A Categorical Treatment of Malicious Behavioral Obfuscation

Romain Péchoux and Thanh Dinh Ta

Université de Lorraine & Inria Grand-Est, LORIA, Nancy, France

TAMC 2014

Traditional malware's writers techniques

Use of program transformation to bypass malwares detectors:

- Useless code injection,
- Function call order change,
- Code encryption, ...

In order to obtain a program having the same malicious behavior (semantically equivalent wrt some formal semantics).

Relative view of obfuscation

Obfuscation as information lost

$$\begin{tikzcd}
\mathcal{P} \arrow[r, "\mathcal{O}"] \arrow[d, "\mathfrak{A}"] & \mathcal{P} \arrow[d, "\mathfrak{A}"] \\
\mathit{Props} \arrow[r, "\supseteq"] & \mathit{Props}
\end{tikzcd}$$

- $\mathcal{P}$: set of programs,
- $\mathcal{O}$: obfuscation function,
- $\mathfrak{A}$: abstraction (or analysis) function,
- *Props*: set of interested properties,
- $\supseteq$: information ordering,

Code obfuscation

$$P \sim_{io} \mathcal{O}(P)$$

Semantic equivalence is undecidable (by Rice's Theorem). Consequently, detectors have to handle code obfuscation **conveniently** and with a good **tractability**. Current works:

- semantics-based detection [Christodorescu, Della Preda et al.]:

Program = Abstraction independent from code transformation

- behavior-based detection [Forrest, Kolbitsch et al.]:

Program abstraction = Observable behaviors

- Detection bypassing [Filiol, Wagner & Soto, ...]

Behavioral obfuscation

- only a few works
- a lack of formalism and general methods
- difficulty to handle new attacks
- the strength of a behavior-based detector might be overestimated (bad resilience to code obfuscation)

Consequently there is a strong need of:

- **high level** formalisms
- allowing to obtain **formal proofs** on malicious behaviors
- while keeping **practical** considerations in mind !

Trojan Dropper as Motivating example

The trojan `Dropper.Win32.Dorgam` works in 3 consecutive stages:

1. It unpacks 2 files whose paths are added in the registry value `AppInit_DLLs`.
2. It creates a key `SOFTWARE`/`AD` and adds some entries.
3. It calls the function `URLDownloadToFile` of MSIE to download malicious codes from some addresses in the stored values.

File unpacking at stage 1 and File downloading at stage 3 are too general to expect any behavior-based detection...

Motivating example

Stage 2: we look at the malware behaviors (syscalls):

NtOpenKey, NtSetValueKey, NtClose, NtOpenKey, ...

However each `NtOpenKey` syscall associated to each `NtSetValueKey` syscall is verbose:

NtOpenKey, NtSetValueKey, NtSetValueKey, ...

Moreover, the key handler can be obtained by duplicating a key handler located in another process, so the call `NtOpenKey` is not mandatory:

NtDuplicateObject, NtSetValueKey, NtSetValueKey, ...

Achievements

- We introduce an abstract model based on monoidal categories
  - where observable behaviors are morphisms
- we show the principle of obfuscation on such a model
  - we use semantics-preserving transformations on such model
- and show that they allow us to capture malwares in practice.

Interaction category

- A memory state $s : \mathcal{B} \rightarrow \{0,1\}$, with $\mathcal{B} \subseteq Adr$
- A memory space $m = \{s \mid dom(s) = B\}$

Interaction category

An interaction category consists in 2 memory spaces m^p, m^k s.t.:

- objects: $n^i \subseteq m^i$, $i \in \{k,p\}$, $n^p \times n^k$, e, ...
- morphisms: π_i, $s^i : n^i \rightarrow o^i$, $i \in \{p,k\}$, $s^{p-k} : n^p \times n^k \rightarrow o^p \times o^k$ (Syscall interactions)
- and with a tensor product $\otimes$ defined on objects by:

$$\begin{tikzcd}
m_1 \otimes m_2 \arrow[r, "s_1 \otimes 1_{m_2}"] \arrow[d, "1_{m_1} \otimes s_2"'] & n_1 \otimes m_2 \arrow[d, "1_{n_1} \otimes s_2"] \\
m_1 \otimes n_2 \arrow[r, "s_1 \otimes 1_{n_2}"] & n_1 \otimes n_2
\end{tikzcd}$$

N.B.: Each interaction category is a (partial) monoidal category.

