# Malware Message Analysis through Binary Traces

Guillaume Bonfante* Jean-Yves Marion* Thanh Dinh Ta*
*Lorraine University
CNRS - LORIA

***Abstract*—Many malware trigger their malicious behaviors only if they detect some particular conditions from the environment. Under normal conditions these behaviors are hidden deeply under obfuscated codes, but they may be activated at any time by a Command and Control (C and C) server. However, connections to the C and C are not completely stealthy. We propose to analyse the way messages are operated by malware to characterize their behavior.**

## I. Introduction

Few times ago, we have been given some samples of the Reveton botnet, a famous trojan which could lock computers, steal passwords, and so on. We observed similarities between the samples, but some of them have different behaviors. However, as far as we understand it, they share the same protocol with respect to the Command and Control server. We present here the techniques that we have developed to verify that fact. We have two side (but important) objectives.

First, as we shall see, protocol analysis asks to perform an extensive code coverage to recover the complete protocol. Doing so, we actually, solve an other problem. Malware are known to use trigger conditions to hide their malicious behaviors, this technique protect them both from static and dynamic analysis. The static approach is almost unpracticable for malware analysis. Indeed, the largest part of malware are encrypted, obfuscated, self-modifying. For instance, Moser showed in [3] that malware are designed so that statically resolving them is very hard. But, dynamic analysis is either problematic. Usually, the malware is self-protected against debugging and more generally code supervision. It is then crucial to ensure a sufficiently correct code coverage to get back the real behavior of the malware.

Second, we want to establish that the analysis of messages reveals the behavior of programs. Behavioral characterization of malware though attracting in theory, remains difficult in facts. Usually, either false positive, either false negative, but one of them is too high. Here, our view is not to consider directly the behavior of the program, but to observe some of its effects: we discriminate programs by the way they are managing input (supposedly from the C and C) messages.

In this contribution, we suppose that we have only access to the binary code of the malware, not to its source code. Furthermore, we suppose that it cannot be statically analysed. Thus, we propose to work by a static analysis of malware code execution traces. Last game of the rules, we suppose that we have only access to *one* real trace execution. Indeed, launching a malware may be observed by the C and C. To keep the malware analysis secret, we forbid testing the malware too many times.

The main issue with traces is that they can be very long, millions of instructions. Standard techniques of code coverage, for instance the ones described by Godefroid [1] cannot be performed as easily on such programs. There is need for clever techniques, and a strong infrastructure.

The approach presented in this contribution uses ideas of *trace transformation*: before analyzing a trace, we transform it into a simpler one. This is done by a tainting technique on assembly instructions, thus involving a semantics of the X86 code instruction set. In the context of code coverage, the transformation is sound in the sense that if a branch in the transformed trace takes different decisions when running with some different inputs then so it does on the original one. In summary, our contributions are:

- We present a sound transformation so that the analysis can switch from the original trace to a more simple one,
- We apply the transformation to improve a fuzz testing procedure,
- We present an experimental implementation using the Pin [2] DBI toolkit.

## II. Code coverage

An *instruction* is the minimal execution unit of the program. Instructions are characterized by an *address* and a *command*, noted as a pair $\langle addr : cmd \rangle$. The address $addr$ denotes the position in the program's memory space. At loading time, the program gets its *static form*, that is a sequence of piece-wise consecutive instructions in memory.

The *instruction pointer* (abbr. IP) is a register determining which instruction will be executed next. For many instructions, their execution shifts deterministically the IP to their succeeding instruction as determined in the program's static form: we qualify them as sequential. Others (e.g. `call`, `ret`, `jmp`, `jne`) may divert the IP to more or less arbitrary address: we call them branching instructions (or branches). In what follows, the *conditional branches* (abbr. $jcc$) are considered as representative of branches.

An *execution trace* (or trace) is a list of successively executed instructions when the program runs:

$$tr = [ins_1, ins_2, \ldots, ins_k \ldots]$$

and an *execution point* (abbr. point) $i$ means that IP points to $ins_i$. Let $[\![P]\!](m)$ denote the trace obtained from running the program $P$ with the input $m$.

