Malware Analysis with Tree Automata Inference*,**

Domagoj Babić, Daniel Reynaud, and Dawn Song

University of California, Berkeley
{babic,reynaud,dawnsong}@cs.berkeley.edu

Abstract. The underground malware-based economy is flourishing and it is evident that the classical ad-hoc signature detection methods are becoming insufficient. Malware authors seem to share some source code and malware samples often feature similar behaviors, but such commonalities are difficult to detect with signature-based methods because of an increasing use of numerous freely-available randomized obfuscation tools. To address this problem, the security community is actively researching behavioral detection methods that commonly attempt to understand and differentiate how malware behaves, as opposed to just detecting syntactic patterns. We continue that line of research in this paper and explore how formal methods and tools of the verification trade could be used for malware detection and analysis. We propose a new approach to learning and generalizing from observed malware behaviors based on tree automata inference. In particular, we develop an algorithm for inferring k-testable tree automata from system call dataflow dependency graphs and discuss the use of inferred automata in malware recognition and classification.

1 Introduction

Over the last several decades, the IT industry advanced almost every aspect of our lives (including health care, banking, traveling,...) and industrial manufacturing. The tools and techniques developed in the computer-aided verification community played an important role in that advance, changing the way we design systems and improving the reliability of industrial hardware, software, and protocols.

In parallel, another community made a lot of progress exploiting software flaws for various nefarious purposes, especially for illegal financial gain. Their inventions are often ingenious botnets, worms, and viruses, commonly known as *malware*. Malware source code is rarely available and malware is regularly designed so as to thwart static analysis through the use of obfuscation, packing, and encryption [34].

* This material is based upon work partially supported by the National Science Foundation under Grants No. 0832943, 0842694, 0842695, 0831501, 0424422, by the Air Force Research Laboratory under Grant No. P010071555, by the Office of Naval Research under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1-0352 and FA9550-09-1-0539. The work of the first author is also supported by the Natural Sciences and Engineering Research Council of Canada PDF fellowship.

** The full version of this paper, the inference engine source code, all dependency graphs used in this paper, and a brief explanation of the file format, are available at http://www.domagoj.info/

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 116–131, 2011.

For the above mentioned reasons, detection, analysis, and classification of malware are difficult to formalize, explaining why the verification community has mostly avoided, with some notable exceptions (e.g., [8,18]), the problem. However, the area is in a dire need of new approaches based on strong formal underpinnings, as less principled techniques, like signature-based detection, are becoming insufficient. Recently, we have been experiencing a flood of malware [31], while the recent example of Stuxnet (e.g., [27]) shows that industrial systems are as vulnerable as our every-day computers.

In this paper, we show how formal methods, more precisely tree automata inference, can be used for capturing the essence of malicious behaviors, and how such automata can be used to detect behaviors similar to those observed during the training phase. First, we execute malware in a controlled environment to extract dataflow dependencies among executed system calls (*syscalls*) using dynamic taint analysis [5,29]. The main way for programs to interact with their environment is through syscalls, which are broadly used in the security community as a high-level abstraction of software behavior [13,23,32]. The dataflow dependencies among syscalls can be represented by an acyclic graph, in which nodes represent executed syscalls, and there is an edge between two nodes, say s_1 and s_2, when the result computed by s_1 (or a value derived from it) is used as a parameter of s_2. Second, we use tree automata inference to learn an automaton recognizing a set of graphs. The entire process is completely automated.

The inferred automaton captures the essence of different malicious behaviors. We show that we can adjust the level of generalization with a single tunable factor and how the inferred automaton can be used to detect likely malicious behaviors, as well as for malware classification. We summarize the contributions of our paper as follows:

- Expansion of dependency graphs into trees causes exponential blowup in the size of the graph, similarly as eager inlining of functions during static analysis. We found that a class of tree languages, namely k-testable tree languages [35] can be inferred directly from dependency graphs, avoiding the expansion to trees.
- We improve upon the prior work on inference of k-testable tree languages by providing an $\mathcal{O}(kN)$ algorithm, where k is the size of the pattern and N is the size of the graph used for inference.
- We show how inferred automata can be used for detecting likely malicious behaviors and for malware classification. To our knowledge, this is the first work applying the theory of tree automata inference to malware analysis. We provide experimental evidence that our approach is both feasible and useful in practice.
- While previous work (e.g., [7,13]) often approximated dependencies by syntactic matching of syscall parameters, we implemented a tool for tracking dependencies via taint analysis [5,29] and we made the generated dependency graphs, as well as the tree automata inference engine, publicly available to encourage further research.

2 Related Work

Tree Automata Inference. Gold [17] showed that no super-finite (contains all finite languages and at least one infinite) is identifiable in the limit from positive examples[1]

[1] Positive examples are examples belonging to the language to be inferred, while negative examples are those not in the language.

only. For instance, regular and regular tree languages [9] are super-finite languages. We have two options to circumvent this negative result; either use both positive and negative examples, or focus on less expressive languages that are identifiable in the limit from positive examples only. Inference of minimal finite state automata from both positive and negative examples is known to be NP-complete [17]. The security community is discovering millions of new malware samples each year and inferring a single minimal classifier for all the samples might be infeasible. Inferring a non-minimal classifier is feasible, but the classifier could be too large to be useful in practice. Thus, we focus on a set of languages identifiable in the limit from positive examples in this paper.

A subclass of regular tree languages — k-testable tree languages [35] — is identifiable in the limit from positive examples only. These languages are defined in terms of a finite set of k-level-deep tree patterns. The k factor effectively determines the level of abstraction, which can be used as a knob to regulate the ratio of false positives (goodware detected as malware) and false negatives (undetected malware). The patterns partition dependency graphs into a finite number of equivalence classes, inducing a state-minimal automaton. The automata inferred from positive (malware) examples could be further refined using negative (goodware) examples. Such a refinement is conceptually simple, and does not increase the inference complexity, because of the properties of k-testable tree languages. We leave such a refinement for future work.

