process and kernel memory spaces n^i such that $n^i \subseteq m^i$, $i \in \{k, p\}$, and cartesian products $n^p \times n^k, \ldots$. The terminal unit object e consists in the empty set.*
- *The set of morphisms is freely generated from cartesian projections: π_i, $i \in \{k, p\}$, process and kernel internal computations: $s^i \colon n^i \to o^i$, $i \in \{k, p\}$, and syscall interactions: $s^{p\text{-}k} \colon n^p \times n^k \to o^p \times o^k$.*
- *The tensor product is partially defined on objects and morphisms by*[4]*:*
  - *if $n^i \cap o^i = \emptyset$ then $n^i \otimes o^i = n^i \cup o^i$,*
  - *if $n^p \otimes o^p$ and $n^k \otimes o^k$ are defined then:*

$$(n^p \times n^k) \otimes (o^p \times o^k) = (n^p \otimes o^p) \times (n^k \otimes o^k),$$

  - *if $n^p \otimes o^p$ or $n^k \otimes o^k$ are defined then:*

$$(n^p \times n^k) \otimes o^p = (n^p \otimes o^p) \times n^k \text{ or}$$
$$(n^p \times n^k) \otimes o^k = n^p \times (n^k \otimes o^k).$$

  - *given $s_1 \colon m_1 \to n_1$ and $s_2 \colon m_2 \to n_2$, then $s_1 \otimes s_2 \colon m_1 \otimes m_2 \to n_1 \otimes n_2$ is defined by $s_1 \otimes s_2(v_1 \otimes v_2) = s_1(v_1) \otimes s_2(v_2)$ whenever the following diagram commutes: (i.e. the tensor is defined on objects):*

$$\begin{array}{ccc} m_1 \otimes m_2 & \xrightarrow{s_1 \otimes 1_{m_2}} & n_1 \otimes m_2 \\ {\scriptstyle 1_{m_1} \otimes s_2}\downarrow & & \downarrow{\scriptstyle 1_{n_1} \otimes s_2} \\ m_1 \otimes n_2 & \xrightarrow{s_1 \otimes 1_{n_2}} & n_1 \otimes n_2 \end{array}$$

[4] The tensor represents the concurrent accesses and modifications performed by both internal computations and syscall interactions on memory spaces. A necessary condition for these operations to be well-defined is that they do not interfere, that is they have to operate on disjoint domains.

Remark 1. The set notation $v \in m$ and the categorical notation $v\colon e \to m$ will be interchangeably used depending on the context in order to denote a value (memory state) v of a memory space m. So do the composition notation $s \circ v$ and the application notation $s\,(v)$ that denote the result of applying morphism s to value v.

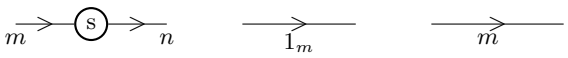
We show that interaction categories enjoy the mathematical abstractions and properties of monoidal categories:

Proposition 1. *Each interaction category is a monoidal category (with a partially defined tensor product operator).*

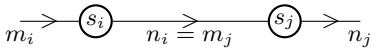
A consequence is that all the abstract properties and graphical representations of monoidal categories can be used in the proofs and remainder of this paper.

Graphical representation Morphism and object terms can be given a standard graphical representation using *string diagrams* [22,14] defined as follows:

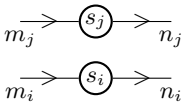
– nodes are morphisms (except for identity morphisms) and edges are objects:



– composition $s_j \circ s_i$:



– tensor product $s_i \otimes s_j$:



In (planar) monoidal categories, diagrams are *progressive* [22], namely edges are always oriented from left to right. Let $\preccurlyeq$ be (by abuse of notation) the reflexive and transitive closure of the relation defined by $s_i \preccurlyeq s_j$ holds if there is an edge from s_i to s_j.

Listing 1.1: Internal computation

```
char *src = 0x00150500;
char *dst = 0x00150770;
strncpy(dst,src,10);
```

Listing 1.2: Syscall invocation

```
char *buf = 0x0015C898;
HANDLE hdl = 0x00000730;
NtWriteFile(hdl,...,buf,1024);
```

Example 1. Listing 1.1 is an example of (process) internal computation. The function `strncpy` can be represented by the (process internal computation) morphism:

$$strncpy^p\colon [src] \otimes [dst] \longrightarrow [src] \otimes [dst],$$

where $[src]$ and $[dst]$ are 10 bytes memory spaces beginning at the addresses `0x150500` and `0x150770`, respectively.

Listing 1.2 is an example of syscall invocation. The invocation of the syscall name `NtWriteFile` is represented by a (syscall interaction) morphism:

$$NtWriteFile^{p\text{-}k}\colon [buf] \times [hdl] \longrightarrow [buf] \times [hdl],$$

where $[buf]$ is a 1024 bytes memory space beginning at the address `0x15C898`, and $[hdl]$ is a memory state identified by the handler `0x730`.

In the interaction category $\mathcal{C}\langle m^p, m^k\rangle$, each internal computation s^p can be considered as a morphism $s^p\colon m^p \longrightarrow m^p$ since internal computations are memory modifiers operating on some previously allocated memory space. In the same spirit, each syscall interaction $s^{p\text{-}k}$ can be seen as a morphism $s^{p\text{-}k}\colon m^p \times n^k \longrightarrow m^p \times o^k$ such that we have either $dom(n^k) = dom(o^k)$ (memory modifier), or $dom(n^k) \subsetneq dom(o^k)$ (memory constructor) or $dom(o^k) \subsetneq dom(n^k)$ (memory destructor).

