A Quantitative Study of Accuracy in System Call-Based Malware Detection

Davide Canali
EURECOM, France
canali@eurecom.fr

Andrea Lanzi
EURECOM, France
lanzi@eurecom.fr

Davide Balzarotti
EURECOM, France
balzarotti@eurecom.fr

Christopher Kruegel
UC Santa Barbara, US
chris@cs.ucsb.edu

Mihai Christodorescu
IBM T.J. Watson, US
mihai@us.ibm.com

Engin Kirda
Northeastern University, US
ek@ccs.neu.edu

ABSTRACT

Over the last decade, there has been a significant increase in the number and sophistication of malware-related attacks and infections. Many detection techniques have been proposed to mitigate the malware threat. A running theme among existing detection techniques is the similar promises of high detection rates, in spite of the wildly different models (or specification classes) of malicious activity used. In addition, the lack of a common testing methodology and the limited datasets used in the experiments make difficult to compare these models in order to determine which ones yield the best detection accuracy.

In this paper, we present a systematic approach to measure how the choice of behavioral models influences the quality of a malware detector. We tackle this problem by executing a large number of testing experiments, in which we explored the parameter space of over 200 different models, corresponding to more than 220 million of signatures. Our results suggest that commonly held beliefs about simple models are incorrect in how they relate changes in complexity to changes in detection accuracy. This implies that accuracy is non-linear across the model space, and that analytical reasoning is insufficient for finding an optimal model, and has to be supplemented by testing and empirical measurements.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness); C.4 [**Performance of Systems**]: Measurement techniques; K.6.5 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses)

General Terms

Security, Experimentation, Measurement

Keywords

Security, evaluation, malware, behavior

1. INTRODUCTION

When studying the published results of existing research on behavior-based detectors [5, 6, 8–10, 14, 20, 23], one can make three interesting observations. First, despite their differences, all proposed malware detectors seem to work quite well. The detection rate is typically very high, with very few (or even zero) false positives. The second observation is that, even though each solution is based on a different *behavioral model* (i.e., type of program abstraction), a clear motivation of why a particular model was chosen is often missing. Finally, the data sets which were used for the experimental evaluations of these systems were often limited to very few benign and malicious samples, calling into question whether the excellent detection results reported would still hold with more extensive and systematic experiments.

The motivation behind these research approaches to behavior-based detection were the increasing limitations exhibited by anti-malware techniques. Recent high-profile incidents such as Operation Aurora [12] and Stuxnet [7] are instructive examples of the ongoing shift towards stealth and evasion in the malware industry. Anti-malware techniques often rely on byte signatures (i.e., strings or regular expressions of bytes in the malware binary file), which are easily evaded by simple code changes (e.g., run-time packing, obfuscation), as previous work has demonstrated [4,16]. Behavior-based methods rely instead on higher-level, more abstract representations of malicious code, usually using system calls instead of instruction bytes, with the often-stated explanation that system calls capture intrinsic characteristics of the malicious behavior and thus are harder to evade.

Unfortunately, simply transitioning a specification of malicious behavior from using bytes or instructions to using system calls does not guarantee more accurate and more resilient detection. The structure of this specification is important, in terms of how the system calls are organized (i.e., as a sequence of calls, as an unordered set, as a dependence graph), how many system calls are part of the specification, how much of the specification has to match for detection to succeed, etc. The existing work provides data points for different models to consider, but there is no clear understanding of which specification models are most suitable for detection given that malware writers continuously adapt their work to be stealthy and to evade detection. Finally, all these approaches have been tested on very limited datasets including only a handful of benign applications run by the authors on a single machine in a controlled environment.

To overcome this limitation, our goal in this paper is to develop a methodology to systematically test and compare the effectiveness of different models to capture program behavior. We argue that, in order to obtain a better understanding of how the choice of the behavioral model and its parameters influences the quality of malware detection, the detection capabilities of different behavior models have to be tested on a realistic and large-scale data sets. In addition, in order to provide a better understanding of *why* a particular model works better than others and under which circumstances, our methodology organizes the space of behavioral models along three dimensions: the granularity of atomic program operations, the ways in which these operations relate to each other, and the number of operations included in a specification.

As program operations constitute the basic building blocks of models, we refer to them as *behavior atoms*. In this work, we focus on using system calls, with and without parameters and with various levels of abstraction, as the atomic operations that models can use to characterize program behavior.

The second dimension captures the ways in which behavioral atoms can be structured and combined together. In particular, we are interested in the ordering constraints that models can formulate on top of these atoms, and the possible structures that these constraints yield. Together these dimensions allow us to explore empirically the design space of specifications over system calls, from unordered sets, to simple sequences, and to general finite state automata.