A number of papers focused on k-testable tree automata inference. Garcia and Vidal [15] proposed an $\mathcal{O}(kPN)$ inference algorithm, where k is the size of the pattern, P the total number of possible patterns, and N the size of the input used for inference. Many patterns might not be present among the training samples, so rather than enumerating all patterns, [14] and [22] propose very similar algorithms that use only the patterns present in the training set. Their algorithms are somewhat complex to implement as they require computation of three different sets (called roots, forks, and leaves). Their algorithms are $\mathcal{O}\left(M^k N \log(N)\right)$, where M is the maximal arity of any alphabet symbol in the tree language. We derive a simpler algorithm, so that computing forks and leaves becomes unnecessary. The complexity of our algorithm is $\mathcal{O}(kN)$, thanks to an indexing trick that after performing k iterations over the training sample builds an index for finding patterns in the training set. Patterns in the test set can be located in the index table in amortized time linear in the size of the pattern. In our application — malware analysis — the k factor tends to be small (≤ 5), so our algorithm can be considered linear-time.

Malware Analysis. From the security perspective, several types of malware analysis are interesting: malware detection (i.e., distinguishing malware from goodware), classification (i.e., determining the family of malware to which a particular sample belongs), and phylogeny (i.e., forensic analysis of evolution of malware and common/distinctive features among samples). All three types of analyses are needed in practice: detection for preventing further infections and damage to the infected computers, and the other two analyses are crucial in development of new forms of protection, forensics, and attribution. In this paper, we focus on detection and classification.

The origins of the idea to use syscalls to analyze software can be traced to Forrest et al. [12], who used fixed-length sequences of syscalls for intrusion detection. Christodorescu et al. [7] note that malware authors could easily reorder data-flow-independent syscalls, circumventing sequence-detection schemes, but if we analyze

data-flow dependencies among syscalls and use such dependency graphs for detection, circumvention becomes harder. Data-flow-dependent syscalls cannot be (easily) reordered without changing the semantics of the program. They compute a difference between malware and goodware dependency graphs, and show how resulting graphs can be used to detect malicious behaviors. Such graph matching can detect only the exact behavioral patterns already seen in some sample, but does not automatically generalize from training samples, i.e., does not attempt to overapproximate the training set in order to detect similar, but not exactly the same behaviors.

Fredrikson et al. [13] propose an approach that focuses on distinguishing features, rather than similarities among dependency graphs. First, they compute dependency graphs at runtime, declaring two syscalls, say s_1 and s_2, dependent, if the type and value of the value returned by s_1 are equal to the type and value of some parameter of s_2 and s_2 was executed after s_1. They extract significant behaviors from such graphs using structural leap mining, and then choose behaviors that can be combined together using concept analysis. In spite of a very coarse unsound approximation of the dependency graph and lack of automatic generalization, they report 86% detection rate on around 500 malware samples used in their experiments. We see their approach as complementary to ours: the tree-automata we infer from real dependency graphs obtained through taint analysis could be combined with leap mining and concept analysis, to improve their classification power.

Bonfante et al. [3] propose to unroll control-flow graphs obtained through dynamic analysis of binaries into trees. The obtained trees are more fine-grained than the syscall dependency graphs. The finer level of granularity could, in practice, be less susceptible to mimicry attacks (e.g., [33]), but is also easier to defeat through control-flow graph manipulations. The computed trees are then declared to be tree automata and the recognizer is built by a union of such trees. Unlike inference, the union does not generalize from the training samples. The reported experiments include a large set of malware samples (over 10,000), but the entire set was used for training, and authors report only false positives on a set of goodware (2653 samples). Thus, it is difficult to estimate how well their approach would work for malware detection and classification.

Taint Analysis. Dynamic taint analysis (DTA) [29] is a technique used to follow data flows in programs or whole systems at runtime. DTA can be seen as a single-path symbolic execution [21] over a very simple domain (set of taints). Its premises are simple: *taint* is a variable annotation introduced through *taint sources*, it is propagated through program execution according to some *propagation rules* until it reaches a *taint sink*. In our case, for instance, taint sources are the syscalls' output parameters, and taint sinks are the input parameters.

As will be discussed in detail later, our implementation is based on the binary rewriting framework Pin [25] and uses the taint propagation rules from Newsome and Song [29]. Since DTA must operate at the instruction-level granularity, it poses a significant runtime overhead. Our DTA implementation executes applications several thousand times slower than the native execution. Our position is that the speed of the taint analysis is less important than the speed of inference and recognition. The taint analysis can be run independently for each sample in parallel, the dependency graph extraction is linear with the length of each execution trace, and hardware-based information flow

tracking has been proposed (e.g., [30,11]) as a potential solution for improving performance. In contrast, inference techniques have to process all the samples in order to construct a single (or a small number of) recognizer(s). An average anti-virus vendor receives millions of new samples annually and the number of captured samples has been steadily growing over the recent years. Thus, we believe that scalability of inference is a more critical issue than the performance of the taint analysis.

3 Notation and Terminology

In this section, we introduce the notation and terminology used throughout the paper. First, we build up the basic formal machinery that allows us to define tree automata. Second, we introduce some notions that will help us define *k-roots* that can be intuitively seen as the top k levels of a tree. Later, we will show how k-roots induce an equivalence relation used in our inference algorithm. Towards the end of this section, we introduce *k-testable* languages, less expressive than regular tree languages, but suitable for designing fast inference algorithms.

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^*$ the free monoid generated by $\mathbb{N}$ with concatenation $(\cdot)$ as the operation and the empty string ε as the identity. The prefix order $\leq$ is defined as: $u \leq v$ for $u, v \in \mathbb{N}^*$ iff there exists $w \in \mathbb{N}^*$ such that $v = u \cdot w$. For $u \in \mathbb{N}^*, n \in \mathbb{N}$, the *length* $|u|$ is defined inductively: $|\varepsilon| = 0, |u \cdot n| = |u| + 1$. We say that a set S is *prefix-closed* if $u \leq v \wedge v \in S \Rightarrow u \in S$. A *tree domain* is a finite non-empty prefix-closed set $D \subset \mathbb{N}^*$ satisfying the following property: if $u \cdot n \in D$ then $\forall 1 \leq j \leq n \,.\, u \cdot j \in D$.