Example 2. The syscall `NtWriteFile(hdl,...)` of Example 1 is a memory modifier while `NtOpenKey(ph,...)` is a memory constructor, allocating a new memory space identified by `*ph`, and `NtClose(h)` is a memory destructor freeing the memory space identified by `h`.

2.2 Process behaviors as path semantics

We now provide definitions of process behaviors in term of *paths*, namely lists of consecutive morphisms that processes realize during an execution. By assuming that processes can only be examined in finite time, the studied paths are finite.

Definition 3 (Execution path). *An* execution path $X \in \mathcal{X}$ *is a finite list of morphisms of the shape* $X = [s_1^{j_1}, s_2^{j_2}, \dots]$, *with* $j_i \in \{p, p\text{-}k\}$, $\forall i$, *satisfying the following condition: for each* $s_i^{p\text{-}k} \in X$ *of the shape* $s_i^{p\text{-}k}\colon m^p \times n_i^k \longrightarrow m^p \times o_i^k$*:*

- *there is no memory duplication: if* $s_i^{p\text{-}k}$ *is a memory constructor then its constructed memory* $T_i^k = o_i^k \setminus n_i^k$ *is not duplicated, that is* $\forall s_j^{p\text{-}k} \in X$, *if* $j > i$ *then* $T_i^k \cap T_j^k = \emptyset$ *else* $T_i^k \cap n_j^k = T_i^k \cap o_j^k = \emptyset$.
- *there is no memory reuse: if* $s_i^{p\text{-}k}$ *is a memory destructor then its destructed memory* $U_i^k = n_i^k \setminus o_i^k$ *is not reused, that is* $\forall s_j^{p\text{-}k} \in X$, *if* $j > i$ *then* $U_i^k \cap n_j^k = U_i^k \cap o_j^k = \emptyset$.

Note that execution paths correspond to paths that are semantically meaningful: The first condition prevents the system from reallocating a memory address and from accessing to an unallocated one, while the second condition prevents it from accessing to a previously freed memory space.

Definition 4 (Observable path). *Given an execution path* X, *its* observable path $O \in \mathcal{O}$ *consists in the list of all syscall interactions in* X. *The function* $obs\colon \mathcal{X} \to \mathcal{O}$ *returns the observable path of an execution path given as input, e.g.* $obs([s_1^{p\text{-}k}, s_2^p, s_3^p, s_4^{p\text{-}k}]) = [s_1^{p\text{-}k}, s_4^{p\text{-}k}]$.

Execution paths will be used to study all the possible computations (internal computations and syscall interactions) at the process level while observable paths only consist in behaviors that can be grasped by an external observer, that is some sequence of syscall invocations, and will be the main concern of our study.

Example 3. Consider the following listings:

Listing 1.3: X_1

```
strncpy(dst,src1,10);
strncpy(dst+10,src1+10,30);
NtOpenKey(h,...{...dst...});
memcpy(src2,src1,1024);
```

Listing 1.4: X_2

```
memcpy(src2,src1,1024);
strncpy(dst+2,src2,15);
strncpy(dst+17,src1+15,25);
NtOpenKey(h,...{...dst+2...});
```

The corresponding execution paths X_1, X_2 are defined by:

$$X_1 = [strncpy_1^p, strncpy_2^p, NtOpenKey_3^{p\text{-}k}, memcpy_4^p]$$
$$X_2 = [memcpy_1^p, strncpy_2^p, strncpy_3^p, NtOpenKey_4^{p\text{-}k}]$$

and their respective observable paths are defined by:

$$obs(X_1) = [NtOpenKey_3^{p\text{-}k}]$$
$$obs(X_2) = [NtOpenKey_4^{p\text{-}k}]$$

The data modifications caused by an execution path X of a given interaction category can be represented by a morphism term built from the morphisms of X together with identity morphisms. In what follows, the morphism corresponding to these data modifications will be called the **path semantics** of X, denoted $s(X)$.

Proposition 2. *Given an execution path $X \in \mathcal{X}$ of the interaction category $\mathcal{C}\langle m^p, m^k\rangle$, the data modifications caused by X on data at memory adresses $dom(m^p)$ and $dom(m^k)$ is a morphism $s(X)$ of the shape:*

$$s(X)\colon m^p \times n^k \longrightarrow m^p \times o^k, \text{ with } n^i, o^i \subseteq m^i.$$

that can be represented by a morphism term obtained by using the morphisms of X together with identity morphisms.

3 Behavioral obfuscation

In this section, we show a theorem stating that if a benign path has the same effects on the kernel data as another (malicious) path, then there exist (malicious) paths having the same path semantics as the initial malicious one, and the same observations as the benign one. Though not surprising, this result has two main advantages. First, it gives a first formal treatment of camouflage techniques. Second, the proof of this theorem is constructive. It means that it does not only show the existence of such malicious paths but also allows us to build them in an automated way. This is a very first step towards an automated way of detecting such malicious paths.

3.1 Obfuscation

First, we need to give a clear definition of obfuscation: One of the main obfuscation techniques consists in camouflaging behaviors of malwares with those of a benign programs. Such a technique was partly illustrated by the trojan of our motivating example that was hiding some of its behaviors through the use of Internet Explorer functionalities.

