Malware Message Classification by Dynamic Analysis

Guillaume Bonfante[1,2] Jean-Yves Marion[1,2] Thanh Dinh Ta[1,2]

[1] Lorraine University
[2] CNRS - LORIA

Abstract. The fact that new malware appear every day demands a strong response from anti-malware forces. For that sake, an analysis of new samples must be performed. Usually, one tries to replay the behavior of malware in a safe environment. However, some samples activate a malicious function only if they receive some particular inputs from its command and control server. The problem is then to get some grasp on the interactions between the malware and its environment. For that sake, we propose to work in four steps. First, we enumerate all possible execution path following the reception of a message. Second, we describe for all execution path the set of corresponding messages. Third, we build an automaton that discriminate types of runs given an arbitrary word. Finally, we unify some equivalent run, and simplify the underlying automaton.

1 Introduction

The essential problem of an anti-virus software is to discriminate malware from safe programs. There are broadly two ways to perform such a task: by a syntactic analysis of the code or by a behavioral one. The first one is the key of standard anti-virus software. Indeed, the principle takes benefit from optimized pattern matching algorithms and may be tuned to avoid false positives. On the other hand, they are very sensitive to malware mutations, and thus may be easily fooled.

Behavioral analyses try to discriminate good usages from bad ones. Usually, this is done by recording interactions of the program with its operating system: file modifications, access to some drive, memory rights alteration and so on. Good/Bad behaviors are modeled by state automata or by temporal logical formulae [19, 23]. Though attracting in theory, such systems suffer from an heavy slow down of the monitored machines. Furthermore, it is not that easy to distinguish good from bad behaviors leading to high false negative/positive rates. And finally, it is not that complicate to make behaviors mutate.

In this contribution, we propose an alternative form of behavioral analysis which focuses on the interactions between a program with its environment via messages. It results more technically in an analysis of the structure of input messages as read by programs. This is called *message format extraction*.

Message format extraction can be considered also in the context of retro-engineering as a first and inevitable step to understand the communication protocols used in the malicious code. In the pioneer work [5], the authors described a method to extract formats of received messages by a pattern analysis. Using similar techniques, this work was pursued later on for respectively sent messages [10], encrypted ones [14] or hierarchically-structured ones [8, 11], and finally for communication protocols [11].

In this line of work, messages are modeled as fields sequences. The main issue is then to detect the fields boundaries in a message. The problem is even more complicate due to fields (qualified as *directed*) which may contain informations about other fields (via length measure for instance). The authors developed algorithms that detect direction fields and *separators*. However, to do that, they make a strong assumption on the parsing algorithm of the target program itself. If one replaces the naive string matching algorithm with a more sophisticated one (e.g. Knuth-Morris-Pratt), the assumptions (described in these interpretations) are no more valid and thus the results.

Our approach relies on a different paradigm. We state that the structure of input messages can be described (or at least approximated) by finite state automata. Suppose we are given a program P which reads some input m at run time, then we provide a finite state automaton which will categorize input messages according to their corresponding execution. Actually, we will show experimentally that one retrieves the shape of messages from the automaton. Second difference, we do not make any hypothesis on the parsing algorithm of the program P.

However, from the idea to its concrete realization, there are some preliminary steps that must be crossed. Suppose we want to study the communications of a malware P. First, one must identify which instructions open a communication. This is not computable in general. But, for self-modifying program—that is almost all malware—standard heuristics will fail. This is a first reason to work dynamically. Second reason, suppose P opens a communication with its Command and Control (C&C). In that case, the C&C behaves like an oracle from our perspective and there is no way to get a simple sample of a message. Letting the communication open (and the program run) gives us access to a (hopefully informative) message from the other side.

But, in the scenario above, there are two issues. First, the C&C server may be closed. Malware managers move very quickly servers to escape traffic surveillance. Second, launching P may be observed by the C&C, thus breaking the stealth of the analysis. In the experiment we report here, we opened the communication *only once*. This can be seen as a kind of trade-off between stealth and information. However, the method we develop cope with the all scenarios, fully open or fully closed communications and all in-between ones.

So, let us suppose we run a program P which at some point receives some data. As said above, we will build an automaton which is fed by some byte sequence w (in other words, a message) will output a state witnessing the execution path in P of w. Actually, states of the automaton describe the analysis of the message by the program P, and consequently reveal the structure of messages.

To build the automaton, we proceed in four steps. First, we enumerate all execution paths from the reception instant that are reachable by message modification. This is known as a coverage issue in the context of software testing or in the context of malware retro-engineering. In recent times [6, 7, 22], the binary code is translated to some intermediate representation, and then some *concolic testing* [3] is performed with help of SMT solvers. Our approach is known as tainting driven fuzzing analysis which gave reasonably good coverage level for the experiments we made. But we already think for the near future to use techniques mentioned above, or some coming from disassembly such as the ones presented by Reps et al. [9] who developed CodeSurfer, or by Kinder and Veith [13], the authors of Jakstab, or the more recent Bardin et al. [18] which combines advantages of both latter methods.

For the second step, we proceed for each execution branch to a tainting analysis. For each conditional instructions (that are junctions in the execution tree), we describe a condition for each successor. Again, symbolic execution techniques could be used here, such as [17]. From these constraints, we build in a third step a dual automaton which serves as a basis for the fourth one. In the last step, we simplify the automaton by state sharing. Those are meant to fold execution loops.

We have implemented the constructions described all along, and we tested it on several programs, but notably on the well-known malware called `Zeus`. In the final Section, we present the result of the approach that we obtained on the malware.

Limitations In this contribution, we do not consider protocols. We work only on one message, not on the dialog between the program and its C&C. Thus, a malware could easily escape our analysis, simply by splitting messages. However, we think that a careful analysis of one message is a first step toward protocol analysis. Second limitation, we consider that the message analysis is done by some non-self-modifying code, globally, the code may be self-modifying, but not the parsing procedure itself. This limitation is not too harsh: during packer's decryption (that is when a malware is essentially self-modifying), there are few interactions with the environment. In his thesis, Calvet [21] showed that less than 2% of malware interact with the system during decryption. But, a code may interpret some commands sent by a C&C, and for those ones, our method is not operant. We let that issue for further researches.

Related work Beside the previously cited works [5, 8, 10, 11, 14], the automata approximation can be though of as a *learning procedure*. This approach has natural relations with researches in *machine learning* [1, 2] and may take benefit from results is this domain for the further development. Actually, we faced a problem very similar to the one of dalla Preda, Giacobazzi, Debray, Coogan and Townsend [15] but in a different setting. They describe execution paths of a program as automata, and at one point they compute approximations to get finite descriptions. Here, we focus on messages rather than execution paths, but actually this fact induces an underlying path approximation.