A *ranked alphabet* is a finite set $\mathscr{F}$ associated with a finite *ranking relation arity* $\subseteq \mathscr{F} \times \mathbb{N}$. Define $\mathscr{F}_n$ as a set $\{f \in \mathscr{F} \mid (f,n) \in arity\}$. The set $T(\mathscr{F})$ of *terms* over the ranked alphabet $\mathscr{F}$ is the smallest set defined by:

1. $\mathscr{F}_0 \subseteq T(\mathscr{F})$
2. if $n \geq 1$, $f \in \mathscr{F}_n$, $t_1, \ldots, t_n \in T(\mathscr{F})$ then $f(t_1, \ldots, t_n) \in T(\mathscr{F})$

Each term can be represented as a finite ordered *tree* $t : D \to \mathscr{F}$, which is a mapping from a tree domain into the ranked alphabet such that $\forall u \in D$:

1. if $t(u) \in \mathscr{F}_n$, $n \geq 1$ then $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$
2. if $t(u) \in \mathscr{F}_0$ then $\{j \mid p \cdot j \in D\} = \emptyset$

As usual in the tree automata literature (e.g., [9]), we use the letter t (possibly with various indices) both to represent a tree as a mathematical object and to name a relation that maps an element of a tree domain to the corresponding alphabet symbol. An example of a tree with its tree domain is given in Figure 1.

The set of all positions in a particular tree t, i.e., its domain, will be denoted $dom(t)$. A *subtree* of t rooted at position u, denoted t/u is defined as $(t/u)(v) = t(u \cdot v)$ and $dom(t/u)$

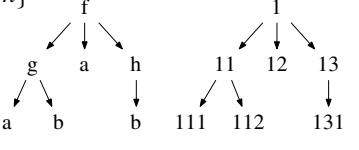

Fig. 1. An Example of a Tree t and its Tree Domain. $dom(t) = \{1, 11, 111, 112, 12, 13, 131\}$, $\mathscr{F} = \{f, g, h, a, b\}$, $\| t \| = 3$, $t(1) = f$, $t/131 = b$.

$= \{v \mid u \cdot v \in dom(t)\}$. We generalize the *dom* operator to sets as usual: $dom(S) = \{dom(u) \mid u \in S\}$. The *height* of a tree t, denoted $\| t \|$, is defined as:

$$\| t \| = \max(\{|u| \text{ such that } u \in dom(t)\})$$

Let $\Xi = \{\xi_f \mid f \in \bigcup_{i>0} \mathscr{F}_i\}$ be a set of new nullary symbols such that $\Xi \cap \mathscr{F} = \emptyset$. The Ξ set will be used as a set of *placeholders*, such that ξ_f can be substituted only with a tree t whose position one (i.e., the *head*) is labelled with f, i.e., $t(1) = f$. Let $T(\Xi \cup \mathscr{F})$ denote the set of trees over the ranked alphabet and placeholders. For $t, t' \in T(\Xi \cup \mathscr{F})$, we define the *link* operation $t \sharp t'$ by:

$$(t \sharp t')(n) = \begin{cases} t(n) & \text{if } t(n) \notin \Xi \vee (t(n) = \xi_f \wedge f \neq t'(1)) \\ t'(z) & \text{if } n = y \cdot z,\ t(y) = \xi_{t'(1)},\ y \in dom(t),\ z \in dom(t') \end{cases}$$

For any two trees, $t, t' \in T(\mathscr{F})$, the *tree quotient* $t^{-1}t'$ is defined by:

$$t^{-1}t' = \{t'' \in T(\Xi \cup \mathscr{F}) \mid t' = t'' \sharp t\}$$

The tree quotient operation can be extended to sets, as usual: $t^{-1}S = \{t^{-1}t' \mid t' \in S\}$. For any $k \geq 0$, define k-root of a tree t as:

$$root_k(t) = \begin{cases} t & \text{if } t(1) \in \mathscr{F}_0 \\ \xi_f & \text{if } f = t(1), f \in \bigcup_{i>0} \mathscr{F}_i,\ k = 0 \\ f(root_{k-1}(t_1), \ldots, root_{k-1}(t_n)) & \text{if } t = f(t_1, \ldots, t_n),\ \| t \| > k > 0 \end{cases}$$

A *finite deterministic bottom-up tree automaton* (FDTA) is defined as a tuple $(Q, \mathscr{F}, \delta, F)$, where Q is a finite set of states, $\mathscr{F}$ is a ranked alphabet, $F \subseteq Q$ is the set of final states, and $\delta = \bigcup_i \delta_i$ is a set of *transition relations* defined as follows: $\delta_0 : \mathscr{F}_0 \to Q$ and for $n > 0$, $\delta_n : (\mathscr{F}_n \times Q^n) \to Q$.

The *k-testable in the strict sense* (k-TSS) languages [22] are intuitively defined by a set of tree patterns allowed to appear as the elements of the language. The following theorem is due to López et al. [24]:

Theorem 1. *Let $\mathscr{L} \subseteq T(\mathscr{F})$. $\mathscr{L}$ is a k-TSS iff for any trees $t_1, t_2 \in T(\mathscr{F})$ such that $root_k(t_1) = root_k(t_2)$, when $t_1^{-1}\mathscr{L} \neq \emptyset \wedge t_2^{-1}\mathscr{L} \neq \emptyset$ then it follows that $t_1^{-1}\mathscr{L} = t_2^{-1}\mathscr{L}$.*

We choose López et al.'s theorem as a definition of k-TSS languages. Other definitions in the literature [14,22] define k-TSS languages in terms of three sets; leaves, roots, and forks. Forks are roots that have at least one placeholder as a leaf. Theorem 1 shows that such more complex definitions are unnecessary. Intuitively, the theorem says that within the language, any two subtrees that agree on the top k levels are interchangeable, meaning that a bottom-up tree automaton has to remember only a finite amount of history. In the next section, we show that we can define an equivalence relation inducing an automaton accepting a k-TSS language using only our definition of the k-root, as expected from Theorem 1.

4 k-Testable Tree Automata Inference

4.1 Congruence Relation

We begin with our definition of the equivalence relation that is used to induce a state-minimal automaton from a set of trees. The equivalence relation, intuitively, compares trees up to k levels deep, i.e., compares k-roots.