Formally, given two execution paths X_1 and X_2 starting at some value $v_0^p \times v_0^k$ and ending at the same value $v_1^p \times v_1^k = s\left(X_2\right)\left(v_0^p \times v_0^k\right) = s\left(X_1\right)\left(v_0^p \times v_0^k\right)$, X_2 **obfuscates** X_1 (or $obs\left(X_2\right)$ **behaviorally obfuscates** $obs\left(X_1\right)$), denoted by $obs\left(X_2\right) \approx obs\left(X_1\right)$, if $obs(X_1) \neq obs(X_2)$.

Example 4. Consider the paths X_1', X_2' respectively consisting of the 3 last morphisms of X_1, X_2 in Listings 1.3 and 1.4, namely:

$$X_1' = [strncpy_2^p, NtOpenKey_3^{p\text{-}k}, memcpy_4^p]$$
$$X_2' = [strncpy_2^p, strncpy_3^p, NtOpenKey_4^{p\text{-}k}]$$

In general, $s(X_1') \neq s(X_2')$ but the equality holds if both execution paths start at values so that the data on $[src1] \cup [src2]$[5] are the same. For these particular values, we have:

$$obs(X_1') = [NtOpenKey_4^{p\text{-}k}] \approx [NtOpenKey_3^{p\text{-}k}] = obs(X_2')$$

Notice that the two syscall invocations have the same name but actually consist in two different morphisms.

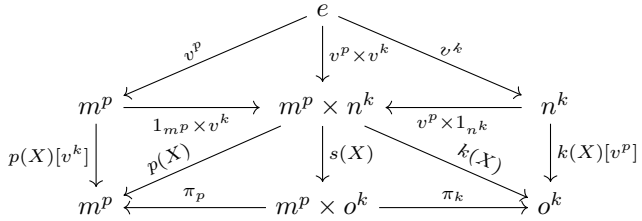
3.2 Camouflage theorem

In order to state the theorem, we need to introduce the notions of process and kernel (partial) semantics to distinguish the effects caused by a path on a kernel memory space from the ones caused on a process memory space.

Definition 5. *Given an execution path $X \in \mathcal{X}$ and its path semantics $s(X)\colon m^p \times n^k \longrightarrow m^p \times o^k$ wrt the interaction category $\mathcal{C}\langle m^p, m^k\rangle$, the:*

- *kernel semantics, noted $k\left(X\right)$, and kernel partial semantics at value v^p, noted $k\left(X\right)[v^p]$,*
- *process semantics, noted $p\left(X\right)$, and process partial semantics at value v^k, noted $p\left(X\right)[v^k]$,*

are defined to be the morphisms making the following diagram commute:



[5] $[src1]$ and $[src2]$ denote memory spaces as explained in Example 1.

Example 5. The semantics of the path X_1 in Listing 1.3 is a morphism[6]:

$$s\left(X_1\right) : \left(\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]\right) \times e \rightarrow \left(\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]\right) \times \left[h\right]$$

and its process and kernel semantics are morphisms:

$$p\left(X_1\right) : \left(\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]\right) \times e \longrightarrow \left[src1\right] \cup \left[src2\right] \cup \left[dst\right]$$
$$k\left(X_1\right) : \left(\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]\right) \times e \longrightarrow \left[h\right]$$

The following theorem shows that if we first find an intermediate path $X_{1\text{-}2}$ having just the same kernel semantics as X_1 (i.e. the same effects on the kernel memory space), then we can later modify $X_{1\text{-}2}$ (while keeping its observable behaviors) to obtain X_2 having the same paths semantics as X_1.

Theorem 1 (Camouflage). *Let $X_1 \in \mathcal{X}$ and $v^p \times v^k \in source\left(s\left(X_1\right)\right)$, for each $X_{1\text{-}2} \in \mathcal{X}$ such that $p(X_{1\text{-}2})[v^k]$ is monic (i.e. injective) and:*

$$k\left(X_{1\text{-}2}\right)\left(v^p \times v^k\right) = k\left(X_1\right)\left(v^p \times v^k\right),$$

there exists $X_2 \in \mathcal{X}$ satisfying $obs\left(X_2\right) = obs\left(X_{1\text{-}2}\right)$ and:

$$s\left(X_2\right)\left(v^p \times v^k\right) = s\left(X_1\right)\left(v^p \times v^k\right).$$

In other words, if X_1 is a malicious path and $X_{1\text{-}2}$ (possibly benign) has the same kernel semantics, then we can build X_2 so that $s\left(X_2\right)\left(v^p \times v^k\right) = s\left(X_1\right)\left(v^p \times v^k\right)$, namely X_2 is also malicious; but $obs\left(X_2\right) = obs\left(X_{1\text{-}2}\right)$, namely it looks like a benign path.

Example 6. The paths X_1 and $X_{1\text{-}2}$ in Listings 1.3 and 1.5 have the same partial kernel semantics at the particular values `"\SYSTEM\CurrentControlSet\..."` $\cup$ $\ldots \cup \ldots$ of $\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]$ but the process partial semantics are not (values on $\left[src1\right] \cup \left[src2\right] \cup \left[dst\right]$ are set to 0 in $X_{1\text{-}2}$). Consequently, it satisfies the hypothesis of Theorem 1 and we can generate a path having the same semantics as the one in Listing 1.3 and the same observations as the one in Listing 1.5.