Given the universe of all possible behavior models that can be expressed based on our atoms and structures, we *explore this space* by running over 200 different testing experiments against several large and diverse datasets of malicious and benign programs. The datasets consist of program execution traces observed both in a synthetic environment (based on Anubis [1]) and on real-world machines with actual users and under normal operating conditions. By using this mix, we ensure that the representations we consider are not biased towards particular runtime environments, or particular usage patterns. Our datasets consist of a total of 1.5 billion system-calls invoked by over 363,000 unique process executions. This exploration process leads us to observe the existence of limit points beyond which accuracy cannot possibly improve and the non-linearity of accuracy with model parameters. It is clear that only an exhaustive (and experimentally-supported) approach can provide sufficient information to allow us to reason about models in comparison. Thus, the impossibility of generalizing results in a closed form for finite-state models is one of the main contributions of the paper.

We summarize our contributions as follows:

- We present a systematic testing technique to evaluate the quality of behavioral-based detection models. In our experiments, we explore the space of behavioral models by generating over 200 different detection models and over 220 million of specifications, and test each one against large, real-world datasets.
- We provide empirical evidence that accuracy varies non-linearly with any parameters of the specification model space. Our paper shows how, by attempting to generalize results in a closed form, it is easy to fall into common pitfalls. Therefore, any proposed model has to be driven by a comprehensive experimental validation.
- We provide empirical evidence that the training set (i.e., the benign and malicious samples used to construct a specification) has a large influence on the resulting accuracy. This means that any analytically developed detection scheme has to be supplemented with experimental support from large data sets.

Finally, we hope that the methodology and the results presented in this paper will provide a benchmark for future malware detector proposals and research efforts.

2. OVERVIEW

Consider the following scenario: A malware analyst is given the task of deriving a representation from the malware "catch," which nowadays can run at over 55,000 new samples per day [15]. The analyst has to construct an optimal representation from the malware and benign datasets on hand, and then he has to translate the result into the format that is understood by the detection engine (possibly creating multiple signatures in that format). The challenge is to find an optimal representation for the large set of malware samples.

If the resulting representation is to be used in a byte-signature antivirus engine, then the easiest path to get there is to derive the byte signature that best covers the given malware set. Unfortunately, this simple approach can result in a suboptimal outcome because it tries to capture the common behavior in a fairly rigid representation. Suboptimal in our case means that the newly derived byte signature will suffer from false positives or false negatives (and likely from both). Therefore, as attackers started using obfuscation strategies, detectors were forced to move toward more complex representations. The first evolution consisted in using regular expressions over byte sequences [21], approach that quickly became obsolete as byte patterns have little predictive power (i.e., they can accurately capture only previously seen malware) and are not resistant to evasion and obfuscation techniques. Other static models, such as byte n-grams [13], system dependencies of the program binary [19], and syntactic sequences of library calls [17,22] have also been proposed, but they had limited success.

Researchers have also tried to describe malware in terms of violations to an information-flow policy. Because it is not feasible for performance reasons to track system-wide information flows accurately, the focus shifted on better and better approximations of the information flow. For example, Bruschi et al. [3] and Kruegel et al. [11] have shown that some classes of obfuscations could be rendered innocuous by modeling programs according to their instruction-level control flow. At the same time, Christodorescu et al. [5] and Kinder et al. [8] built obfuscation-resilient detectors based on instruction-level information flow. However, extracting and or collecting instruction-level information is either very hard (from a static point of view) or very inefficient (from a dynamic perspective).

To avoid the previously mentioned limitations and achieve a precise and harder to evade malware characterization, recent research has focused on detection techniques that model the runtime behavior of malware samples [6,9,10,14,20]. To better illustrate the challenge of extracting a behavioral signature we can use the simple example of Figure 1. On the

```
NtOpenKey("SYSTEM\Cu ... 70B}", 131097)
NtQueryValueKey(1640, "EnableDHCP", 2)
NtQueryValueKey(1640, "DhcpServer", 2)
NtQueryValueKey(1640, "DhcpServer", 2)
NtClose(1640)
NtCreateFile("\\Device\ ... 70B}", 3, 0)
NtClose(1640)
```

(a) Short system-call trace.

s_1: `NtOpenKey`
s_2: `NtOpenKey("SYSTEM\Cu ... 70B}", 131097)`
s_3: $\langle$ `NtOpenKey, NtQueryValueKey` $\rangle$
s_4: $\{$ `NtOpenKey, NtQueryValueKey` $\}$
s_5: $[$ `NtOpenKey("SYSTEM\Cu ... 70B", 131097), ...,`
`NtQueryValueKey(1640, "EnableDHCP", 2)` $]$

(b) Five behavioral specifications derived from (a).