Example: Function and Syscall

Process internal computation and syscall interaction

```
char *src = 0x00150500;
char *dst = 0x00150770;
strncpy(dst,src,10);
...
char *buf = 0x0015C898;
HANDLE hdl = 0x00000730;
NtWriteFile(hdl,...,buf,1024);
```

`strncpy` is represented by a process internal computation:

$$strncpy^{p} : [src] \otimes [dst] \longrightarrow [src] \otimes [dst],$$

`NtWriteFile` is represented by a syscall interaction:

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

Observable paths

Definition [Observable path]

- An execution path is a finite list of morphisms $X = [s_1^{j_1}, s_2^{j_2}, \ldots]$, with $j_i \in \{p, p\text{-}k\}$
- The observable path O of an execution path X consists in syscall interactions of X.
  Let *obs* be the mapping from X to O.

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

Its execution and observable paths are defined by:

$$X = [strncpy_1^p, strncpy_2^p, NtOpenKey_3^{p\text{-}k}, memcpy_4^p]$$

$$O = [NtOpenKey_3^{p\text{-}k}] = obs(X)$$

Obfuscation

Definition [Semantics]

The path, kernel and process semantics of X, $s(X), k(X)$ and $p(X)$ resp. are the morphisms making the following diagram commute:

$$\begin{tikzcd}
& m^p \times n^k \arrow[dl, "p(X)"'] \arrow[d, "s(X)"] \arrow[dr, "k(X)"] & \\
m^p & m^p \times o^k \arrow[l, "\pi_p"'] \arrow[r, "\pi_k"] & o^k
\end{tikzcd}$$

Definition [Behavioral obfuscation]

X_2 **obfuscates** X_1 if:

- $s\left(X_2\right)\left(v_0^p \times v_0^k\right) = s\left(X_1\right)\left(v_0^p \times v_0^k\right)$
- and $obs(X_1) \neq obs(X_2)$.

Obfuscation Theorem

Theorem [Camouflage]

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

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

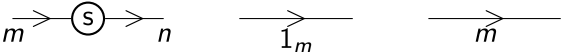
there exists $X_2 \in \mathcal{X}$ satisfying $\mathit{obs}\left(X_2\right) = \mathit{obs}\left(X_{1\text{-}2}\right)$ and:

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

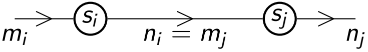
- proved using path replaying techniques:
  - replay= path with same kernel effect, distinct observations
- Can we use the categorical abstraction a bit further ?

Graphical representation

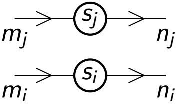
- nodes are morphisms and edges are objects:



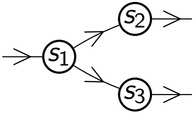
- composition $s_j \circ s_i$:



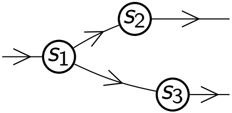
- tensor product $s_i \otimes s_j$:



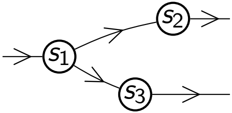
Path diagrams



(a) $(s_2 \otimes s_3) \circ s_1$



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



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

The string diagrams (b) and (c) are path diagrams but the string diagram (a) is not.

Theorem [cf. Joyal-Street]

In monoidal category, term equivalence can be deduced from axioms iff the corresponding string diagrams are planar isotopic.

Obfuscation by diagram deformation

```
Input: an observable path $obs(rep(X_1))$
Output: a permutation $Y$ satisfying $s(Y) = s(obs(rep(X_1)))$
begin
    $M_1 \leftarrow$ a morphism term of $s(obs(rep(X_1)))$;
    $G_1 \leftarrow$ a string diagram of $M_1$;
    $(obs(rep(X_1)), \preccurlyeq) \leftarrow$ a poset with order induced from $G_1$;
    $(Y, \leq) \leftarrow$ a linear extension of $(obs(rep(X_1)), \preccurlyeq)$;
end
```

Obfuscation by node replacement

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 induced by $G_2$;
    $(Y, \leq) \leftarrow$ a linear extension of $((obs\,(rep\,(X_1)) \setminus s) \cup X, \preccurlyeq)$
end
```

Obfuscation by node replacement

```
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 induced by $G_2$;
    $(Y, \leq) \leftarrow$ a linear extension of $((obs(rep(X_1)) \setminus s) \cup X, \preccurlyeq)$
end
```

Detection and conclusion

- Algorithms 1 and 2 have been written in C++ and Haskell using Pin (path tracing) and FGL (path transforming).
- They manage to capture obfuscated variants of well-known malwares including:
    - `Dropper.Win32.Dorgam`
    - `Gen:Variant.barys.159`
- Verifying whether a path is equivalent to a path generated by Algorithm 1 is tractable in polynomial time (an instance of DAG automorphism problem).
- Algorithm 2 is more challenging and needs to use semantics rewriting techniques to capture more obfuscated versions (left as future work).