Listing 1.5: $X_{1\text{-}2}$

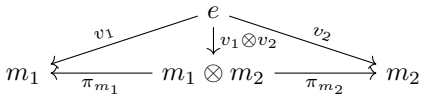
```
NtOpenKey(h,..."\SYSTEM\CurrentControlSet\..."...);
memset(dst,0,1024);
memset(src1,0,1024);
memset(src2,0,1024);
```

3.3 Obfuscated path generation

As previously mentioned, the proof of Theorem 1 will allow us to generate paths with camouflaged behaviors through a procedure called *path replaying*. The intuition behind such a procedure is to transform a path X_1 by specializing some invoked values inside the process memory space m^p. For that purpose, the projection morphisms allowing us to extract the partial values of a given total value are introduced:

[6] $\left[src1\right], \left[src2\right], \left[dst\right]$ and $\left[h\right]$ still denote memory spaces.

Definition 6 (Projection morphisms). *Let $v \in m_1 \otimes m_2$, the partial values $v_1 \in m_1$ and $v_2 \in m_2$ are respectively defined by the morphisms π_{m_1} and π_{m_2} making the following diagram commute:*



Given an execution path $X = [s_1^{j_1}, s_2^{j_2}, \ldots, s_n^{j_n}]$ and a value $v^p \times v^k \in source\,(s\,(X))$; for $1 \leq i \leq n$, define X_l to be the path containing the first l morphisms of X, i.e. $X_l = [s_1^{j_1}, s_2^{j_2}, \ldots, s_l^{j_l}]$. Consider a morphism $s_l^{p\text{-}k} \in obs(X)$, the source value $v_l^p \in m^p$ invoked by $s_l^{p\text{-}k}$ during the execution corresponding to path X can be computed by:

$$v_l^p = \begin{cases} p\,(X_{l-1})\left(\pi_{source(s(X_{l-1}))}\left(v^p \times v^k\right)\right) & \text{if } l > 1 \\ v^p & \text{otherwise} \end{cases}$$

Definition 7 (Replay path). *Given an execution path $X = [s_1^{j_1}, s_2^{j_2}, \ldots, s_n^{j_n}]$ and a value $v^p \times v^k \in source\,(s\,(X))$, the* replay path *$rep(X) = [r_1, r_2, \ldots, r_n]$ of X at $v^p \times v^k$ is defined by:*

$$r_i = \begin{cases} 1_{m^p} \times k(s_i^{j_i})[v_i^p] & \text{if } s_i^{j_i} \in obs(X) \\ s_i^{j_i} & \text{otherwise} \end{cases}$$

Example 7. The path in Listing 1.6 is the replay of the path in Listing 1.4 at value `"\SYSTEM\CurrentControlSet\..."` of $[dst]$.

Listing 1.6: Replay $rep(X_2)$ of X_2

```
memcpy(src2,src1,1024);
strncpy(dst+2,src2,15);
strncpy(dst+17,src1+15,25);
NtOpenKey(h,..."\SYSTEM\CurrentControlSet\..."...);
```

Now we can state the following result:

Proposition 3. *Given an execution path $X_1 \in \mathcal{X}$, let $rep\,(X_1)$ be the replay of X_1 at values $v^p \times v^k \in source\,(s\,(X_1))$. For each $X_{1\text{-}2} \in \mathcal{X}$ satisfying $s\,(X_{1\text{-}2}) = s\,(obs\,(rep\,(X_1)))$, $p\,(X_{1\text{-}2})\,[v^k]$ is monic and the following properties hold:*

- $k(X_{1\text{-}2})\left(v^p \times v^k\right) = k(X_1)\left(v^p \times v^k\right)$,
- *if $X_2 = X_{1\text{-}2}@[p(X_1)[v^k]]$, where @ is the usual concatenation operator on lists, then:*

$$s\,(X_2)\left(v^p \times v^k\right) = s\,(X_1)\left(v^p \times v^k\right).$$

The former shows that $X_{1\text{-}2}$ satisfies the assumptions of Theorem 1 while the later explicitly builds an execution path X_2 such that $obs\,(X_2)$ *behaviorally obfuscates* $obs\,(X_1)$. Indeed, since $rep\,(X_1)$ is constructed out of X_1 by replacing each $s_i^{p\text{-}k} \in obs\,(X_1)$ by $1_{m^p} \times k(s_i^{p\text{-}k})[v_i^p]$, the observable paths $obs\,(rep\,(X_1))$ and $obs\,(X_1)$ have the shapes:

$$obs\,(X_1) = [s_i^{p\text{-}k}, \ldots, s_l^{p\text{-}k}]$$
$$obs\,(rep\,(X_1)) = [1_{m^p} \times k(s_i^{p\text{-}k})[v_i^p], \ldots, 1_{m^p} \times k(s_l^{p\text{-}k})[v_l^p]]$$

Proposition 3 provides a straightforward way of generating an obfuscated path X_2 of X_1 by setting:

$$X_2 = X_{1\text{-}2}@[p(X_1)[v^k]]$$

for some $X_{1\text{-}2}$ such that $X_{1\text{-}2} = obs\,(rep\,(X_1))$. The obtained path X_2 complies with $obs\,(X_2) = obs\,(rep\,(X_1))$ and $obs\,(X_2) \neq obs\,(X_1)$.

3.4 Graph-based path transformation

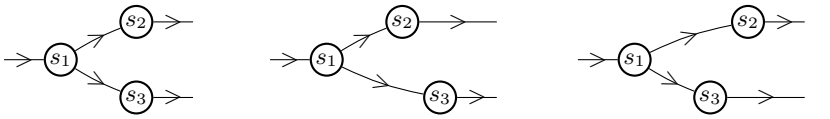
Though having distinct syscall invocations, the replay paths obtained in previous section are "not that different" in the sense that involved syscall names are still identical (see Example 7). The general objective of this subsection is to show how to generate paths that are semantically equivalent to $obs\,(rep\,(X_1))$ but with distinct observations. For that purpose, the string diagram formalism induced by the considered monoidal categories and introduced at the end of Subsection 2.1 will be used throughout the remainder of this section in order to consider *semantics-preserving transformations* on the syscall invocations in $obs\,(rep\,(X_1))$.
By Proposition 2, the path semantics $s\,(obs\,(rep\,(X_1)))$ is represented by a morphism term constructed from the morphisms $1_{m^p} \times k(s_i^{p\text{-}k})[v_i^p]$. Hence, by Proposition 1, it has a graphical representation as a string diagram, moreover we can safely omit the identity morphism 1_{m^p} in considering these morphisms.