Figure 1: A program execution trace and five examples of behavioral specifications that match it.

left side, there is a brief system-call trace from a program execution, similar to what an analyst would see if he were to run the malware samples in a honeypot. On the right side, we list five possible behavioral representations that match this system-call trace. Each of these behavioral representations is a candidate for use in a malware detector. The first representation, s_1, matches all programs whose execution traces include an invocation of `NtOpenKey`, irrespective of its arguments, while s_2 will match only if the invocation has the specified arguments. Behavioral representations s_3 and s_4 match programs that invoke `NtOpenKey` and `NtQueryValueKey` in sequential order and any order, respectively. The fifth representation, s_5, matches all programs that invoke `NtOpenKey` with the specified arguments, followed by any number of arbitrary system calls, followed by an invocation of `NtQueryValueKey` with the specified arguments.

Each of the behavioral representations in Figure 1(b) differs in its expressive power. For example, s_1 is less specific than s_2 because it does not impose any constraints on program arguments, so a detector using s_1 will likely have higher detection rate and higher false positive rate than one using s_2. Thus, in a practical scenario, the analyst is faced with the problem of finding the optimal behavioral representation from an extremely large set of closely related, yet slightly different candidates. The automation of this kind of exploratory task of searching for and comparing candidates is exactly what we focus on in this paper.

Table 1 summarizes how the problem was solved by some of the most relevant previous publications in the area of behavioral malware detection. Some of the proposed approaches are specific to detect particular classes of malware, e.g., spyware [9] or botnets [20], while others have a broader scope that can cover different domains. The table contains several columns, reporting information about the data source used to build the models and the structure of the model themselves. For instance, Christodorescu et al. [6] use automatically constructed models based on direct acyclic graphs of system calls, while Kirda et al. [9] manually selected a subset of potentially dangerous API calls.

The last two columns of table 1 report the size of the malware and the benign datasets used to test the detection and false positive rates. Unfortunately, these sets are very small and often collected on a single machine in a controlled environment. For instance, MiniMal was tested on six benign applications run for a maximum of 2 minutes each. Finally, the *efficiency* column describes if the proposed solution is more suitable for runtime detection or for offline malware analysis. We marked a technique as "Detection" if the authors present experiments to support a possible end-user installation of their solution, without taking into account the real performance overhead (that, unfortunately, was often too high to be used in a real environment). In fact, in many cases the authors presented solutions based on complex models (such as graphs enriched with program slices, or multi-layered graphs) that are hard to apply in realtime.

Even though all these approaches seem to provide acceptable results, a motivation of why such complex models are required is still missing. Are simpler behavioral models inadequate or insufficient to detect malware? Which ones are their intrinsic limitations?

To answer these questions, in this paper, we present a bottom-up testing methodology to evaluate behavioral-based representations.

3. ATOMS, SIGNATURES, AND MODELS

We describe a detection specification as a *model* that combines a number of *signatures*, where each signature is formed from *atoms* that represent basic program operations in a particular temporal and state relation to each other. Conceptually, signatures correspond to low-level program behaviors (e.g., reading a system-configuration file), while models correspond to high-level program behaviors (e.g., exfiltration of sensitive configuration data) that arise from the coupling of low-level operations.

A *signature* captures a program behavior in terms of the relevant program operations together with the relationships between them. Signatures are the basic blocks used in the malware detection process, where the core operation is the matching of a program against a signature.

According to our approach, we define a signature based on the following three elements:

- A *signature atom* is the fundamental behavioral element that appears in a program trace. For example, program instructions, library calls, system calls, and system calls together with their parameters are all possible signature atoms.
- A *signature structure* describes how the atoms must be ordered, and in what trace contexts they may appear for a match to occur. An example of a structure is a *set*, where atoms from the set can be matched in any order, and within any context in an execution trace.
- A *signature cardinality* defines how many atoms are included in the structure.

We write $\mathcal{A}$ to represent the set of all atoms. A program matches a signature if the program matches the signature atoms according to the structure of the signature.

Definition 1 (Signature Matching). *A program P matches a signature $s = \langle A, \Gamma, \alpha \rangle$ if an execution trace of the*

Table 1: Summary of the main behavioral-based approaches

Approach	Data Source	Model Type	Model Extraction	Efficiency	Malware Set	Benign Set
Kirda et al. [9]	API Calls	API Blacklist	Manual	Analysis	33	18
MiniMal [6]	Syscalls	Graph	Automated	Detection	16	6
BotSwat [20]	System & API Calls	Tainted Args in Selected Calls	Manual	Detection	6	8
Martignoni et al. [14]	Syscalls	Layered Graphs	Manual	Analysis	7	11
Kolbitsch et al. [10]	Syscalls	Graph	Automated	Detection	563 (6 families)	5

program P contains each atom in the set A only in the order specified by Γ and separated only by atom strings specified by α, where:

- *$A \subseteq \mathcal{A}$ is a set of atoms,*
- *$\Gamma \subseteq A \times A$ is an order relation between atoms, and*
- *$\alpha : \Gamma \times \mathcal{A}^+ \mapsto \{\text{true}, \text{false}\}$ is a matching filter restricting what atom substrings the program execution trace can contain between two matched signature atoms (i.e., $\alpha\big((s_i, s_j), x\big) = \text{true}$ means that a match is allowed on a program trace containing the substring $s_i x s_j$, for some string of atoms x).*

Note that α and Γ together capture what we informally refer to as the structure of a signature.