## A. Branch resolver and code coverage

The treatment of the code coverage is in the same vein with one in [1], it begins with a branch resolver:

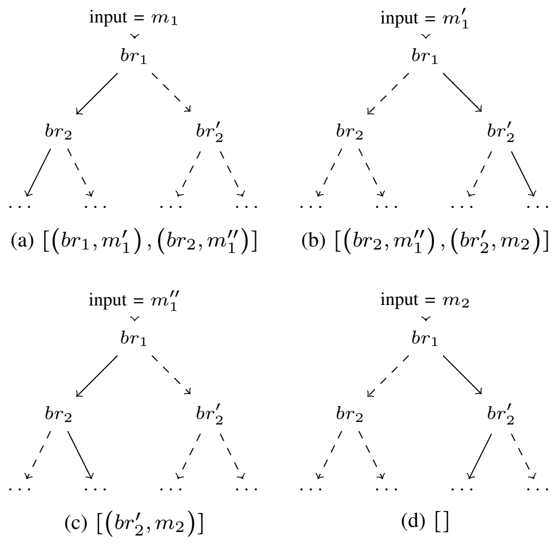
**Problem 1** (Branch resolving)**.** *Let $P$ be a program, $M$ be the set of inputs of $P$. Given $m \in M$ and the trace:*

$$[\![P]\!]\,(m) = tr_1{+}{+}[b, c, \dots]$$

*where $tr_1$ is finite and $b$ is a jcc. Find $m' \in M$ so that:*

$$[\![P]\!]\,(m') = tr_1{+}{+}[b, c', \dots] \text{ where } c' \neq c$$

Applying branch resolution several times, one gets a tree of traces covering behaviors of the program.



(a) $[(br_1, m_1'), (br_2, m_1'')]$ (b) $[(br_2, m_1''), (br_2', m_2)]$

(c) $[(br_2', m_2)]$ (d) $[]$

Figure 1: Code coverage

So from now on, we can focus in the branch resolving stated in Problem 1.

## B. Data flow analysis and trace transformation

Consider the following program. It reads one byte input buffer located at the address `0x601174`. For the illustration purpose, suppose that the input value is `0x68`, in that case, the `jcc1` does not take and the function at `0x400987` is called, after the function returns, the `jcc2` will be reached.

```
0x400974: movsx eax, byte ptr [0x601174]
0x40097c: cmp eax, 0x68
0x400981: jnle 0x4009b3    /* jcc1 */
0x400987: call 0x400830
0x40098c: movsx eax, byte ptr [0x601174]
0x400994: cmp eax, 97
0x400999: jl 0x4009a9      /* jcc2 */
```

Observe that the decision of `jcc2` is independent from the results of the function so the program can be "patched" as

```
0x400974: movsx eax, byte ptr [0x601174]
0x40097c: cmp eax, 0x68
0x400981: jnle 0x4009b3    /* jcc1 */
0x400987: nop
...
0x40098b: nop
0x40098c: movsx eax, byte ptr [0x601174]
0x400994: cmp eax, 97
0x400999: jl 0x4009a9      /* jcc2 */
```

To conclude, we perform an instrumented analysis of binary codes, but on dynamically optimized codes. Discarding unnecessary code is done via a tainting verification. We build a data flow graph linking conditional jumps to the pieces of input memory they depend on. Each binary instruction relate some inputs and outputs according to its semantics as a hyper-graph, tainting is obtained then by glueing these graphs together.

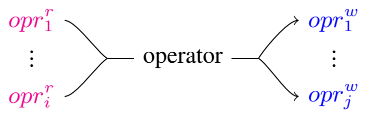


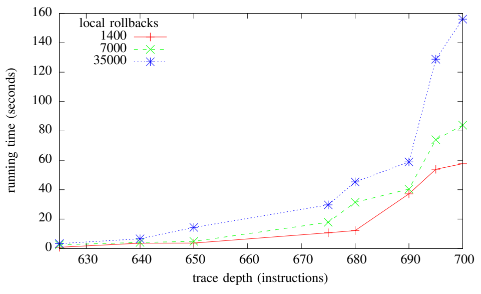
Figure 2: Hyper-graph of an instruction

## C. Branch resolver by fuzzing

We solve branches running an input fuzzier. Naturally, the fuzzier take into account tainting. We have observed that many branching instruction depend only on few bits of the inputs. This explains why we get relatively good results with such a rough technique.

Table I: Fuzz testing results on `wget`

| trace depth | local rollbacks | resolved/de-tected branches | used rollbacks | run-ning time(s) |
|---|---|---|---|---|
| 650 | 1400 | 18/39 | 42163 | 3.7 |
| 650 | 7000 | 20/50 | 275016 | 4.8 |
| 650 | 35000 | 38/87 | 2341715 | 14.3 |
| 675 | 1400 | 45/110 | 123350 | 10.7 |
| 675 | 7000 | 64/157 | 935366 | 17.8 |
| 675 | 7000 | 108/262 | 1496531 | 29.7 |
| 700 | 1400 | 212/558 | 723114 | 57.7 |
| 700 | 7000 | 247/744 | 5684159 | 83.9 |
| 700 | 35000 | 257/767 | 29708198 | 156.1 |



## References

[1] P. Godefroid et al. "DART: Directed Automated Random Testing". In: PLDI. 2005.

[2] C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: PLDI. 2005.

[3] A. Moser et al. "Limits of Static Analysis for Malware Detection". In: ACSAC. 2007.