Among string diagrams, we will only consider **path diagrams**, namely the diagrams such that the projection of nodes on an horizontal axis is an injective function, so the projection allows us to define a total order on nodes. The reason for restricting our graphical representation to path diagrams is that they represent their corresponding paths in an unambiguous way.

Example 8. Consider the three following string diagrams: The string diagrams (b) and (c) are path diagrams representing the paths $[s_1, s_2, s_3]$ and $[s_1, s_3, s_2]$, respectively, but the string diagram (a) is not a path diagram.

The following theorem on coherence of progressive plane diagrams, when applied to the corresponding string (or path) diagrams, gives us a sound property on semantics-preserving transformations from one path to another.

Theorem 2 ([14,22]). *In monoidal categories, morphism terms equivalence can be deduced from axioms iff their corresponding string diagrams are planar isotopic.*



(a) $(s_2 \otimes s_3) \circ s_1$ (b) $(1 \otimes s_3) \circ (s_2 \otimes 1) \circ s_1$ (c) $(s_2 \otimes 1) \circ (1 \otimes s_3) \circ s_1$

Fig. 3: String diagrams

Following [22], we accept an informal definition of *planar isotopy* between string diagrams as "...one can be transformed to the other by continuously moving around nodes..." (but keep the diagram always progressive), the formal treatment can be referenced in [14], e.g. The three string diagrams of Example 8 are planar isotopic.

Between path diagrams, planar isotopy can be though of as moving the nodes but keeping the total oder compatible with the partial order $\preccurlyeq$ (see Section 2.1). Hence, a *linear extension* Y of $obs\left(rep\left(X_1\right)\right)$, namely a permutation where the total order remains to be compatible with $\preccurlyeq$, will preserves the semantics of $obs\left(rep\left(X_1\right)\right)$. This leads to the following Algorithm:

Input: an observable path $obs\left(rep\left(X_1\right)\right)$
Output: a permutation Y satisfying $s\left(Y\right) = s\left(obs\left(rep\left(X_1\right)\right)\right)$

begin
$M_1 \leftarrow$ a morphism term of $s\left(obs\left(rep\left(X_1\right)\right)\right)$;
$G_1 \leftarrow$ a string diagram of M_1;
$\left(obs\left(rep\left(X_1\right)\right), \preccurlyeq\right) \leftarrow$ a poset with order induced from G_1;
$\left(Y, \leq\right) \leftarrow$ a linear extension of $\left(obs\left(rep\left(X_1\right)\right), \preccurlyeq\right)$;
end

Algorithm 1: Obfuscation by diagram deformation

Example 9. Consider the below listings corresponding to execution paths X_3 and X_4. They can be respectively represented by path diagrams (b) and (c). Consequently, given X_3, Algorithm 1 can generate X_4 (or the converse).

Listing 1.7: X_3

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...); /*s1*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"abc"); /*s2*/
NtSetValueKey(h,...{..."URL"...}...,"xyz"); /*s3*/
```

Listing 1.8: X_4

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...); /*s1*/
NtSetValueKey(h,...{..."URL"...}...,"xyz"); /*s3*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"abc"); /*s2*/
```

A variable in a morphism term (resp. a node in the string diagram) is also a *placeholder* [22] that can be substituted by another term (resp. another diagram) having the same semantics, so the below Algorithm can be derived from Algorithm 1:

Input: an observable path $obs\,(rep\,(X_1))$
Output: a new path Y satisfying $s\,(Y) = s\,(obs\,(rep\,(X_1)))$

begin
$M_1 \leftarrow$ a morphism term of $obs\,(rep\,(X_1))$;
$s \leftarrow$ a morphism of M_1;
$X \leftarrow$ an execution path satisfying $s(X) = s$;
$M \leftarrow$ a morphism term of X;
$M_2 \leftarrow$ the morphism term $M_1\{M/s\}$;
$G_2 \leftarrow$ a string diagram of M_2;
$((obs\,(rep\,(X_1)) \setminus s) \cup X, \preccurlyeq) \leftarrow$ poset with order induced from G_2;
$(Y, \leq) \leftarrow$ a linear extension of $((obs\,(rep\,(X_1)) \setminus s) \cup X, \preccurlyeq)$
end

Algorithm 2: Obfuscation by node replacement

Example 10. Consider the replacement of s_2 in Listing 1.7 by $X = [s'_2, s_2]$ where:

$$s'_2 = \texttt{NtSetValueKey}(\texttt{h}, \ldots \{\ldots \texttt{"DOWNLOAD"} \ldots\} \ldots, \texttt{"a'b'c'"})$$

Using this replacement, given the execution path X_3 in Listing 1.7, Algorithm 2 can generate the execution path X_5 corresponding to the following listing:

Listing 1.9: X_5

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...); /*s1*/
NtSetValueKey(h,...{..."DOWNLOAD"}...,"a'b'c'");/*s'2*/
NtSetValueKey(h,...{..."URL"...}...,"xyz"); /*s3*/
NtSetValueKey(h,...{..."DOWNLOAD"}...,"abc"); /*s2*/
```

Proposition 4 (Soundness). *Given an execution path X_1, Algorithms 1 and 2 generate an observable path Y as output that behaviorally obfuscates $obs(rep(X_1))$ (provided that considered the linear extension is not the identity).*

4 Experiments and detection

4.1 Experimental implementation

Algorithms 1 and 2 have been applied to several sub-paths extracted from the malwares `Dropper.Win32.Dorgam` [3] and `Gen:Variant.Barys.159` [1]. The programs (written in C++ and Haskell) use Pin [13] for path tracing and FGL [9]

for path transforming. The implemented pieces of code are available at the repository [2].

In the following experiments, the string diagrams of paths are illustrated as follows: The numbers appearing as node labels represent the total order in the original path. In each diagram, the fictitious nodes `Input` and `Output` are added as the *minimum* and the *maximum* in such a way that the path can be considered as a lattice. On a fixed line, the number appearing as edge labels represent the handlers which identify the corresponding memory space inside kernel. On different lines, the same numbers may identify different memory spaces. The obfuscated paths generated by Algorithm 1 are linear extensions which are compatible with the order defined in the lattice. Note that their corresponding diagrams are always path diagrams (but they are not illustrated here).

Experiment 1 The trojan `Dropper.Win32.Dorgam` has an execution path X corresponding to the trace in Figure 2 that consists in 24 morphims. Let $[h_i], i \in \{1, 4, 7, 10, 13,\ 16, 19, 22\}$ denote the memory spaces identified by the handlers of the accessed registry keys (i being the listing line number), the replay path is formulated by morphisms:

$$\begin{aligned} NtOpenKey_i^{p\text{-}k} &: e \to [h_i] \\ NtSetValueKey_{i+1}^{p\text{-}k} &: [h_i] \to [h_i] \\ NtClose_{i+2}^{p\text{-}k} &: [h_i] \to e \end{aligned}$$

Its string diagram is represented in Figure 4.

The number $e(X)$ of possible linear extensions of X is computed by:

$$e(X) = \binom{24}{3}\binom{21}{3}\binom{18}{3}\binom{15}{3}\binom{12}{3}\binom{9}{3}\binom{6}{3}\binom{3}{3} = \frac{24!}{6^8}$$

namely more than 369 quadrillion extensions (and paths) can be generated by Algorithm 1.

Experiment 2 The trojan also uses the trace in Listing 1.10 to create a copy of `iexplore.exe`, its replay path has the string diagram provided in Figure 5(a).

Listing 1.10: File copying

```
NtCreateFile (FileHdl=>0x00000730,File<=\??\C:\Program Files\
Internet Explorer\IEXPLORE.EXE)
NtCreateFile (FileHdl=>0x0000072C,File<=\??\C:\Program
Files\iexplore.exe)
NtReadFile   (FileHdl<=0x00000730,BuffAddr<=0x0015C898, ByteNum<=65536)
NtWriteFile  (FileHdl<=0x0000072C, BuffAddr<=0x0015C898, ByteNum<=65536)
......
NtReadFile   (FileHdl<=0x00000730,BuffAddr<=0x0015C898, ByteNum<=65536)
NtWriteFile  (FileHdl<=0x0000072C, BuffAddr<=0x0015C898, ByteNum<=48992)
NtReadFile   (FileHdl<=0x00000730, BuffAddr<=0x0015C898, ByteNum<=65536)
NtClose      (Hdl<=0x00000730)
NtClose      (Hdl<=0x0000072C)
```

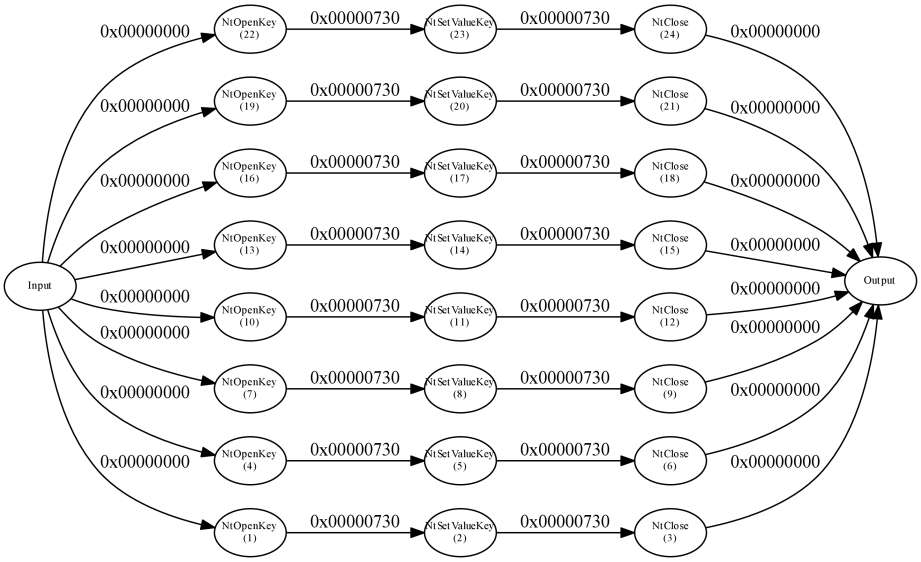


Fig. 4: Registry initializing string diagram

This path can be considered as an obfuscated version (generated by Algorithm 2) of the path whose string diagram is in Figure 5(b), by considering the equivalences:

$$NtReadFile^{p\text{-}k}_{3(orig)} = [NtReadFile^{p\text{-}k}_{3}, NtReadFile^{p\text{-}k}_{5}, \dots]$$
$$NtWriteFile^{p\text{-}k}_{4(orig)} = [NtWriteFile^{p\text{-}k}_{4}, NtWriteFile^{p\text{-}k}_{6}, \dots]$$

It also means that a *behavior matching* detector can detect an obfuscated path, assuming the prior knowledge of both the original path and the semantics equivalences described above.

Experiment 3 We consider the ransomware `Gen:Variant.Barys.159` [1]. The extracted path in Listing 1.11 explains how the malware conceals itself by injecting code into file explorer process `explorer.exe`.

Listing 1.11: Code injecting