A *model* is defined by a set of signatures and an alert threshold. The alert threshold defines how many different signatures must be matched by a program before raising an alert. For example, it is possible to require 4 different sets of 7 system calls, or 15 sequences of 3 high-level actions. We note that models do not require that signatures are matched in any particular order, but just that a minimum number (given by the alert threshold) of signatures are matched.

We can now define what it means for a program to match a model. Intuitively, a program matches a model if it matches its signatures (or at least a minimum number of them).

Definition 2 (Model Matching). *A program P matches a model $\mathcal{M} = \langle \mathcal{S}, t \rangle$, where $\mathcal{S}$ is a set of signatures $\mathcal{S} = \{s_1, \ldots, s_N\}$ and t is the alert threshold, if the program P matches each signature in some set $\{s_{j_1}, \ldots, s_{j_t}\} \subseteq \mathcal{S}$, with $1 \leq j_1, \ldots, j_t \leq N$.*

The definitions for signature and model are sufficiently generic to encompass the vast majority of behavioral representations that have been used or proposed for malware detection. For example, byte signatures are models using program instructions as atoms, with a total ordering relation, and a matching filter that always returns true. Malspecs (aka system-call dependency graphs) are models using system calls as atoms, with a partial ordering relation, and a matching filter that determines whether a depedency relation is preserved by a sequence of system calls. Thus, our model definition gives us the key dimensions along which to explore accuracy.

The number of all the potential models (i.e., the combination of all the possible parameters) that can be extracted from a program is extremely large. The alert threshold is bounded from above by the total number of possible signatures of a particular type. The cardinality of the signature is, in the limit, bounded by the maximum number of atoms in the samples. Finally, the number of signature structures does not even have a theoretical upper bound (given that both the order relation and the matching filter are arbitrary).

4. MODEL CONSTRUCTION

The fact that the space of all possible models is infinitely large prevents a complete testing exploration in any meaningful sense. We choose, therefore, to limit our exploration to a well-defined region containing the models of practical importance. In this section, we first discuss our choices to constrain the exploration, and then describe our methodology for exhaustively covering the resulting space and efficiently comparing the accuracy of models.

4.1 Restricting the Model Space

We define limits for each of the four parameters that characterize a model: atoms, structures, number of signatures, and alert thresholds.

4.1.1 Atoms

In our experiments we consider four types of atoms: system calls, system calls with arguments, actions, and actions with arguments. An action corresponds to a higher-level operation (e.g., reading a file, or loading a library) and it is obtained by grouping together a set of similar system calls (as illustrated in Table 2). For example, reading a file requires more than one low-level operations (at least one to open the file, and one or more to read the content). The action atoms have been defined by us, based on the traditional grouping of system calls by Microsoft into different classes [18].

4.1.2 Structures

We consider three possible structures to combine atoms together: n-grams, tuples, and bags.

An n-gram is a sequence of n atoms that appear in consecutive order in the program execution trace. In the terms of the Definition 1, an n-gram is given by the total order relation $\Gamma = \{(a_1, a_2), \ldots, (a_i, a_{j>i}), \ldots, (a_{n-1}, a_n)\}$ and the matching filter α equal to *false* everywhere.

A bag of cardinality n (or n-bag) contains n atoms without any particular order relation. Matching between a program and a n-bag signature consists of checking whether each atom in the bag appears at least once in a program execution trace. Formally, a bag signature has an empty order relation, $\Gamma = \emptyset$, and an always-*true* matching filter α.

A tuple signature of cardinality n (n-tuple) combines the strict order relation of n-gram with the always-*true* matching filter of bag signatures. A tuple signature is matched by

Table 2: The four types of atoms considered in this study.

Atom Type	*Examples*
system call	`NtOpenFile`, `NtClose`, ...
system call with arguments	`NtOpenFile(104860, "C:\WINDOWS\system32\Msimtf.dll", 5, 96)`, `NtClose(100)`
action	`ReadFile`, `LoadLibrary`, ...
action with arguments	`ReadFile("C:\WINDOWS\REGISTRATION\R00000000000F.CLB")`, `Loadibrary("KNOWNDLLS\NTDSAPI.DLL")`

a program if its atoms appear in order, but at any distance from each other, in the program execution trace.