2 Background

In this contribution, we model the memory as a flat sequence of 8-bit bytes. The set of bytes is noted BYTES. The set of addresses (the address space) is noted DWORDS. Typically, the address space is the range $0..2^{32}-1$, that is 4 BYTES. The memory is represented as a finite mapping $\mu : \text{DWORDS} \to \text{BYTES}$. Aside from the memory, there is a finite set of *registers*, next denoted REGISTERS, which contains `eip`, the *instruction pointer*. The state of registers is represented by a finite function $\nu : \text{REGISTERS} \to \text{DWORDS}$.

An *execution environment* (μ, ν) represents the state of the system. At each step of the computation, an *opcode* `c` pointed by `eip` is fetched from memory, and depending on it, the execution environment is updated. The couple $(\nu(\texttt{eip}), \texttt{c})$ is the *underlying instruction* of (μ, ν) next denoted $\mathcal{I}(\mu, \nu)$.

Accordingly, the execution from an initial state (μ_1, ν_1) is the sequence $(\mu_1, \nu_1) \to (\mu_2, \nu_2) \to \cdots$. From this execution, we extract the *sequence of underlying instructions* called the *trace* $T(\mu, \nu) = \texttt{i}_1, \texttt{i}_2, \ldots$ where $\texttt{i}_j = \mathcal{I}(\mu_j, \nu_j)$ for all $j \geq 1$ (see Example 1).

A running program P may ask for some inputs. To do that, it gives the control back to the operating system which will itself (a) push in memory a sequence of bytes m and (b) resume P. We call the execution environment at instant (b) an *input environment*. Input environments are characterized to be the "return" state of reading interruptions, or to follow input functions calls (e.g. `WSARecv` or `recv` on the Windows® OS). The instruction corresponding to an input environment is said to be an *input instruction*.

Let $N = (\mu, \nu)$ be an input environment. The *input message* m is stored in memory in an interval of addresses $A = [\texttt{b}, \texttt{b}+1, \ldots, \texttt{b}+\ell-1]$ with ℓ the length of the message m and `b` some (more or less arbitrary) address. Given some message $m' = b_0 \cdots b_{\ell'-1} \in \text{BYTES}^*$, let $N[m'] = (\mu', \nu')$ be the same environment as N but the one which received m' instead of m. More technically, suppose ℓ is stored in `eax` then $\mu'(\texttt{b} + i) = b_i$ for all $i < \ell'$, otherwise $\mu'(\texttt{a}) = \mu(\texttt{a})$ and $\nu'(r) = \nu(r)$ for all registers except `eax` for which $\nu'(\texttt{eax}) = \ell'$. For the remaining, we define $m[i] = b_i$ for all $i < \ell'$.

Given some input environment N, consider an instruction $\texttt{i}_j$ in a trace $T(N) = \texttt{i}_1, \ldots, \texttt{i}_j, \texttt{i}_{j+1}, \ldots$, we say that $\texttt{i}_j$ is *deterministic* if for any message m such that $T(N[m]) = \texttt{i}_1, \ldots, \texttt{i}_j, \texttt{i}', \ldots$, then $\texttt{i}' = \texttt{i}_{j+1}$. In other words, the successor of a deterministic instruction does not depend on messages. Typically, sequential instructions (e.g. `mov ebp, esp`), calls and unconditional jumps to some fixed address (e.g. `call 0xf1b542` or `jmp 0xf1ae33`) enter this category.

Non deterministic instructions are called *message-triggered*. In this category, one finds calls or jumps to some register (e.g. `call edx` or `jmp eax`), but the most representative are the conditional jumps (e.g. `jnz 0xf1a9ce`).

2.1 Execution tree

Let us consider a program P asking for some input m, its behavior will then depend on m. In a first step, we compute all possible run depending on m seen as a parameter. In a second step, for each execution branch, we describe the language of messages corresponding to that branch. But, for the moment, let us define the execution tree of an input environment N.

Consider an input environment N, define i_0^N to be the index of the first message triggered instruction in the trace $T(N) = \mathtt{i}_1, \ldots, \mathtt{i}_{i_0^N}, \ldots$. Actually, the prefix $\mathtt{i}_1, \ldots, \mathtt{i}_{i_0^N}$ is shared by all traces $T(N[m])$ with m a message. Indeed, by definition, $\mathtt{i}_1, \ldots, \mathtt{i}_{i_0^N}$ are deterministic but the last one, thus the sequence is uniquely determined by N. In the sequel, we shall drop the superscript N when the context is clear.

The trace $T_N(m) = \mathtt{i}_1, \ldots, \mathtt{i}_n$ is the sub-sequence of $T(N[m])$ cut after a second *input* instruction $\mathtt{i}_n$ (if such an $\mathtt{i}_n$ exists, otherwise $T_N(m) = T(N[m])$). In other words, we forget anything which would follow the analysis of a second message coming from the environment. The *trigger-trace* of m is the sequence $B_N(m) = [(\mathsf{a}_1, \mathtt{i}_{i_1}), \ldots, (\mathsf{a}_k, \mathtt{i}_{i_k})]$ where:

– $\mathtt{i}_{i_0}, \mathtt{i}_{i_1}, \ldots, \mathtt{i}_{i_k}$ is the subsequence of $T_N(m)$ containing only the *message-triggered instructions* and (if relevant) the input instruction at the end of $T_N(m)$,
– $\mathsf{a}_{\ell+1}$ is the address pointed by `eip` at step $\mathtt{i}_{i_\ell+1}$ for all $\ell \geq 0$, that is the address of the instruction following $\mathtt{i}_{i_\ell}$.

$$T_N(m) = \mathcal{I}(N) \rightarrow \cdots \rightarrow \mathtt{i}_{i_0} \rightarrow \mathtt{i}_{\mathsf{a}_1} \rightarrow \cdots \rightarrow \mathtt{i}_{i_1} \rightarrow \mathtt{i}_{\mathsf{a}_2} \rightarrow \ldots$$

One may observe that the trace $T_N(m)$ is uniquely determined by $B_N(m)$. Indeed, with the notation above, for all $\ell \geq 0$, instructions $\mathtt{i}_{i_\ell+1}, \ldots, \mathtt{i}_{i_{\ell+1}-1}$ are deterministic, thus uniquely determined by $\mathsf{a}_{\ell+1}$.

Example 1. Let us run `wget` on some `URL`, the program receives a first message from the server via the operating system. It then verifies that the first four bytes of this message are "HTTP". Figure 1 displays the corresponding piece of trace:

address	trace	trigger-trace
0x404f89	cmpsb [esi],[edi]	(0x403a20,cmpsb [esi],[edi])
0x404f89	cmpsb [esi],[edi]	(0x404f89,cmpsb [esi],[edi])
0x404f89	cmpsb [esi],[edi]	(0x404f89,cmpsb [esi],[edi])
0x404f89	cmpsb [esi],[edi]	
0x404f8b	pop edi	
0x404f8c	pop esi	
0x404f8d	jz 0x404f94	(0x404f89,jz 0x404f94)
0x404f94	test edx, edx	...