```
NtOpenProcess(ProcHdl=>0x00000780,DesiredAccess<=1080,ProcId<=0x00000240)
NtCreateSection(SecHdl=>0x00000778,AllocAttrs<=SEC_COMMIT,FileHdl<=0x00000000)
NtMapViewOfSection(SecHdl<=0x00000778,ProcHdl<=0xFFFFFFFF,BaseAddr<=0x02660000)
NtReadVirtualMemory(ProcHdl<=0x00000780,BaseAddr<=0x7C900000,BuffAddr<=0x026
60000,ByteNum<=729088)
NtMapViewOfSection(SecHdl<=0x00000778,ProcHdl<=0x00000780,BaseAddr<=0x7C900000)
```

The malware first obtains the handler `0x780` from the running instance (whose process id is `0x240`) of `explorer.exe` and then creates a section object identified by the handler `0x778`. It maps this section to the malware memory, it copies some data of the instance into the mapped memory, it performs data

modification on this memory and, finally, it maps the section (now contains modified data) back to the instance.

Let $[h_1], [h_2]$ denote the memories identified by handlers of the opened process and of the created section, the replay path is formulated by morphisms:

$$NtOpenProcess_1^{p\text{-}k} : e \to [h_1]$$
$$NtCreateSection_2^{p\text{-}k} : e \to [h_2]$$
$$NtMapViewOfSection_3^{p\text{-}k} : [h_2] \to [h_2]$$
$$NtReadVirtualMemory_4^{p\text{-}k} : [h_1] \to [h_1]$$
$$NtMapViewOfSection_5^{p\text{-}k} : [h_1] \cup [h_2] \to [h_1] \cup [h_2]$$

and the corresponding string diagram is provided in Figure 5(c).

If the morphism $NtReadVirtualMemory_4^{p\text{-}k}$ is replaced by a path $[s_{4\text{-}1}^{p\text{-}k}, s_{4\text{-}2}^{p\text{-}k}]$ corresponding to the syscall invocations in Listing 1.12 then this replacement leads to the string diagram in Figure 5(d).

Listing 1.12: NtReadVirtualMemory