We also consider more complex structures derived from combining the three basic structures with themselves in a recursive way. For example, it is possible to build models containing bags of n-grams, n-grams of tuples, or any other combination of the basic structures. We extend our definitions from signatures over atoms to signatures over signatures in the natural way. For example, a k-bag of n-grams is a signature with a bag structure and k n-grams as elements. Matching naturally extends in a similar way, where matching a signature requires the recursive matching of its component signatures. In the case of a bag of k n-grams, the matching requires matching each of the k n-grams in any order.

We observe that not all the structure combinations result in new signature types. For example, n-grams of n-grams, bags of bags, and tuples of tuples do not add complexity because they are equivalent to increasing the cardinality of the basic structure. Other combinations may instead generate models with a confused semantic. For example, an n-gram of bags would require to match in a strict order (without anything else in between) two groups of system calls that can appear in any order and at any distance between each other. In our experiments we excluded these ambiguous cases and we focused on the following non-trivial combinations: bags of n-grams, bags of tuples, tuples of bags, and tuples of n-grams.

These structures may look too simple and "behind the times" when compared with graph models recently introduced in literature (often adopting graph-based structures). However, this focus is deliberate, for two reasons. First, the basic models that we consider should be comprehensively assessed for their limitations before new research delves into increasingly more complex models. Second, combinations of our basic structures (n-grams, bags, and tuples) have the same expressive power as loop-free DFAs and NFAs (although we allow for whole-alphabet self-loops, i.e., $\Sigma^\star$). Intuitively, for example, it is possible to enumerate all paths in a loop-free DFA by using a number of *tuple* signatures.

4.1.3 Signature Cardinality

We decided not to put an upper bound on the number of atoms that can be included in a signature. Instead, we performed an exhaustive exploration by first generating all models with 1 atoms, then all models with 2 atoms, etc. Since this upper bound is extremely high, we kept exploring this direction until the resulting models became too inaccurate to serve any purpose (see Section 5 for more details about the stopping criterion we adopted in our experiments).

4.1.4 Alert Thresholds

A model's alert threshold is naturally limited by the number of signatures in the model. In our experiments, we evaluate all values of the alert threshold, even though most of the useful results are obtained with thresholds below 1,000.

4.2 Rationale

One might wonder at this point whether we are restricting ourselves to a set of representations that are too simple to be useful in a malware detector. There is a long history of projects that used finite automata or richer models derived from finite automata to capture program behavior. We observe that models built on recursive combinations of structures can easily reach the same expressivity as finite automata. An additional reason for our focus on basic structures is that we want to restrict our analysis to behavior representations that can be enforced in real time on end users' machines. This excludes representations that need to collect detailed, "inner" information about a program's execution, including information about individual instructions that are executed and detailed data flows between system calls. Collecting such data (in particular, taint information) requires a special runtime for program execution, and it incurs a significant performance penalty. While data flow tracking mechanisms are invaluable for the fine-grained *analysis* of malware (in a controlled environment), they are typically impractical for malware detectors running on the user's machine. We argue that the mix of our basic structures and the way they can be combined together provides models with significant flexibility and expressiveness to capture program behaviors.

4.3 Exploration of the Model Space

Our testing approach is fundamentally experimental, in that we evaluate models against each other by comparing their associated detectors' accuracies against test sets of malware and benign programs. A detailed description of our experimental methodology appears in Section 5.

4.3.1 Number of Possible Signatures

For a program execution trace of length M, there are $M-n+1$ possible sequences of length n. Since this number is usually not too high, we are able to construct them all without applying any approximations. The number of bags of cardinality n for an execution trace containing x *unique* atoms is given by $\binom{x}{n}$. While the set of unique atoms is limited in the case of system calls and actions (since, for example, Microsoft Windows has no more than 350 system calls, depending on the version), the domain of atoms explodes in size for the types of atoms with arguments (i.e., system calls with arguments and actions with arguments).

For tuples, the problem is even bigger. From an execution trace containing x atoms, it is possible to extract $\binom{x}{n}$ n-tuples.

As it is evident from these simple computations, even extracting all models based on simple structures is often impractical, as the number of signatures to be evaluated grows factorially in the size of the execution trace. Thus, we have to resort to some sort of pruning technique to reduce the number of signatures generated, and therefore the number of test to execute.