Definition 1 (Root Equivalence Relation $\sim_k$). *For some $k \geq 0$, two trees $t_1, t_2 \in T(\mathscr{F})$ are root-equivalent with degree k, denoted $t_1 \sim_k t_2$, if $root_k(t_1) = root_k(t_2)$.*

Lemma 1. *The $\sim_k$ relation is a congruence (monotonic equivalence) relation of finite index.*

Proof (Sketch). It is obvious that $\sim_k$ is an equivalence relation (reflexive, symmetric, and transitive). Monotonicity can be proven by a simple induction on the height of the two trees being compared and the $root_k$ definition.

The size of a k-root is bounded by M^k, where $M = \max(\{n \mid \mathscr{F}_n \in F, \mathscr{F}_n \neq \emptyset\})$. Each position u in the k-root's domain can be labelled with at most $|\mathscr{F}_{arity(t(u))}|$ symbols. Thus, $root_k$ generates a finite number of equivalence classes, i.e., is of finite index.

As a consequence of Lemma 1, inference algorithms based on the root equivalence relation need not propagate congruences using union-find [10] algorithms, as the root equivalence relation is a congruence itself.

Definition 2 ($\sim_k$-induced Automaton). *Let $T' \subseteq T(\mathscr{F})$ be a finite set of finite trees. The $A^{\sim_k}(T') = (Q, \mathscr{F}, \delta, F)$ automaton induced by the root equivalence relation $\sim_k$ is defined as:*

$$
\begin{aligned}
Q &= \{root_k(t') \mid \exists t \in T' \,.\, \exists u \in dom(T') \,.\, t' = t/u\} \\
F &= \{root_k(t) \mid t \in T'\} \\
\delta_0(f) &= f \quad \text{for } f \in \mathscr{F}_0 \\
\delta_n(f, root_k(t_1), \ldots, root_k(t_n)) &= root_k(f(t_1, \ldots, t_n)) \quad \text{for } n \geq 1, f \in \mathscr{F}_n
\end{aligned}
$$

Corollary 1 (Containment). *From the definition it follows that $\forall k \geq 0 \,.\, T' \subseteq \mathscr{L}(A^{\sim_k}(T'))$. In other words, the $\sim_k$-induced automaton abstracts the set of trees T'.*

Theorem 2. *$\mathscr{L}(A^{\sim_k})$ is a k-TSS language.*

Proof. We need to prove that $\forall t_1, t_2 \in T(\mathscr{F}),\ k \geq 0 \,.\, root_k(t_1) = root_k(t_2) \wedge t_1^{-1}\mathscr{L}(A^{\sim_k}) \neq \emptyset \wedge t_2^{-1}\mathscr{L}(A^{\sim_k}) \neq \emptyset \Rightarrow t_1^{-1}\mathscr{L}(A^{\sim_k}) = t_2^{-1}\mathscr{L}(A^{\sim_k})$. Suppose the antecedent is true, but the consequent is false, i.e., $t_1^{-1}\mathscr{L}(A^{\sim_k}) \neq t_2^{-1}\mathscr{L}(A^{\sim_k})$. Then there must exist t such that $t\sharp t_1 \in \mathscr{L}(A^{\sim_k})$ and $t\sharp t_2 \notin \mathscr{L}(A^{\sim_k})$. Let u be the position of $\xi_{t_2(1)}$, i.e., $(t\sharp t_2)/u = t_2$. Without loss of generality, let t be the tree with minimal $|u|$. Necessarily, $|u| > 1$, as otherwise $t_1^{-1}\mathscr{L}(A^{\sim_k}) = \emptyset$. Let $u = w \cdot i,\ i \in \mathbb{N}$. We prove that $t\sharp t_2$ must be in $\mathscr{L}(A^{\sim_k})$, contradicting the initial assumption, by induction on the length of w.

Base case ($|w| = 1$): Let $(t(w))(1) = f,\ f \in \mathscr{F}_n$. There are two subcases: $n = 1$ and $n > 1$. For $n = 1$, the contradiction immediately follows, as $\delta(f, root_k(t_1))$

$= \delta(f, root_k(t_2))$. For the $n > 1$ case, observe that for all positions $w \cdot j$ such that $1 \leq j \leq n$ and $j \neq i$, $(t\sharp t_1)/w \cdot j = (t\sharp t_2)/w \cdot j = t/w \cdot j$. From that observation and $root_k(t_1) = root_k(t_j)$, it follows that

$$\begin{aligned} &\delta\big((t\sharp t_1/w)(1), root_k(t\sharp t_1/w \cdot 1), \ldots, root_k(t\sharp t_1/w \cdot n)\big) \\ &= \delta\big((t\sharp t_2/w)(1), root_k(t\sharp t_2/w \cdot 1), \ldots, root_k(t\sharp t_2/w \cdot n)\big) \end{aligned}$$

Induction step ($|w| > 1$): Let $w = w' \cdot m,\ m \in \mathbb{N}$. From the induction hypothesis, we know that for all m, $root_k(t\sharp t_1/w) = root_k(t\sharp t_2/w)$, thus it follows:

$$\begin{aligned} &\delta\big((t\sharp t_1/w')(1), root_k(t\sharp t_1/w' \cdot 1), \ldots, root_k(t\sharp t_1/w' \cdot n)\big) \\ &= \delta\big((t\sharp t_2/w')(1), root_k(t\sharp t_2/w' \cdot 1), \ldots, root_k(t\sharp t_2/w' \cdot n)\big) \end{aligned}$$

Theorem 3 (Minimality). *$A^{\sim k}$ is state-minimal.*

Proof. Follows from Myhill-Nerode Theorem [20, pg. 72] and Lemma 1.

Minimality is not absolutely crucial for malware analysis in a laboratory setting, but it is important in practice, where antivirus tools can't impose a significant system overhead and have to react promptly to infections.

Theorem 4 (Garcia [14]). $\mathscr{L}(A^{\sim k+1}) \subseteq \mathscr{L}(A^{\sim k})$

An important consequence of Garcia's theorem is that the k factor can be used as an abstraction knob — the smaller the k factor, the more abstract the inferred automaton. This tunability is particularly important in malware detection. One can't hope to design a classifier capable of perfect malware and goodware distinction. Thus, tunability of the false positive (goodware detected as malware) and false negative (undetected malware) ratios is crucial. More abstract automata will result in more false positives and fewer false negatives.