```
NtReadVirtualMemory(ProcHdl<=0x00000780,BaseAddr<=0x7C900000,BuffAddr<=0x026
60000,ByteNum<=9088)
NtReadVirtualMemory(ProcHdl<=0x00000780,BaseAddr<=0x7C909088,BuffAddr<=0x026
69088,ByteNum<=72000)
```

The numbers of linear extensions for the original string diagram and the obfuscated one are respectively:

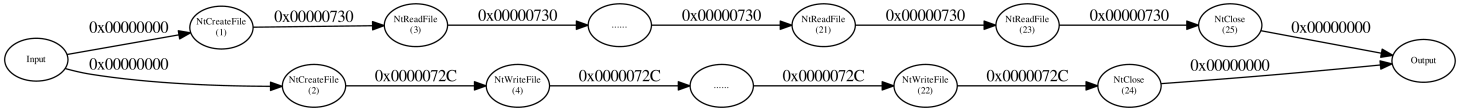
$$e_1(X) = \binom{4}{2}\binom{2}{2} = 6 \qquad e_2(X) = \binom{5}{3}\binom{2}{2} = 10$$

4.2 Obfuscated path detection

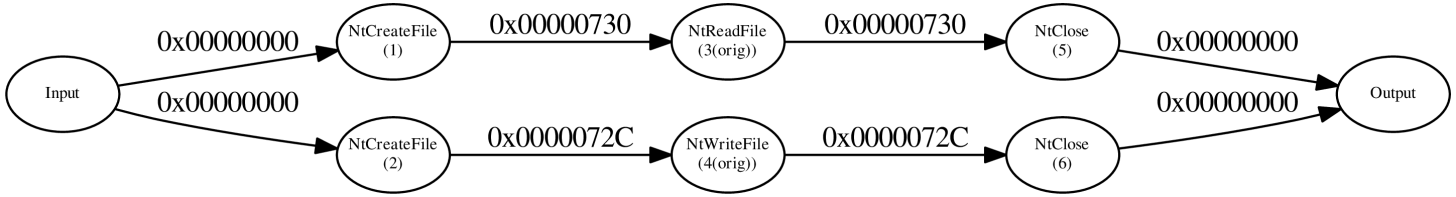
We will now discuss the detection by using practical detectors introduced in previous existing works on *behavior matching* [7,15,12].

Basically, a behavior matching detector first represents an observable path by a directed acyclic graph (DAG) using the causal dependency between morphisms, a morphism s_j (directly or indirectly) depends on s_i if the sources values of s_j are (directly or indirectly) deduced from the target values of s_i. Then the detector decides a path is malicious or not by verifying whether there exists a malicious pattern occurring as a subgraph of the original DAG. Here the malicious pattern is a (sub-)DAG and it can be obfuscated to another semantics equivalent DAG.

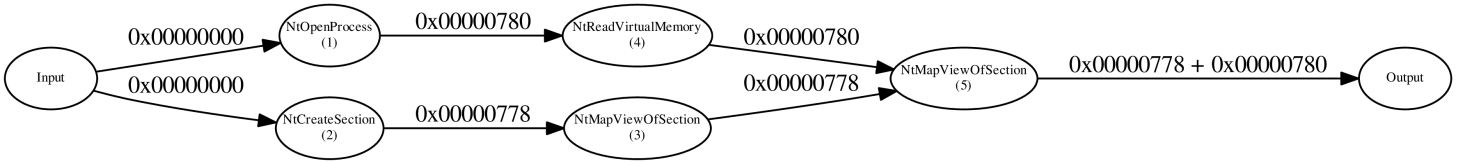
Whereas Algorithm 1 can generate a large amount of paths, the verification of whether an obfuscated path is semantically equivalent to the original path is simple: it is an instance of the *DAG automorphism* problem where every vertex is mapped to itself. The instance can be decided in *P-time* by repeatedly verifying whether two paths have the same set of minimal elements, if they do then remove the set of minimal elements from both paths and repeat; if they do not then decide **No**; if the sets are both empty then decide **Yes** and stop.



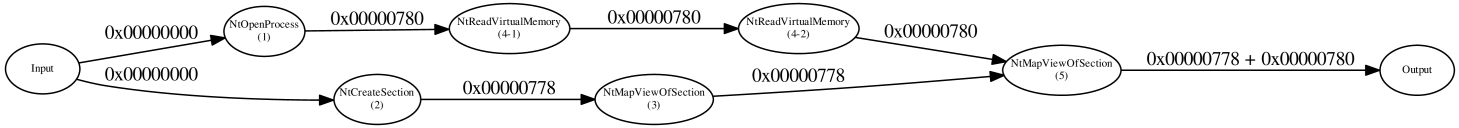
(a) File copying string diagram



(b) File copying original string diagram



(c) Code injecting string diagram



(d) Code injecting obfuscated string diagram

Fig. 5: Experiments string diagrams

The detection of obfuscated paths generated by Algorithm 2 is more challenging. When applied naively, the behavior matching does not work since the algorithm can generate paths of morphisms corresponding to syscall names and invocations distinct from those of the original path. More generally, it may be nonsense to compare an obfuscator and a detector which use different sets of behavioral transformations. In other words, as discussed in [19], a detector which abstracts behaviors by using the transformation set T, will be bypassed by an obfuscator which generates behaviors by using a set T' so that $T \cap T' \neq \emptyset$.

The original *behavior matching* techniques can be strengthened by generating (e.g. by using Algorithm 2) in prior a set of patterns that are semantically equivalent to the original one (see also the discussion in Experiment 2). Conversely, that means simplifying obfuscated paths to their original unique form, several simplifying techniques has been studied in some existing works on *semantics rewriting* (e.g. [5]). So we might suggest that a combination of *behavior matching* and *semantics rewriting* will improve the presented analysis. We reserve such an improvement as a future work.

References

1. Gen:Variant.Barys.159. 2013. http://goo.gl/YDC1o.
2. Trace Transformation Tool. 2013. http://goo.gl/rqCSQ.
3. Trojan-Dropper.Win32.Dorgam.un. 2013. http://goo.gl/3e1AR.
4. S. Awodey. *Category Theory (Oxford Logic Guides)*, volume 49. Oxford University Press, USA, 2006.
5. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior Abstraction in Malware Analysis. *Runtime Verification*, pages 168–182, 2010.
6. M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. *Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.
7. M. Christodorescu, C. Kruegel, and S. Jha. Mining Specifications of Malicious Behavior. *ISEC*, pages 5–14. ACM, 2008.
8. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
9. M. Erwig. FGL/Haskell - A Functional Graph Library for Haskell. 2008. http://web.engr.oregonstate.edu/ erwig/fgl/haskell/.
10. E. Filiol. Formalisation and implementation aspects of k -ary (malicious) codes. *Journal in Computer Virology*, 3(2):75–86, 2007.
11. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. *Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.
12. M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. *Symposium on Security and Privacy*, pages 45–60. IEEE, 2010.
13. Intel. Pin - A Dynamic Binary Instrumentation Tool. 2013. http://software.intel.com/en-us/articles/pintool.
14. A. Joyal and R. Street. The Geometry of Tensor Calculus, I. *Advances in Mathematics*, 88:55–112, 7 1991.
15. C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. USENIX Security, pages 351–366, 2009.

16. W. Ma, P. Duan, S. Liu, G. Gu, and J-C. Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1-2):1–13, 2012.
17. J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, 88(2):105–155, October 1990.
18. R. Milner. Action Structures. Technical report, 1992. http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-249/.
19. M. Dalla Preda. The grand challenge in metamorphic analysis. *ICISTM*, volume 285 of *Communications in Computer and Information Science*, pages 439–444. Springer, 2012.
20. M. Dalla Preda, M. Christodorescu, S. Jha, and S. K. Debray. A semantics-based approach to malware detection. *POPL*, pages 377–388. ACM, 2007.
21. M. Ramilli and M. Bishop. Multi-Stage Delivery of Malware. MALWARE, pages 91–97, 2010.
22. P. Selinger. A survey of graphical languages for monoidal categories. *New structures for physics*, pages 289–355. Springer, 2011.
23. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. *Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.