4.3.2 Experimentally Pruning the Signature Generation

If the number of atoms is high, generating all bags is computationally infeasible. In such cases, we apply the following pruning rule in order to use information gleaned from models considered up to now to reduce the number of models we consider in the future. A new signature is generated and considered for use in a model if and only if it covers some minimum number M_{min} of malware samples that are not already covered by a minimum number S_{min} of existing signatures. In our experimental evaluation, we use $M_{min} = 5$ and $S_{min} = 20,000$. In other words, for our purpose, a new signature is generated only if it covers at least 5 samples that are not already covered by more than 20,000 existing signatures.

The meaning of these thresholds is as follows. The first threshold (M_{min}) is in place to prevent overfitting (i.e., the creation of signatures that only detect malware samples that are already covered by other signatures). Such signatures are not general enough and, therefore, will not contribute to the detection rate. The second threshold (S_{min}) is used to prevent the generation of too many signatures for a single sample. This ensures that the generated signatures are sufficiently diverse to cover different subsets of the malware test set.

We found this simple pruning rule, controlled by only two parameters, to be sufficient in reducing the signature numbers to a manageable level. Because it requires that each new signature has enough "support" in the test set, we know that signatures that are not generated are guaranteed not be part of an optimal model.

5. MODEL EVALUATION

To evaluate the effectiveness of each model in distinguishing between malicious and benign behaviors, we performed a large number of experiments. In each test, we extracted the set of signatures according to the approach described below, and we then measured the number of false alarms and the number of detected malware samples. In particular, we grouped the false positives by application (i.e., a false-positive rate of 1% means that 1 application out of 100 was erroneously flagged as malicious), instead of counting individual executions that were detected as malicious. Similarly, the detection rate is measured per sample so that a 1% detection rate indicates that 1 malware sample out of 100 was correctly detected by the signatures set.

5.1 Testing Methodology

We used four different datasets in our experiments. The first is a collection of execution traces of 6,000 malware samples randomly extracted from Anubis [1]. This set, that we call `malware`, includes a mix of all the existing categories (botnets, worms, dropper, Trojan horses, ...), drawn from malware that is active in the wild. The second dataset contains 180 GB of execution traces collected from 10 different real-world machines, where we observed normal day-to-day operation of regular computer users. We label this set `goodware`. The third dataset (called `anubis-good`) contains the traces of 36 benign application executed under Anubis. Finally, we used a dataset of execution traces for 1,200 malware samples that have been collected on a different machine than the ones normally used for Anubis (we call this `malware-test`).

The purpose of having execution traces from different machines for both malware and benign programs is to eliminate any machine-specific artifacts (e.g., machine IDs, user IDs) that introduce noise in our results in the form of falsely discriminative atoms.

The `malware` dataset, `anubis-good` dataset, and traces for nine out of the 10 machines in the `goodware` dataset are used to build the models. The traces from the tenth machine in the `goodware` dataset and the `malware-test` dataset are instead used to evaluate the models. The choice of the nine machines used for model construction is done using 10-fold cross-validation approach. The evaluation results presented here are averages across the 10 tests. For any given structure and cardinality, extracting the best model works as follows:

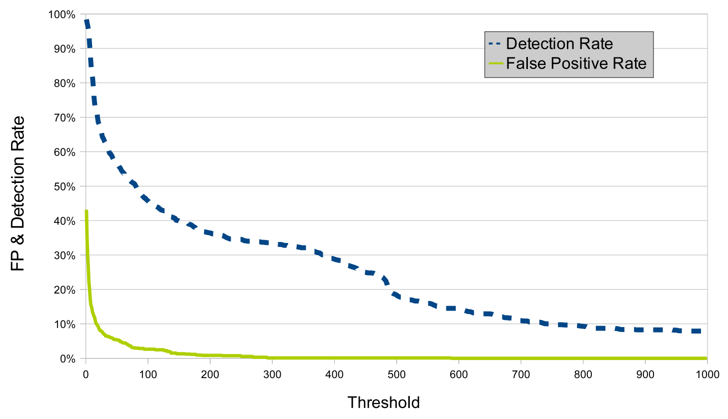
1. We extract the models from the `malware` set. This phase may include pruning to limit the maximum number of signatures generated to a manageable level.
2. We remove from the models the ones that match the `anubis-good` dataset. This is required to be able to automatically "clean" our signatures of Anubis-specific items (e.g., the name of the sample file, usernames, etc.).
3. We create the signature by removing from the models extracted so far the ones that match on 9 out of 10 `goodware` machines.
4. We test the false positive of the signature set on the 10th machine, and the test detection rate on the `malware-test` dataset. The results are extracted for all possible values of the matching threshold.
5. We repeat Steps 3 and 4 for a total of 10 times, each time excluding a different goodware machine. At the end, we compute the accuracy by calculating the average of both the detection and the false-positive rate between the 10 experiments.