4.2 Inference Algorithm

In this section, we present our inference algorithm, but before proceeding with the algorithm, we discuss some practical aspects of inference from data-flow dependency graphs. As discussed in Section 2, we use taint analysis to compute data-flow dependencies among executed syscalls at runtime. The result of that computation is not a tree, but an acyclic directed graph, i.e., a partial order of syscalls ordered by the data-flow dependency relation, and expansion of such a graph into a tree could cause exponential blowup. Thus, it would be more convenient to have an inference algorithm that operates directly on graphs, without expanding them into trees.

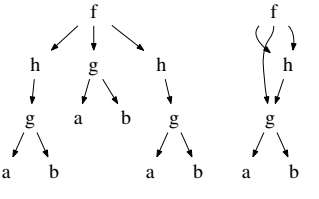

Fig. 2. Folding a Tree into a Maximally-Shared Graph

Fortunately, such an algorithm is only slightly more complicated than the one that operates on trees. In the first step, our implementation performs common subexpression elimination [1] on the dependency graph to eliminate syntactic redundancies. The result

is a maximally-shared graph [2], i.e., an acyclic directed graph with shared common subgraphs. Figure 2 illustrates how a tree can be folded into a maximally-shared graph. In the second step, we compute a hash for each k-root in the training set. The hash is later used as a hash table key. Collisions are handled via chaining [10], as usual, but chaining is not described in the provided algorithms. The last step of the inference algorithm traverses the graph and folds it into a tree automaton, using the key computed in the second phase to identify equivalent k-roots, which are mapped to the same state.

To simplify the exposition, we shall use the formal machinery developed in Section 3 and present indexing and inference algorithms that work on trees. The extension to maximally-shared graphs is trivial and explained briefly later.

input : Tree t, factor k
result : Key computed for every subtree of t

$tmp \leftarrow hash(t(1))$
foreach $1 \leq i \leq arity(t(1))$ **do**
  $t_s \leftarrow t/i$
  $tmp \leftarrow tmp \oplus hash(t_s.key)$
  ComputeKey(t_s, k)
$t.key \leftarrow tmp$

Algorithm 1. ComputeKey — Computing k-Root Keys (Hashes). The $\oplus$ operator can be any operator used to combine hashes, like bitwise exclusive OR. The $hash : \mathscr{F} \rightarrow \mathbb{N}$ function can be implemented as a string hash, returning an integral hash of the alphabet symbols.

Algorithm 1 traverses tree t in postorder (children before the parent). Every subtree has a field *key* associated with its head, and the field is assumed to be initially zero. If the algorithm is called once, for tree t, the key of the head of each subtree t_s will consist only of the hash of the alphabet symbol labeling t_s, i.e., $hash(t_s(1))$. If the algorithm is called twice (on the same tree), the key of the head of each subtree will include the hash of its own label and the labels of its children, and so on. Thus, after k calls to ComputeKey, the key of each node will be equal to its k-root key. Note that the temporary key, stored in the *tmp* variable, has to be combined with the children's $(k-1)$-root key. The algorithm can be easily extended to operate on maximally-shared graphs, but has to track visited nodes and visit each node only once in postorder. The complexity of the algorithm is $\mathscr{O}(k \cdot N)$, where N is the size of the tree (or maximally-shared graph). For multi-rooted graphs (or when processing multiple trees), all roots can be connected by creating a synthetic super-root of all roots, and the algorithm is then called k times with the super-root as the first operand.

Algorithm 2 constructs the $A^{\sim_k}$ automaton. The tree (alternatively maximally-shared graph) used for training is traversed in postorder, and k-root of each subtree is used to retrieve the representative for each $\sim_k$-induced equivalence class. Multi-rooted graphs can be handled by introducing super-roots (as described before). Amortized complexity is $\mathscr{O}(kN)$, where N is the size of the tree (or maximally-shared graph).

input : Tree t, factor k, alphabet $\mathscr{F}$
output: $A^{\sim_k} = (Q, \mathscr{F}, \delta, F)$

foreach *subtree* t_s *in* $\{t/u \mid u \in dom(t)\}$ *traversed in postorder* **do**
  if $rep[t_s.key] = \emptyset$ **then**
    $q \leftarrow root_k(t_s)$
    $rep[t_s.key] = q$
    $Q \leftarrow Q \cup q$
  $n \leftarrow arity(t_s(1))$
  $\delta \leftarrow \Big(\big(t_s(1), rep[(t_s/1).key], \ldots, rep[(t_s/n).key]\big), rep[t_s.key]\Big)$
$F = F \cup rep[t.key]$
return $(Q, \mathscr{F}, \delta, F)$

Algorithm 2. k-Testable Tree Automaton Inference. The $rep : hash(root_k(T(\mathscr{F}))) \rightarrow root_k(T(\mathscr{F}))$ hash map contains representatives of equivalence classes induced by $\sim_k$. Collisions are handled via chaining (not shown).

5 Experimental Results

5.1 Benchmarks

For the experiments, we use two sets of benchmarks: the malware and the goodware set. The malware set comprises 2631 samples pre-classified into 48 families. Each family contains 5–317 samples. We rely upon the classification of Christodorescu et al. [6] and Fredrikson et al. [13].[2] The classification was based on the reports from antivirus tools. For a small subset of samples, we confirmed the quality of classification using virustotal.com, a free malware classification service. However, without knowing the internals of those antivirus tools and their classification heuristics, we cannot evaluate the quality of the classification provided to us. Our classification experiments indicate that the classification antivirus tools do might be somewhat ad-hoc. Table 1 shows the statistics for every family.

The goodware set comprises 33 commonly used applications: AdobeReader, Apple SW Update, Autoruns, Battle for Wesnoth, Chrome, Chrome Setup, Firefox, Freecell, Freeciv, Freeciv server, GIMP, Google Earth, Internet Explorer, iTunes, Minesweeper, MSN Messenger, Netcat port listen and scan, NetHack, Notepad, OpenOffice Writer, Outlook Express, Ping, 7-zip archive, Skype, Solitaire, Sys info, Task manager, Tux Racer, uTorrent, VLC, Win. Media Player, and WordPad. We deemed these applications to be representative of software commonly found on the average user's computer, from a number of different vendors and with a diverse set of behaviors. Also, we used two micro benchmarks: a HelloWorld program written in C and a file copy program. Micro-benchmarks produce few small dependency graphs and therefore might be potentially more susceptible to be misidentified for malware.