Fig. 1: Trace and its corresponding triggered-trace

The notion of message-triggered instruction is relative to traces. Indeed, the instruction `cmpsb [esi],[edi]` at address `0x404f89` is message triggered the first three times, not the fourth one (that is because of `cmpsb` stops either when `[esi]` $\neq$ `[edi]` or if `ecx` $= 0$, and `ecx`'s value is 0 after the third time).

Definition 1 (Trigger Execution Tree (TET)). *Given an input environment N, its trigger execution tree $\mathcal{E}_N$ is a tree with labeled vertices and labeled edges, that is defined inductively as follows:*

- *the root is labeled $\mathtt{i}_{i_0^N}$, it is associated with the empty sequence $[]$.*
- *suppose $B_N(m) = [(\mathfrak{a}_1, \mathtt{i}_{i_1}), \ldots, (\mathfrak{a}_k, \mathtt{i}_{i_k})]$ is a trigger-trace, suppose that the vertex n is associated with $[(\mathfrak{a}_1, \mathtt{i}_{i_1}), \ldots, (\mathfrak{a}_j, \mathtt{i}_{i_j})]$ for some $j < k$, then $\mathcal{E}_N$ contains a transition $n \xrightarrow{\mathfrak{a}_{j+1}} n'$ with n' a (fresh) vertex labeled $\mathtt{i}_{i_{j+1}}$ and associated to $[(\mathfrak{a}_1, \mathtt{i}_{i_1}), \ldots, (\mathfrak{a}_{j+1}, \mathtt{i}_{i_{j+1}})]$.*

The trigger execution tree is the standard execution tree once removed intermediate deterministic instructions. Each path from the root to some leaf of $\mathcal{E}_N$ corresponds to a trigger trace, thus to a trace in the original program.

Since traces are infinite in general, so may be the execution tree. Given $c > 0$, let $T_c(N)$ be the prefix of length c of the trace $T(N)$, then we define $\mathcal{E}_{N,c}$ to be the sub-tree of $\mathcal{E}_N$ cut to traces of length at most c.

Example 2. Figure 3a shows the TET of `wget` with traces limited to 50 instructions. The vertices $\perp_i$'s mark the ends of traces due to length limit.

Remark 1. From now on, the trigger execution trees are understood as to be cut at some length limit c. We then omit c in $\mathcal{E}_{N,c}$ when it is clear from the context.

3 Input messages analysis

3.1 Dynamic tainting analysis

Dynamic tainting is a well known technique which tracks the dependency of data within programs. The below is a very brief introduction, we refer the reader to [16] for details.

The semantics of an instruction $\mathtt{i} = \mathcal{I}(N)$ of some execution environment N depends on a set of registers[1] R and memory content at a set of addresses A, and modify a set register[2] R' and memory content at a set of addresses A'. For such an $\mathtt{i}$, one derives a *hyper-edge* $(R \cup A, \mathtt{i}, R' \cup A')$ where source nodes are "read" memory addresses or registers, targets are "written" ones. Source and target nodes are considered separate. We say that $\mathtt{i}$ depends on source nodes.

Let us consider a prefix $\mathtt{i}_1, \ldots, \mathtt{i}_j$ of some trace $T(N) = \mathtt{i}_1, \mathtt{i}_2, \ldots$. The *hyper-graph* $\mathcal{H}(\mathtt{i}_1, \ldots \mathtt{i}_j)$ is the chain of edges obtained by *glueing* corresponding target and sources (cf. Example 3). The sources of $\mathcal{H}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$, denoted by $\mathcal{S}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$, consists of source nodes that are not targets of any hyper-edge (that is source nodes which are not glued). The set $\mathcal{S}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$ describes the dependency of $\mathtt{i}_1, \ldots, \mathtt{i}_j$ on N. The local sources $\mathcal{S}_\ell(\mathtt{i}_1, \ldots, \mathtt{i}_j)$ are source nodes in $\mathcal{S}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$ related by a path to the targets of $\mathtt{i}_j$. They describe the dependency of $\mathtt{i}_j$ on N (supposedly it is executed).

[1] but `eip` which is treated apart.

[2] Idem.

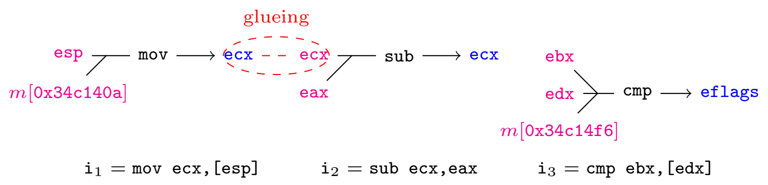

glueing
esp
mov
ecx
ecx
sub
ecx
m[0x34c140a]
eax
ebx
edx
cmp
eflags
m[0x34c14f6]
$\mathtt{i}_1 = \mathtt{mov\ ecx,[esp]}$
$\mathtt{i}_2 = \mathtt{sub\ ecx,eax}$
$\mathtt{i}_3 = \mathtt{cmp\ ebx,[edx]}$

Fig. 2: Hyper-edge glueing

Example 3. Given that `esp` and `edx` have respectively the value `0x34c140a` and `0x34c14f6` at the beginning. The hyper-graph $\mathcal{H}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_2)$ in Figure 2 has sources $\mathcal{S}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) = \{\mathtt{esp}, m[\mathtt{0x34c140a}], \mathtt{eax}, \mathtt{ebx}, \mathtt{edx}, m[\mathtt{0x34c14f6}]\}$ and local sources $\mathcal{S}_\ell(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) = \{\mathtt{ebx}, \mathtt{edx}, m[\mathtt{0x34c14f6}]\}$.

Proposition 1. *The tainting analysis leads to two main facts.*

1. *Any instruction in a finite trace depends only on finitely many input bytes from memory. It is a direct consequence from the fact that assembly instructions read and modify only locally the memory.*
2. *Given some execution environment N and some instruction $\mathtt{i}_j$ occurring in its trace $T(N) = \mathtt{i}_1, \ldots, \mathtt{i}_j, \ldots$, let N' be an environment coinciding with N on the sources of $\mathcal{H}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$, then, $T(N') = \mathtt{i}_1, \ldots, \mathtt{i}_j, \ldots$ coincide with the trace of N at least up to the j-th instruction.*

3.2 Trace formula construction

In this section, and the following ones, N denotes an input environment and $A = [\mathtt{b}, \ldots, \mathtt{b} + \ell - 1]$ is the relative interval of addresses. A *local condition* is a subset $\gamma \subseteq \textsc{Bytes}^S$ for some (finite) set S, called the support of γ.