5.1.1 Experimental Setup

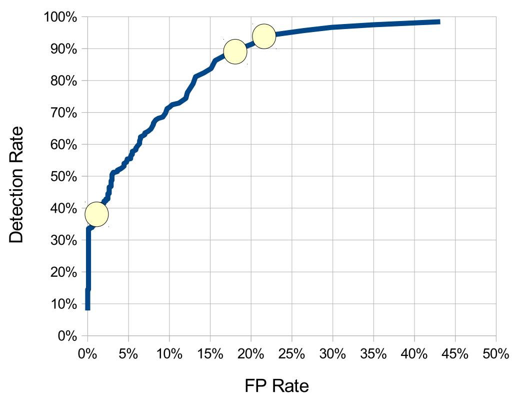
All the experiments were executed using two clusters: one with eight 4-core Xeon(R) machines with 16 GB of RAM each, and the second with eight 16-core AuthenticAMD machines with 45 GB of RAM.

5.1.2 Model Construction and Performance Evaluation

In our experiments, we automatically constructed and tested a total of 215 different detection models, containing a total of more than 220 million of signatures. For each of them, we performed 10 cross-validation experiments and we averaged the results. This required more that 2,100 different unique tests. In addition, we designed our matching algorithm to test in parallel for all possible values of the matching threshold.



(a) Detection and false-positive rates by alert threshold.



(b) ROC curve.

Figure 2: Accuracy for models of 4-grams of system calls with arguments.

Each experiment required two distinct phases: one to extract the signatures, and one to evaluate the models to measure detection rate and false positives. Extracting the signatures to cover the training set of 6,000 malware samples required a substantial computational power. The time required for this task greatly depends on the complexity of the signatures, and on the number of distinct atoms. It ranged from 20 minutes for n-grams of system calls without parameters to almost 2 days for tuples of system calls with parameters.

Another important factor to take into consideration when evaluating a certain model is the time required by a malware detector to perform the matching. For this reason, we conducted a separate experiment to measure the matching speed on a standard desktop computer (Intel dual Core 2.66 GHz with 4 GB of RAM). We tested our prototype malware detector on execution traces covering twelve hours of user activity and recorded the time and amount of memory required by the detection process. Our results shows that the memory consumption may quickly become a problem when the number of signatures increases (reaching around 1GB for 5 millions signatures). Since the number of signatures is strictly related to the signature's cardinality, using signatures containing too many elements may become impractical in a real world deployment.

5.1.3 Experimental Strategy

For each experiment, we plotted a graph depicting the detection and false positive rates for all possible values of the matching threshold. For example, Figure 2(a) shows the results obtained with 4-grams of system calls with arguments. The graph shows that both false-positive and detection rates decrease when we increase the number of signatures that we require to match in any given sample. With a threshold of 1, the model can detect around 98% of unknown malware samples, but also misclassify almost 45% of the benign applications as malicious. In order to get a false positive rate close to zero, we would need to significantly increase the threshold, thus affecting also the detection rate that would drop down to a mere 35%. While this graph is useful in showing the overall effect of the threshold, it is difficult to tell which is the right value to chose in order to get the best trade-off between false positive and detection.

To answer this question, we also plotted the results of each experiment as a ROC curve, in which the X axis represents the false-positive rate, and the Y axis represents the detection rate. For example, Figure 2(b) shows the ROC curve obtained with the same model of 4-grams of system calls with arguments.

These two graph types are useful to summarize the effectiveness of a given model. However, our experiments generated hundreds of such graphs, and it is not obvious how they can be automatically compared in an objective way. For this reason, we decided to summarize each experiment using three indicators that represent different ways to choose the best configuration on the ROC curve. The first value (V_{max}) corresponds to the configuration that maximizes the area under the curve. The second value (V_1) is the one in which the model provides a 1% false-positive rate. Finally, the third value we considered (V_{90}) corresponds to the configuration that provides a 90% detection rate.

By studying how these three indicators (emphasized by small circles in the ROC curve in Figure 2(b)) change, it is possible to quickly compare different models, or the effect of different parameters in a given model.

In the rest of the section, we discuss the results of our empirical evaluation with respect to these three indicators (V_{max}, V_{90}, and V_1), and highlight several interesting discoveries along the way.

5.2 Global Comparison

The main objective of our experiments is to study the impact of different parameters on the effectiveness of a malicious behavior detector. Table 3 shows a summary of the best results across all the basic structures and atom types. The model providing the best results is the "2-bags of 2-tuples of actions with arguments", closely followed by the "2-tuple of actions with arguments" and the "4-bags of actions with arguments". The first model was able to detect

Table 3: Evaluation summary of different types of models.