[2] The full set of malware contains 3136 samples, but we eliminated samples that were not executable, executable but not analyzable with Pin (i.e., MS-DOS executables), broken executables, and those that were incompatible with the version of Windows (XP) that we used for experiments.

Table 1. Malware Statistics per Family. All dependency graphs were obtained by running each sample for 120sec in a controlled environment. The identifier that will be used in later graphs is given in the first column. The third column shows the number of samples per family. The *Avg.* column shows the average height of the dependency graphs across all the samples in the family. The *Nodes* column shows the total number of nodes in the dependency graph (after CSE). The *Trees* column shows the total number of different trees (i.e., roots of the dependency graph) across all the samples. The *Max* column gives the maximal height of any tree in the family.

ID	Family Name	Samples	Avg.	Nodes	Trees	Max.	ID	Family Name	Samples	Avg.	Nodes	Trees	Max.
1	ABU.Banload	16	7.71	544	303	21	25	Hupigon.AWQ	219	24.63	7225	3758	62
2	Agent	42	8.86	965	593	27	26	IRCBot.Sdbot	66	16.51	3358	1852	47
3	Agent.Small	15	8.88	950	588	27	27	LdPinch	16	16.88	1765	1012	66
4	Allaple.RAHack	201	8.78	1225	761	44	28	Lmir.LegMir	23	9.00	1112	667	28
5	Ardamax	25	6.21	144	69	16	29	Mydoom	15	5.78	484	305	20
6	Bactera.VB	28	7.09	333	177	28	30	Nilage.Lineage	24	9.64	1288	657	83
7	Banbra.Banker	52	13.97	1218	686	37	31	Games.Delf	11	8.44	971	632	22
8	Bancos.Banker	46	14.05	742	417	45	32	Games.LegMir	76	17.18	11892	8184	59
9	Banker	317	17.70	2952	1705	43	33	Games.Mmorpg	19	7.00	654	478	25
10	Banker.Delf	20	14.78	939	521	50	34	OnLineGames	23	7.30	718	687	16
11	Banload.Banker	138	19.38	2370	1332	152	35	Parite.Pate	71	14.31	1420	816	36
12	BDH.Small	5	5.82	348	199	21	36	Plemood.Pupil	32	6.29	330	189	24
13	BGM.Delf	17	7.04	339	199	25	37	PolyCrypt.Swizzor	43	10.32	415	213	30
14	Bifrose.CEP	35	11.17	1190	698	50	38	Prorat.AVW	40	23.47	1031	572	58
15	Bobax.Bobic	15	8.98	859	526	30	39	Rbot.Sdbot	302	14.23	4484	2442	47
16	DKI.PoisonIvy	15	9.22	413	227	40	40	SdBot	75	14.13	2361	1319	40
17	DNSChanger	22	12.62	874	483	36	41	Small.Downloader	29	11.93	2192	1216	34
18	Downloader.Agent	13	12.89	1104	613	49	42	Stration.Warezov	19	9.76	1682	1058	34
19	Downloader.Delf	22	10.76	1486	906	32	43	Swizzor.Obfuscated	27	21.75	1405	770	49
20	Downloader.VB	17	10.80	516	266	29	44	Viking.HLLP	32	7.84	512	315	24
21	Gaobot.Agobot	20	17.54	1812	1052	45	45	Virut	115	11.76	3149	1953	40
22	Gobot.Gbot	58	7.01	249	134	22	46	VS.INService	17	11.42	307	178	37
23	Horst.CMQ	48	16.86	1030	541	42	47	Zhelatin.ASH	53	12.14	1919	1146	39
24	Hupigon.ARR	33	23.58	2388	1244	55	48	Zlob.Puper	64	15.16	2788	1647	90

In behavioral malware detection, there is always a contention between the amount of time the behavior is observed and the precision of the analysis. For malware samples, which are regularly small pieces of software, we set the timeout to 120sec of running in our environment. For goodware, we wanted to study the impact of the runtime on the height and complexity of generated dependency graphs, and the impact of these differences on the false positive rates. Thus, we ran goodware samples for both 120 and 800sec. To give some intuition of how that corresponds to the actual native runtime, it takes approximately 800s in our DTA analysis environment for Acrobat Reader to open a document and display a window.

We noticed a general tendency that detection and classification tend to correlate positively with the average height of trees in samples used for training and testing. We provide the average heights in Table 1.

5.2 Malware and Goodware Recognition

For our malware recognition experiments, we chose at random 50% of the entire malware set for training, and used the rest and the entire goodware set as test sets. Training with $k = 4$ took around 10sec for the entire set of 1315 training samples, and the time required for analyzing each test sample was less than the timing jitter (sub-second range). All the experiments were performed in Ubuntu 10.04, running in a VMware

7.1.3 workstation, running on Win XP Pro and dual-core 2.5GHz Intel machine with 4GB of RAM. In Figure 3a (resp. 3b), we show the results, using the goodware dependency graphs produced with an 800sec (resp. 120sec) timeout.

The detection works as follows. We run all the trees (i.e., roots of the dependency graph) in each test sample against the inferred automaton. First, we sort the trees by height, and then compute how many trees for each height are accepted by the automaton. Second, we score the sample according to the following function:

$$score = \frac{\sum_i \frac{accepted_i}{total_i} * i}{\sum_i i} \quad (1)$$

where i ranges from 1 to the maximal height of any tree in the test sample (the last column of Table 1), $accepted_i$ is the number of trees with height i accepted by the automaton, and $total_i$ is the total number of trees with height i. The test samples that produce no syscall dependency graphs are assumed to have score zero.

The score can range from 0 to 1. Higher score signifies a higher likelihood the sample is malicious. The ratio in the nominator of Eq. 1 is multiplied by the depth of the tree to filter out the noise from shallow trees, often generated by standard library functions, that have very low classification power.