Definition 2 (Trace language). *Given a program P and some input environment N, its execution tree (possibly cut at some length c) is $\mathcal{E}_N$. Let p be a finite path in $\mathcal{E}_N$ from the root to some leaf, its language is the set of messages*

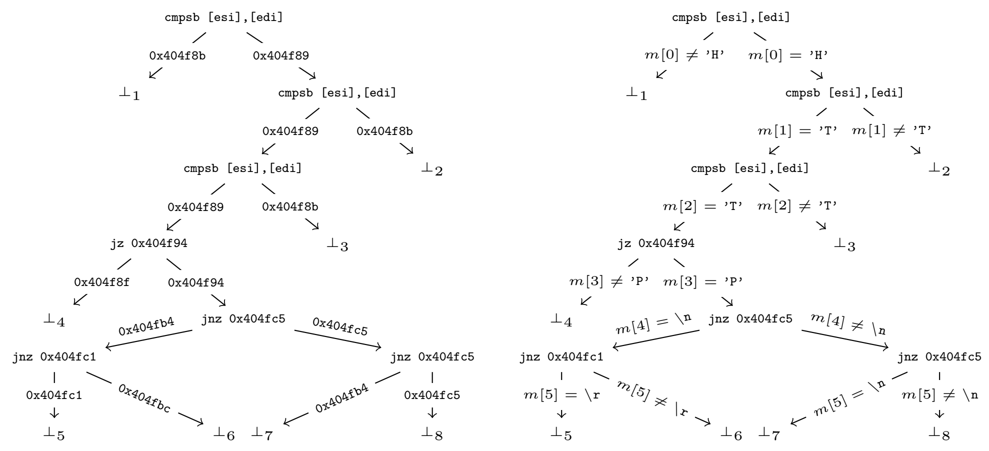
$$M_N(p) = \{m \in \mathcal{M} \mid B_N(m) = p\}.$$

A path (from the root to some leaf) p in $\mathcal{E}_N$ corresponds to a trigger-trace $B_N(m)$ of some $m \in M_N(p)$, and thus to a sub-trace $T_N(m) = \mathtt{i}_1, \ldots, \mathtt{i}_j$. Let $\mathcal{S}_p$ be the addresses of A that are sources of $\mathcal{H}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$, namely $\mathcal{S}_p = A \cap \mathcal{S}(\mathtt{i}_1, \ldots, \mathtt{i}_j)$. In other words, $\mathcal{S}_p$ is the set of addresses within the message which are read by the processor along p.

Proposition 2. *For any message $m \in M_N(p)$, for any message m', if $m' =_{|\mathcal{S}_p} m$ then $m' \in M_N(p)$.*

Proposition 2 can be restated as follows: the set $M_N(p)$ is isomorphic to $\gamma_p \times \textsc{Bytes}^{A \setminus \mathcal{S}_p}$ for some set $\gamma_p \subseteq \textsc{Bytes}^{\mathcal{S}_p}$. In terms of languages, it means that $M_N(p)$ is a regular one.

Example 4. In Figure 3a, writing $[\hat{}\mathtt{X}]$ for $(\textsc{Bytes} \setminus \{\mathtt{X}\})$, we have $M_N(\bot_1) = [\hat{}\mathtt{H}] \cdot \textsc{Bytes}^*$, $M_N(\bot_2) = \mathtt{H} \cdot [\hat{}\mathtt{T}] \cdot \textsc{Bytes}^*$, $M_N(\bot_3) = \mathtt{H} \cdot \mathtt{T} \cdot [\hat{}\mathtt{T}] \cdot \textsc{Bytes}^*$, etc.



(a) Trigger execution tree (b) Trigger message tree

Fig. 3: Trigger trees of `wget`

3.3 Message execution tree

A trigger execution tree can be seen as a decision tree, which gives a set of messages given a trace. Here, we consider the dual view that gives a trace given a message.

Definition 3. *A Message Execution Tree (MET) is a tree whose transitions are labeled by finite sets of bytes and leaves are labeled by a trigger path; for each vertex n and letter $b \in$ Bytes, there is a unique vertex n' such that $n \xrightarrow{S\ni b} n'$.*

Given a MET G and a message m, let us follow m within G until we reach a leaf (if any). $G(m)$ denote the trigger path labeling that leaf. The MET G is said to be compatible with a TET $\mathcal{E}_N$ if $m \in M_N(G(m))$ for all $m \in \text{Bytes}^*$.

We can always build a compatible MET for any finite[1] TET $\mathcal{E}_N$. Indeed, let $L = \max\left(\cup_{p\in\mathcal{E}_N}(\mathcal{S}_p)\right) - \mathtt{b}$ as the largest offset appearing in the sources of a path $p \in \mathcal{E}_N$, and let G be the full 256-ary tree of depth L. Each leaf of G corresponds to some word in $m \in \text{Bytes}^L$, we associate to that leaf the path p such that $m \in M_N(p)$, then G is compatible with $\mathcal{E}_N$.

However, one may notice that the full tree G described above is very large in general, its size is of the order 256^L, thus hardly computable. We developed some heuristics to get a much more compact representation.

3.4 Message decomposition

Given a path $p = [(\mathtt{a}_1, \mathtt{i}_{i_1}), \dots, (\mathtt{a}_k, \mathtt{i}_{i_k})]$ in a TET $\mathcal{E}_N$, it corresponds to a trace $T_N(m) = \mathtt{i}_1 \to \cdots \to \mathtt{i}_{i_0} \to \mathtt{i}_{\mathtt{a}_1} \to \cdots \to \mathtt{i}_{i_1} \to \mathtt{i}_{\mathtt{a}_2} \to \dots$. The tainting analysis shows that any $\mathtt{i} \in T_N(m)$ depends on the environment N with respect to a local sources $\mathcal{S}_l(\mathtt{i}_1, \mathtt{i}_2, \dots, \mathtt{i})$. Hence the set $\mathcal{S}^A(\mathtt{i}) = \mathcal{S}_\ell(\mathtt{i}_1, \mathtt{i}_2, \dots, \mathtt{i}) \cup A$ describes the *local dependency* of $\mathtt{i}$ with respects to the input.

[1]Possibly due to a length limit.

In the TET $\mathcal{E}_N$, we replace the label $\mathtt{a}_{j+1}$ of each edge $\mathtt{i}_{i_j} \xrightarrow{\mathtt{a}_{j+1}} \mathtt{i}_{i_{j+1}}$ by its *local condition*, that is the set $\gamma_{\mathtt{a}_{j+1}} = \{m_{|\mathcal{S}^A(\mathtt{i})} \mid m \in \textsc{Bytes}^* \wedge T_N(m) = \mathtt{i}_1, \ldots, \mathtt{i}_{\mathtt{a}_{j+1}}, \ldots\}$. The tree thus obtained from $\mathcal{E}_N$ is called the *trigger message tree* relative to $\mathcal{E}_N$, next denoted by $\mathcal{F}_N$.

Example 5. The fig. 3b shows the trigger message tree relative to the TET in Figure 3a.

Let $\mathcal{S}_{\mathcal{F}_N}$ be the union $\bigcup_{\mathtt{i} \in \mathcal{F}_N} \mathcal{S}^A(\mathtt{i})$ of local dependencies in $\mathcal{F}_N$. The relation $R_{\mathcal{F}_N} \subseteq \mathcal{S}_{\mathcal{F}_N} \times \mathcal{S}_{\mathcal{F}_N}$ is defined by $(a, a') \in R_{\mathcal{F}_N}$ if $a, a' \in \mathcal{S}^A(\mathtt{i})$ for some $\mathtt{i} \in \mathcal{F}_N$.