The results turned out to be slightly better with an 800sec timeout than with the 120sec timeout, as the average height of dependency graphs was slightly larger. As expected, we found that with the rising k factor (and therefore decreasing level of abstraction), the capability of inferred tree automaton to detect malware decreases, which obviously indicates the value of generalization achieved through tree automata inference. On the other hand, with the rising k factor, the detection becomes more precise and therefore the false positive rate drops down. Thus, it is important to find the right level of abstraction. In our experiments, we determined that $k = 4$ was the optimal abstraction level. The desired ratio between false positives and negatives can be adjusted by selecting the score threshold. All samples scoring above (resp. below) the threshold are declared malware (resp. goodware). For example, for $k = 4$, timeout of 800sec, and score 0.6, our approach reports two false positives (5%) — Chrome setup and NetHack, and 270 false negatives (20%), which corresponds to an 80% detection rate. For $k = 4$, timeout of 800sec, and score 0.6, our approach reports one additional false positive (System info), and the same number of false negatives, although a few malware samples are somewhat closer to the threshold. Obviously, the longer the behavior is observed, the better the classification.

It is interesting to notice that increasing the value of k above 4 does not make a significant difference in (mis)detection rates. We ran the experiments with k up to 10, but do not show the results as they are essentially the same as for $k = 4$. From our preliminary analysis, it seems that generalization is effective when a sequence of dependent syscalls are executed within a loop. If two samples execute the same loop body a different number of times, our approach will be able to detect that. Changing k effectively changes the window with which such loop bodies are detected. During the inference, it seems like one size (of k) does not fit all cases. We believe that by analyzing the repetitiveness of patterns in dependency graphs, we could detect the sizes of loop bodies much more accurately, and adjust the k factor according to the size of the body, which

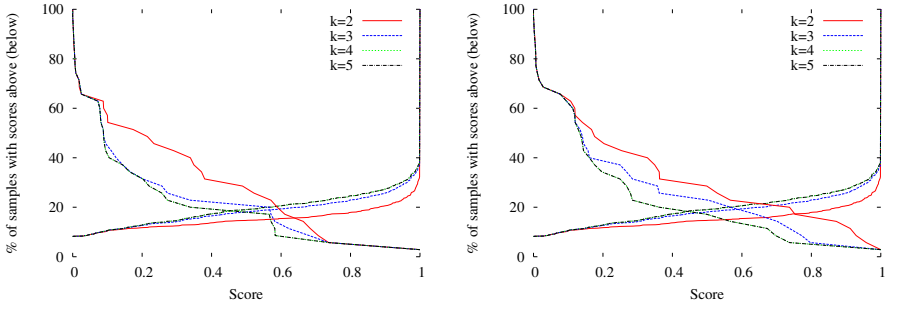

Fig. 3. Malware and Goodware Recognition. Timeouts for generating the dependency graphs were 120sec for malware test and training sets and 800sec (resp. 120sec) for the goodware test set in the figure on the left (resp. right). The training set consists of 50% of the entire malware set, chosen at random. The test set consists of the remaining malware samples (curves rising from left to right), and the goodware set (curves falling from left to right). The rising curves represent the percentage of malware samples for which the computed score was *less* than the corresponding value on the x axis. The falling curves represent the percentage of goodware samples for which the score was *greater* than the corresponding value on the x axis. The figure shows curves for four different values of k, there is essentially no difference between the cases when $k = 4$ and $k = 5$. For the rising curves, the lowest curve is for $k = 2$, the next higher one for $k = 3$, and the two highest ones for the remaining cases. For the falling curves, the ordering is reversed. The optimal score for distinguishing malware from goodware is the lowest intersection of the rising and falling curves for the same k.

should in turn improve the generalization capabilities of the inference algorithm. Many other improvements of our work are possible, as discussed later.

5.3 Malware Classification

We were wondering what is the classification power of inferred automata, so we did the following experiment. We divided at random each family into training and test sets of equal size. For each training set, we inferred a family-specific tree automaton. For each test set, we read the dependency graphs for all the samples in the set, and compute a single dependency graph, which is then analyzed with the inferred tree automaton. The scores are computed according to Equation 1, with $k = 3$. The only difference from the experiment done in the previous section is that the score is computed for the entire test set, not individual samples in the set. Results are shown in Figure 4.

The pronounced diagonal in Figure 4 shows that our inferred automata clearly have a significant classification power and could be used to classify malware into families. There is some noise as well. The noise could be attributed to many factors: over-generalization, over- and under-tainting of our DTA [5,19], insufficiently large dependency graphs, frequently used dynamic libraries that are shared by many applications and malware, and a somewhat ad-hoc pre-classification by the antivirus tools.

6 Limitations

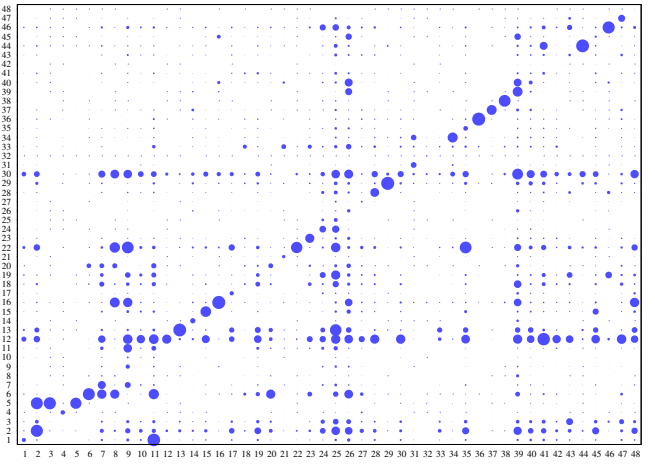

Fig. 4. Malware Classification Results. The x (y) axis represents the training (test) sets. The size of the shaded circle corresponds to the score computed by Eq. 1.

There are several inherent limitations of our approach. An attacker could try to mask syscall dependencies so as to be similar (or the same) as those of benign applications. This class of attacks are known as *mimicry attacks* [33]. All intrusion and behavioral malware detection approaches are susceptible to mimicry attacks. One way to make this harder for the attacker, is to make the analysis more precise, as will be discussed in the following section.

Triggering interesting malware behavior is another challenge. Some behaviors could be triggered only under certain conditions (date, web site visited, choice of the default language, users' actions,...). Moser et al. [28,4] proposed DART [16] as a plausible approach for detecting rarely exhibited behaviors.

As discussed earlier, our DTA environment slows the execution several thousand times, which is obviously too expensive for real-time detection. A lot of work on malware analysis is done in the lab setting, where this is not a significant constraint, but efficiency obviously has to be improved if taint-analysis based approaches are ever to be broadly used for malware detection. Hardware taint-analysis accelerators are a viable option [30,11], but we also expect we could probably achieve an order of magnitude speedup of our DTA environment with a very careful optimization.

7 Conclusions and Future Work

In this paper, we presented a novel approach to detecting likely malicious behaviors and malware classification based on tree automata inference. We showed that inference, unlike simple matching of dependency graphs, does generalize from the learned patterns and therefore improves detection of yet unseen polymorphic malware samples. We proposed an improved k-testable tree automata inference algorithm and showed how the k factor can be used as a knob to tune the abstraction level. In our experiments, our approach detects 80% of the previously unseen polymorphic malware samples, with a 5% false positive rate, measured on a diverse set of benign applications.

There are many directions for further improvements. The classification power of our approach could be improved by a more precise analysis of syscall parameters (e.g., using their actual values in the analysis), by dynamically detecting the best value of the k factor in order to match the size of loop bodies that produce patterns in the dependency graphs, by using goodware dependency graphs as negative examples during training, and by combining our approach with the leap mining approach [13].

Another interesting direction is inference of more expressive tree languages. Inference of more expressive languages might handle repeated patterns more precisely, generalizing only as much as needed to fold a repeatable pattern into a loop in the tree automaton. Further development of similar methods could have a broad impact in security, forensics, detection of code theft, and perhaps even testing and verification, as the inferred automata can be seen as high-level abstractions of program's behavior.

Acknowledgments

We are grateful to Matt Fredrikson and Somesh Jha for sharing their library of classified malware with us. We would especially like to thank Lorenzo Martignoni, who wrote the libwst library [26] for extracting and parsing arguments of Windows's system calls. We also thank reviewers for their insightful and constructive comments.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc. Redwood City (1986)
2. Babić, D.: Exploiting Structure for Scalable Software Verification. Ph.D. thesis, University of British Columbia, Vancouver, Canada (2008)
3. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Architecture of a morphological malware detector. Journal in Computer Virology 5, 263–270 (2009)
4. Brumley, D., Hartwig, C., Zhenkai Liang, J.N., Song, D., Yin, H.: Automatically Identifying Trigger-based Behavior in Malware. In: Botnet Detection Countering the Largest Security Threat, Advances in Information Security, vol. 36, pp. 65–88. Springer, Heidelberg (2008)
5. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proc. of 13th USENIX Security Symp. (2004)
6. Christodorescu, M., Jha, S.: Testing malware detectors. In: ISSTA 2004: Proc. of the 2004 ACM SIGSOFT Int. Symp. on Software Testing and Analysis, pp. 34–44. ACM Press, New York (2004)
7. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proc. of the the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering, pp. 5–14. ACM Press, New York (2007)
8. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: SP 2005: Proc. of the 2005 IEEE Symp. on Security and Privacy, pp. 32–46. IEEE Computer Society Press, Los Alamitos (2005)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
11. Crandall, J., Chong, F.: Minos: Control data attack prevention orthogonal to memory model. In: the Proc. of the 37th Int. Symp. on Microarchitecture, pp. 221–232. IEEE Computer Society Press, Los Alamitos (2005)

12. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: Proc. of the 1996 IEEE Symp. on Security and Privacy, pp. 120–129. IEEE Computer Society Press, Los Alamitos (1996)
13. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: Proc. of the 2010 IEEE Symp. on Security and Privacy, pp. 45–60. IEEE Computer Society Press, Los Alamitos (2010)
14. García, P.: Learning *k*-testable tree sets from positive data. Tech. rep. In: Dept. Syst. Inform. Comput. Univ. Politecnica Valencia, Spain (1993)
15. García, P., Vidal, E.: Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. IEEE Trans. Pattern Anal. Mach. Intell. 12, 920–925 (1990)
16. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. pp. 213–223. ACM Press, New York (2005)
17. Gold, E.M.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
18. Holzer, A., Kinder, J., Veith, H.: Using verification technology to specify and detect malware. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 497–504. Springer, Heidelberg (2007)
19. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp.San Diego, CA (2011)
20. Khoussainov, B., Nerode, A.: Automata Theory and Its Applications. Birkhäuser, Basel (2001)
21. King, J.C.: Symbolic execution and program testing. Comm. of the ACM 19(7), 385–394 (1976)
22. Knuutila, T.: Inference of *k*-testable tree languages. In: Bunke, H. (ed.) Advances in Structural and Syntactic Pattern Recognition: Proc. of the Int. Workshop, pp. 109–120. World Scientific, Singapore (1993)
23. Kolbitsch, C., Milani, P., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: The 18th USENIX Security Symp. (2009)
24. López, D., Sempere, J.M., García, P.: Inference of reversible tree languages. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(4), 1658–1665 (2004)
25. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of the 2005 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl., pp. 190–200. ACM Press, New York (2005)
26. Martignoni, L., Paleari, R.: The libwst library (a part of WUSSTrace) (2010), `http://code.google.com/p/wusstrace/`
27. Matrosov, A., Rodionov, E., Harley, D., Malcho, J.: Stuxnet under the microscope. Tech. rep. Eset (2010)
28. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: SP 2007: Proc. of the IEEE Symp. on Security and Privacy, pp. 231–245. IEEE Computer Society Press, Los Alamitos (2007)
29. Newsome, J., Song, D.: Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In: Proc. of the Network and Distributed Systems Security Symp. (2005)
30. Suh, G., Lee, J., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. ACM SIGOPS Operating Systems Review 38(5), 85–96 (2004)
31. Symantec: Symantec global internet security threat report: Trends for 2009. Tech. rep.Symantec, vol. XV (2010)
32. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proc. of the IEEE Symp. on Security and Privacy, p. 156. IEEE Computer Society Press, Los Alamitos (2001)
33. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proc. of the 9th ACM Conf. on Comp. and Comm. Security, pp. 255–264. ACM Press, New York (2002)
34. You, I., Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In: Int. Conf. on Broadband, Wireless Computing, Communication and Applications, pp. 297–300 (2010)
35. Zalcstein, Y.: Locally testable languages. J. Comput. Syst. Sci. 6(2), 151–167 (1972)