Definition 4 (Address closure). *The reflexive-transitive closure of $R_{\mathcal{F}_N}$ is called the address closure of $\mathcal{F}_N$, denoted by $\alpha_{\mathcal{F}_N}$.*

The address closure $\alpha_{\mathcal{F}_N}$ induces a partition $\mathcal{S}_{\mathcal{F}_N} = A_1 \cup \cdots \cup A_r \subseteq A$ of equivalence classes. Let p be some path in $\mathcal{F}_N$, from the root to some leaf, each $A_i \subseteq A$ gives a *local decomposition* of $M_N(p)$ defined by $\gamma_{p,A_i} = \{m_{|A_i} \mid m \in M_N(p)\}$. Then the path language $M_N(p)$ can be decomposed into:

Proposition 3. $M_N(p) = \gamma_{p,A_1} \times \gamma_{p,A_2} \times \cdots \times \gamma_{p,A_r} \times \textsc{Bytes}^{A \setminus \mathcal{S}_{\mathcal{F}_N}}$

local condition	path	
$m[0] =$'H'	`cmpsb [esi],[edi]`	i1
$m[1] =$'T'	`cmpsb [esi],[edi]`	i2
$m[2] =$'T'	`cmpsb [esi],[edi]`	i3
$m[3] =$'P'	`jz 0x404f94`	i4
$m[4] =$\n	`jnz 0x404fc5`	i5
$m[5] \neq$\n	`jnz 0x404fc1`	i6
$m[5] =$\r	`jnz 0x404fc5`	i7

Fig. 4: Path of `wget` and message decomposition

Example 6. The path p in Figure 4 is extracted from the execution tree of `wget` cut at length 60, it is of the form

$$\mathtt{i}_1 \xrightarrow{m[0]=\mathtt{'H'}} \mathtt{i}_2 \xrightarrow{m[1]=\mathtt{'T'}} \ldots \xrightarrow{m[4]=\backslash n} \mathtt{i}_6 \xrightarrow{m[5]\neq\backslash n} \mathtt{i}_7 \xrightarrow{m[5]=\backslash r} \perp$$

Both instructions $\mathtt{i}_6$ and $\mathtt{i}_7$ depend on the 5-th byte of the input, the local decomposition groups their local conditions into a single set $M_N(p)_{|\{5\}} = \{m[5] \neq \backslash n\} \cap \{m[5] = \backslash r\} = \{m[5] = \backslash r\}$, thus $M_N(p)$ decomposes:

$$M_N(p) = \{\mathtt{H}\} \times \{\mathtt{T}\} \times \{\mathtt{T}\} \times \{\mathtt{P}\} \times \{\backslash\mathtt{n}\} \times \{\backslash\mathtt{r}\} \times \textsc{Bytes}^{A \setminus \{0,1,2,3,4,5\}}$$

From the partition $\mathcal{S}_{\mathcal{F}_N} = A_1 \cup \cdots \cup A_r$ induced from the closure $\alpha_{\mathcal{F}_N}$ of $\mathcal{F}_N$, we derive a tree $\overline{\mathcal{F}_N}$ whose branches have the form where γ_i is a local condition on A_i. Moreover, each path in $\mathcal{E}_N$ corresponds uniquely to a leaf in $\overline{\mathcal{F}_N}$. Algorithm 1 constructs the tree $\overline{\mathcal{F}_N}$ when the following hypothesis holds.

Hypothesis 1 *For any path $\mathtt{i}_{i_0} \overset{\gamma_1}{\to} \mathtt{i}_{i_1} \overset{\gamma_2}{\to} \cdots$ in $\mathcal{F}_N$, let $A_{d(k)}$ denote the equivalence class containing the support of γ_k. Then, for all $k \leq l$, $d(k) \leq d(l)$.*

Proposition 4. *Let s_0 be the root of the trigger message tree $\overline{\mathcal{F}_N}$. For any path p in $\mathcal{E}_N$, there is a unique path $\overline{p} = s_0 \overset{\gamma_1}{\to} \cdots \overset{\gamma_r}{\to} s_r$ in $\overline{\mathcal{F}_N}$ such that $M_N(p) = \gamma_1 \times \cdots \times \gamma_r \times \text{Bytes}^{\complement A_1 \cup \cdots \cup A_r}$.*

```
Input: $\mathcal{P} = \{M_N(p) = \gamma_{p,A_1} \times \cdots \times \gamma_{p,A_r} \times \gamma_{p,\complement} \mid p$ is a path on $\mathcal{F}_N\}$.
Output: the tree $\overline{\mathcal{F}_N}$.
begin
    $\overline{\mathcal{F}_N} \leftarrow \{$initial state $s\}$;
    foreach $M_N(p) \in \mathcal{P}$ do
        $i \leftarrow 1, s_1 \leftarrow s$; while edge $e = s_i \xrightarrow{\gamma_{p,A_i}} s'_i$ for some $s'_i$ exists on $\overline{\mathcal{F}_N}$ do
            $i \leftarrow i + 1$; $s_1 \leftarrow$ source of $e$; $s_2 \leftarrow$ target of $e$;
        end
        while $i \neq j$ do
            $s_1 \leftarrow s_2$;
            if $i + 1 \leq n$ then
                $s_2 \leftarrow$ new state;
            else
                $s_2 \leftarrow$ new state labeled $\perp$;
            end
            add new edge $s_1 \xrightarrow{\gamma_{p,A_i}} s_2$ into $\overline{\mathcal{F}_N}$; $i \leftarrow i + 1$;
        end
    end
end
```

Algorithm 1: Tree construction

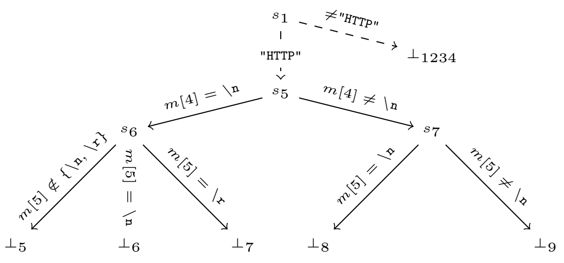


Fig. 5: Decomposition tree

Finally, local conditions in the tree $\overline{\mathcal{F}_N}$ can be reordered to follow the progression of a MET. We end with a message execution tree, next denoted $\mathcal{A}_N$.

Example 7. The fig. 5 shows the tree $\overline{\mathcal{F}_N}$ corresponding to the $\mathcal{E}_N$ discussed in Example 6. To save space, we grouped transitions of the first four characters, that is from s_1 to s_4. One can observe that $\overline{\mathcal{F}_N}$ in this case is indeed $\mathcal{A}_N$.

4 Message classification

In Section 3, we showed how to relate messages to traces of execution using path languages and their dual representation by message execution trees. With this analysis, some messages m and m' that are not treated in the same manner are not in the same path language, even if they are clearly related. For example, consider a program which looks for the first newline character by examining each byte, then the execution traces of the parsing of `"aaa\n"` and the one of `"aaaa\n"` fall into different categories. In this section, we propose a method to avoid such irrelevant details. The principle is to gather equivalent path languages with respect to the remaining execution of the program.

4.1 Message automaton

Definition 5 (Message automaton (MA)). *A message automaton $\mathscr{A}$ is a quadruple $\langle Q, \iota, E, P \rangle$ where*

- *Q is a set of states, $\iota \in Q$ being the one qualified to be initial,*
- *$E \subseteq Q \times \textsc{Bytes} \to Q$ is a transition function which is totally determined at some states, and totally undetermined at other states.*
- *$P : Q \to 2^{\mathscr{B}}$ maps each state to a set of trigger paths (in general, the trigger paths of $\mathscr{B}$ are distinguished from the paths within $\mathscr{A}$).*

In other words, it is a pair made of a (deterministic) automaton and the path function P. Given some word m, $E(\iota, m)$ and $Q(\iota, m)$ denote respectively the state and the path (within the automaton) following the word m along the transition function E from the initial state ι.

Proposition 5. *The message execution tree $\mathcal{A}_N$ has form of a message automaton $\langle Q, \iota, E, P \rangle$ where Q consists of all vertices, ι is the root vertex, E consists of $s \times \gamma \to s'$ whenever $s \xrightarrow{\gamma} s'$ is an edge of $\mathcal{A}_N$, and $P(q) = \{\text{the path from root to q}\}$ for any leaf q, otherwise $P(q) = \emptyset$.*

Given some trigger path equivalence $\approx \subseteq \mathscr{B} \times \mathscr{B}$, a *morphism* $h \colon \mathscr{A} \to \mathscr{A}'$ where $\mathscr{A} = \langle Q, \iota, E, P \rangle$ and $\mathscr{A}' = \langle Q', \iota', E', P' \rangle$ is a function $h \colon Q \to Q'$ satisfying: 1. $h(\iota) = \iota'$, 2. $E'(h(q), b) = h(q')$ for all $q \in Q$ and $b \in \textsc{Bytes}$, 3. $P'(h(q)) \approx P(q)$[1] for any $q \in Q$ satisfying $P(q) \neq \emptyset$.

Definition 6 (Observational approximation). *Given a MA $\mathscr{A} = \langle Q, \iota, E, P \rangle$, a MA $\mathscr{A}' = \langle Q', \iota', E', P' \rangle$ is said to approximate observationally $\mathscr{A}$ if $E(\iota, m)$ exists[2] then $E'(\iota', m)$ exists, moreover if $P(E(\iota, m)) \neq \emptyset$ then $P'(E'(\iota', m)) \approx P(E(\iota, m))$ and if $Q(\iota, m_1) \neq Q(\iota, m_2)$ then $Q'(\iota', m_1) \neq Q'(\iota', m_2)$, for all $m, m_1, m_2 \in \textsc{Bytes}^*$.*

Intuitively, the message automaton $\mathscr{A}'$ recognizes and distinguishes at least the paths of $\mathscr{A}$. Morphisms justify observational approximation:

[1] This is the set equality up to the equivalence $\approx$.

[2] Because E is a partial function, $E(\iota, m)$ does not always exist for any $m \in \textsc{Bytes}^*$.

Proposition 6. *If there is a morphism $h \colon \mathscr{A} \to \mathscr{A}'$, then $\mathscr{A}'$ approximates observationally $\mathscr{A}$.*

Example 8. The automaton $\mathscr{A}$ in Figure 6 is constructed from $\mathcal{A}_N$ in Example 7 using the morphism that maps $s_i \mapsto S_i$ and $\perp_i \mapsto \perp$ for all $1 \leq i \leq 7$. On $\mathscr{A}$, set the path function to be $\perp \mapsto \cup_{i \leq 7} \perp_i$ (with the identification of $\perp_i$ with its corresponding path in $\mathcal{A}_N$) and $S_i \mapsto \emptyset$ otherwise. One can observe that $\mathscr{A}$ approximates observationally $\mathcal{A}_N$.

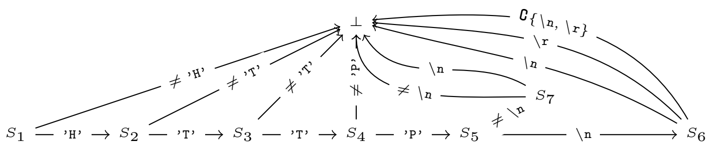


Fig. 6: Message automaton constructed from a morphism

Definition 7 (State equivalence). *Given a MA $\mathscr{A} = \langle Q, \iota, E, P \rangle$, a state equivalence relative to $\approx$ is an equivalence $\sim \subseteq Q \times Q$ satisfying: if $q_1 \sim q_2$ then*

- *if there is a transition $q_1 \xrightarrow{t} q_1'$*
  - *then either q_2 has no outgoing transitions,*
  - *or there is a state q_2' with transition $q_2 \xrightarrow{t} q_2'$ and $q_1' \sim q_2'$.*
- *if $P(q_1), P(q_2) \neq \emptyset$ then $P(q_1) \approx P(q_2)$.*

The co-recursive definition of the state equivalence is constructive. Indeed, to look for a state equivalence we can start the verification from any pair (q_1, q_2): first, add it as the first element of the relation whenever q_1 or q_2 has no transition, or both of them have similar transitions, then continue verifying pairs of corresponding states (q_1', q_2') or some pair obtained from the transitive closure of the current constructed relation. At any verification step, if states of the verified pair invalidate the definition, then the relation cannot be a state equivalence. In this case, we will restart verifying with another pair until an equivalence is completely constructed. This construction is given in the technical report.

Example 9. The fig. 7a shows the MET of `wget` with traces cut to length 70. Given the path equivalence $\perp_5 \approx \perp_9$, $\perp_6 \approx \perp_{10}$, $\perp_7 \approx \perp_{11}$, we have a state equivalence $s_5 \sim_1 s_7 \sim_1 \perp_8$, $s_6 \sim_1 s_{12}$, $\perp_5 \sim_1 \perp_9$, $\perp_6 \sim_1 \perp_{10}$ and $\perp_7 \sim_1 \perp_{11}$.

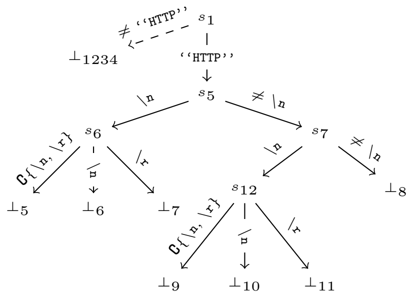
Given a MA $\mathscr{A} = \langle Q, \iota, E, P \rangle$, let $\sim$ be a state equivalence relative to some path equivalence $\approx$, then the MA $\mathscr{A}/\sim = \langle Q', \iota', E', P' \rangle$ constructed as follows

- $Q' = \{[q]/\sim \mid q \in Q\}$, $\iota' = [\iota]/\sim$,
- $E' = \{[q_1]/\sim \xrightarrow{t} [q_2]/\sim \mid q_1 \xrightarrow{t} q_2 \in E\}$
- $P' : [q]/\sim \mapsto \bigcup_{q' \in [q]/\sim} P(q')$.

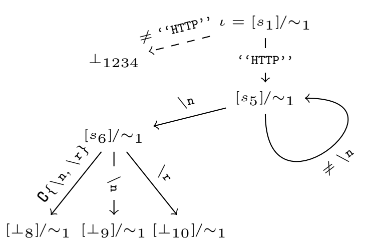
is called the *quotient of $\mathscr{A}$ by $\sim$*. This construction leads to:

Proposition 7. *The function $[\,]/\sim \colon Q \to Q'$ defines a morphism $\mathscr{A} \to \mathscr{A}/\sim$, then $\mathscr{A}/\sim$ approximates observationally $\mathscr{A}$.*

Example 10. The MA in Figure 7b is constructed from the state equivalence $\sim_1$ in Example 9, it approximates observationally the MET in Figure 7a.



(a) Message execution tree



(b) Quotient message automaton

4.2 Minimal message automaton

Let $\mathscr{A}$ be a message automaton, let $\sim$ be a nontrivial[1] equivalence on $\mathscr{A}$ then the quotient $\mathscr{A}/\sim$ has fewer state than $\mathscr{A}$. A message automaton $\mathscr{A}$ is said to be irreducible if there is no nontrivial equivalence on $\mathscr{A}$.

Proposition 8. *The composition $f \circ g$ of two morphisms is a morphism.*

Consider a sequence of morphisms $\mathcal{A}_N \xrightarrow{\texttt{id}} \mathscr{A}_1 \xrightarrow{h_1} \mathscr{A}_2 \xrightarrow{h_2} \dots$ where the h_i's are induced from some nontrivial state equivalence on $\mathscr{A}_i$, then this sequence is always finite (the number of states is strictly decreasing). It ends on an irreducible machine automaton $\mathscr{A}_{n+1}$. By Proposition 7 and Proposition 8, $h = h_n \circ h_{n-1} \circ \dots \circ h_1$ is a morphism. By Proposition 6, $\mathscr{A}_{n+1}$ approximates observationally $\mathcal{A}_N$. Moreover, since $\mathscr{A}_{n+1}$ is not reducible, it is minimal:

Proposition 9. *There is no message automaton that approximates observationally $\mathcal{A}_N$ but has less states than $\mathscr{A}_{n+1}$.*

Example 11. The MA in Figure 8 is reduced from the MA in Figure 7b using the morphism constructed from the state equivalence $\perp_1 \sim_2 \perp_2 \sim_2 \perp_3 \sim_2 \perp_4 \sim_2 {}^{[s_8]}/_{\sim_1} \sim_2 {}^{[s_9]}/_{\sim_1} \sim_2 {}^{[s_{10}]}/_{\sim_1}$, it approximates observationally the MET in Figure 7a. We let the reader verify that it is minimal.

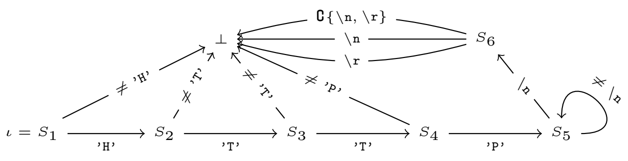


Fig. 8: Minimal message automaton

5 Implementation and experiments

To verify the relevance of the approach, we have implemented a *binary code coverage module* that builds the execution tree by a fuzzing techniques optimized

[1]Namely $\sim$ is not the equality.

by means of a tainting analysis; then the results are fed to an *automaton construction module* that constructs a minimal message automaton that represents the language of input messages for each examined program. We repeatedly apply non trivial state equivalence transformations up to a non reducible automaton.

The implementation of both consists in about 16500 lines of C++ code and is freely available online at [24]. The binary code coverage module is implemented with the help of the `Pin` DBI framework [4].

5.1 Experiments

The results given here obtained by our modules when examining `Zeus`, a well-known family of malwares disclosed in 2009. Some previous analyses [12, 20] have shown that this malware steals private informations on infected hosts. Below, we provide some details on `Zeus` that, as far as we know, are not published.

With an initial dynamic analysis, we could observe that `Zeus` hooks the API `send` and `WSASend`. Then it parses the data at the hooked socket to collect the informations. The Figure 9 shows the message automaton obtained from

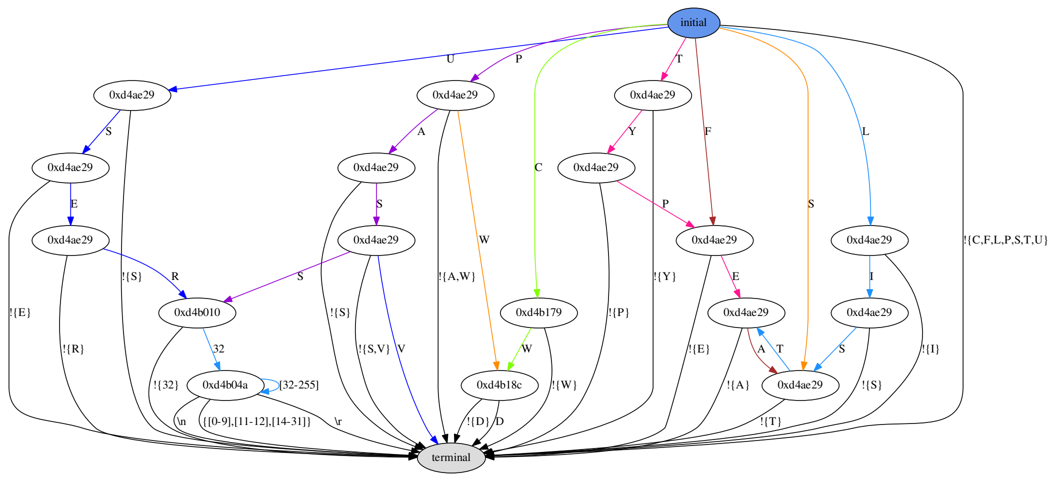


Fig. 9: Message automaton of `Zeus`

a message execution tree where traces are limited to the length 450 (for the path equivalence $\approx$, again, we identify all pending vertices $\perp_i$). Actually, the automaton classifies data that `Zeus` is interested in. First, one reads keywords directly within the automaton. Let us consider now the languages corresponding to path between the state mentioned as "initial" and the one as "terminal":

$$M_1 = \texttt{'U'}.\texttt{'S'}.\texttt{'E'}.\texttt{'R'}.\{32\}.\{32, 33, \ldots, 255\}^*$$
$$M_2 = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'S'}.\{32\}.\{32, 33, \ldots, 255\}^*$$
$$M_3 = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'V'}.\textsc{Bytes}^*$$

$$M_4 = \texttt{'P'}.\texttt{'W'}.\texttt{'D'}.\textsc{Bytes}^*$$
$$M_5 = \texttt{'C'}.\texttt{'W'}.\texttt{'D'}.\textsc{Bytes}^*$$
$$M_6 = \texttt{'T'}.\texttt{'Y'}.\texttt{'P'}.\texttt{'E'}.\{!\texttt{'A'}\}.\textsc{Bytes}^*$$
$$M_7 = \texttt{'F'}.\texttt{'E'}.\texttt{'A'}.\texttt{'T'}.\{!\texttt{'A'}\}.\textsc{Bytes}^*$$
$$M_8 = \texttt{'S'}.\texttt{'T'}.\texttt{'A'}.\texttt{'T'}.\{!\texttt{'A'}\}.\textsc{Bytes}^*$$
$$M_9 = \texttt{'L'}.\texttt{'I'}.\texttt{'S'}.\texttt{'T'}.\{!\texttt{'A'}\}.\textsc{Bytes}^*$$

The observed words `USER`, `PASS`, `PASV`, `PWD`, `CWD`, `TYPE`, `FEAT`, `STAT`, `LIST` are commands defined in the `ftp` protocol. That may be interpreted as an insight that `Zeus` steals information concerned with `ftp` communications. Moreover, besides these words there is no more commands of `ftp` occurring in the automaton, so we may think that `Zeus` is interested only in this subset of `ftp` commands. In the following languages that occur also within the automaton:

$$M_{10} = \texttt{'U'}.\texttt{'S'}.\texttt{'E'}.\texttt{'R'}.\{!32\}.\textsc{Bytes}^*$$
$$M_{11} = \texttt{'U'}.\texttt{'S'}.\texttt{'E'}.\texttt{'R'}.\{32\}.\{32, 33, \ldots, 255\}^k.\backslash\texttt{r}.\textsc{Bytes}^*$$
$$M_{11} = \texttt{'U'}.\texttt{'S'}.\texttt{'E'}.\texttt{'R'}.\{32\}.\{32, 33, \ldots, 255\}^k.\backslash\texttt{n}.\textsc{Bytes}^*$$
$$M_{12} = \texttt{'U'}.\texttt{'S'}.\texttt{'E'}.\texttt{'R'}.\{32\}.\{32, 33, \ldots, 255\}^k.\left(\{1, 2 \ldots, 31\} \setminus \{\backslash\texttt{r}, \backslash\texttt{n}\}\right).\textsc{Bytes}^*$$
$$M_{13} = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'S'}.\{!32\}.\textsc{Bytes}^*$$
$$M_{14} = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'S'}.\{32\}.\{32, 33, \ldots, 255\}^k.\backslash\texttt{r}.\textsc{Bytes}^*$$
$$M_{15} = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'S'}.\{32\}.\{32, 33, \ldots, 255\}^k.\backslash\texttt{n}.\textsc{Bytes}^*$$
$$M_{16} = \texttt{'P'}.\texttt{'A'}.\texttt{'S'}.\texttt{'S'}.\{32\}.\{32, 33, \ldots, 255\}^k.\left(\{1, 2, \ldots, 31\} \setminus \{\backslash\texttt{r}, \backslash\texttt{n}\}\right).\textsc{Bytes}^*$$

we notice that 32 is the `ASCII` code of the space character, and the set $\{1, 2, \ldots, 31\}$ contain all control characters. So these languages show clearly how `Zeus` parses a `USER` or `PASS` command: it checks whether the next character is space or not, if yes then it collects the bytes until a control character is detected. In other words, we retrieved the delimiter-oriented analysis of Caballero et al. [5].

6 Conclusion

The experiments we made so far showed that the hypothesis we made are generally speaking fulfilled. The parsing methods look more or less transparent. However, the shape of the automaton depends largely on the choice on the length of traces. At the same time, the computation time is more or less exponential in the size of the execution tree, thus, usually exponential in the length of traces. There a good reasons to optimize the computation of the execution tree and all derived ones, we think of symbolic execution, SMT-solvers and advanced tainting techniques. Finally, there is a parameter that we did not really used, namely the path equivalence $\approx$. Again, some first tries showed that this must be deepened.

References

[1] D. Angluin. “Inductive inference of formal languages from positive data ”. In: *Information and Control* 45.2 (1980), pp. 117–135.

[2] L. G. Valiant. “A Theory of the Learnable”. In: *CACM* 27 (1984).

[3] P. Godefroid et al. “DART: Directed Automated Random Testing”. In: PLDI. 2005.

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

[5] J. Caballero et al. “Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis”. In: CCS. 2007.

[6] A. Moser et al. “Exploring Multiple Execution Paths for Malware Analysis”. In: SSP. 2007.

[7] D. Brumley et al. “Automatically Identifying Trigger-based Behavior in Malware”. In: *Botnet Analysis and Defense*. 2008, pp. 65–88.

[8] Z. Lin et al. “Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution”. In: NDSS. 2008.

[9] T. W. Reps and G. Balakrishnan. “Improved Memory-Access Analysis for x86 Executables”. In: ETAPS. 2008.

[10] J. Caballero et al. “Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering”. In: CCS. 2009.

[11] P. M. Comparetti et al. “Prospex: Protocol Specification Extraction”. In: SSP. 2009.

[12] N. Falliere and E. Chien. *Zeus: King of the Bots*. Tech. rep. 2009.

[13] J. Kinder et al. “An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries”. In: VMCAI. 2009.

[14] Z. Wang et al. “ReFormat: Automatic reverse engineering of encrypted messages”. In: ESORICS. 2009.

[15] M dalla Preda et al. “Modeling metamorphism by Abstract Interpretation”. In: The 17th International Static Analysis Symposium, SAS’10. 2010.

[16] E. J. Schwartz et al. “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution”. In: *SSP*. 2010.

[17] S. Bardin and P. Herrmann. “OSMOSE: Automatic Structural Testing of Executables”. In: (2011).

[18] S. Bardin et al. “Refinement-based CFG Reconstruction from Unstructured Programs”. In: VMCAI. 2011.

[19] P. Beaucamps et al. “Abstraction-Based Malware Analysis Using Rewriting and Model Checking”. In: ESORICS. 2012.

[20] IOActive. *Reversal and Analysis of Zeus and SpyEye Banking Trojans*. Tech. rep. 2012.

[21] J. Calvet. “Analyse dynamique de logiciels malveillants”. Ecole polytechnique de Montréal et Université de Lorraine., 2013.

[22] S. Lecomte. “Élaboration d’une représentation intermédiaire pour l’exécution concolique et le marquage de données sous Windows”. In: SSTIC (2014).

[23] F. Song and T. Touili. “Pushdown model checking for malware detection”. In: *STTT* 16.2 (2014), pp. 147–173.

[24] G. Bonfante et al. *PathExplorer*. URL: https://github.com/tathanhdinh/PathExplorer.