Model	Cardinality Range	V_{max}	Best Cardinality	V_{90}	V_1
n-grams of syscalls	2–40	0.615	10	31.7%	4.1%
n-grams of syscalls with args	2–40	0.775	3	15.8%	43.3%
n-grams of action	2–75	0.423	15	62.2%	0.4%
n-grams of action with args	2–75	0.737	2	27.1%	45.9%
bags of syscalls	1–10	0.127	3	–	12.8%
bags of syscalls with args	1–20	0.736	1	26.4%	43.3%
bags of actions	1–10	0.004	1	–	–
bags of actions with args	1–15	0.970	4	0.4%	97.3%
tuples of syscalls	2–10	–	–	–	–
tuples of syscalls with args	2–10	0.616	2	–	28.0%
tuples of actions	2–10	–	–	–	–
tuples of actions with args	2–10	0.987	2	0.0%	99.2%
bags of n-grams of syscalls	2–4/2–4	0.500	2/2	–	8.2%
bags of n-grams of syscalls with args	2–4/2–4	0.648	2/4	–	30.2%
bags of n-grams of action	2–4/2–4	0.111	3/4	–	–
bags of n-grams of action with args	2–4/2–4	0.529	2/3	–	22.0%
bags of tuples of syscalls	2-4/2-4	–	–	–	–
bags of tuples of syscalls with args	2-4/2-4	0.497	2/2	–	33.8%
bags of tuples of action	2-4/2-4	–	–	–	–
bags of tuples of action with args	2-4/2-4	0.990	2/2	0.42%	–
tuples of n-grams of syscalls	2-4/2-4	0.509	2/2	–	2.9%
tuples of n-grams of syscalls with args	2-4/2-4	0.624	2/3	–	26.5%
tuples of n-grams of action	2-4/2-4	0.142	3/4	–	0.1%
tuples of n-grams of action with args	2-4/2-4	0.536	2/2	–	24.9%
tuples of bags of syscalls	2-4/2-4	–	–	–	–
tuples of bags of syscalls with arguments	2-4/2-4	0.480	2/2	–	32.4%
tuples of bags of actions	2-4/2-4	–	–	–	–
tuples of bags of actions with arguments	2-4/2-4	0.873	2/2	–	–

99% of unknown malware samples with 0.4% false positives (and a variance of 0.00016), while the second achieved a 90% detection with zero false alarms.

An interesting observation is that the three indicators (max ROC area, false positives at 90% detection rate, and detection at 1% false-positive rate) do not always provide consistent results. In other words, there are models that have a good V_{90}, but do not perform equally well on the V_1 scale, and vice versa. Therefore, the best model also depends on the user's choice of the optimization goal (e.g., detection rate, false-positive rate, or a combination of the two). On the other hand, variance values keep being consistent, and are in average 0.01 on the false positive rate at 90 and 95% detection rate and 0.00001 on the detection rate at 1% false positive rate. The highest variance value we encountered all over the experiments was of 0.033, on a model whose performances are not of high interest (20-grams).

5.3 Impact of Pruning Techniques

Before proceeding in the analysis of the different parameters, it is important to verify that the pruning techniques we adopt to generate the models of bags and tuples are not affecting the validity of the results.

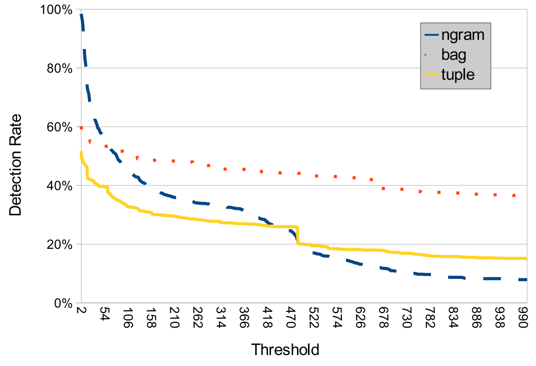
As mentioned in Section 4, in order to contain the number of signatures to a reasonable volume, we discarded the signatures that were not detecting at least five malware samples (i.e., not general enough) not already covered by 20,000 other signatures (i.e., too redundant). As a consequence, the signature set extracted with this greedy approach depends on the order in which the samples in the training set are analyzed. Changing the order is equivalent to changing the starting point of the algorithm, and thus the greedy approach would converge to a different solution. To test the effect of the initial condition on our results, we chose a class of models (tuples of system calls with parameters, of various cardinality) and we repeated three times the signature extraction phase with three different orders of the training samples: normal, reversed, and random.

What we observed is that changing the initial conditions sensibly affects the total number of signatures that we extracted. However, the fluctuation in the quality indicators were quite small (e.g., ± 3% for V_{max}) suggesting that the properties of the models were affected only marginally by the different pruning. Even more importantly for our results, the trends between different models were not affected at all. For example, the effect on the graphs when moving from a 4-tuple to a 10-tuple did not depend on the starting point of the pruning algorithm.

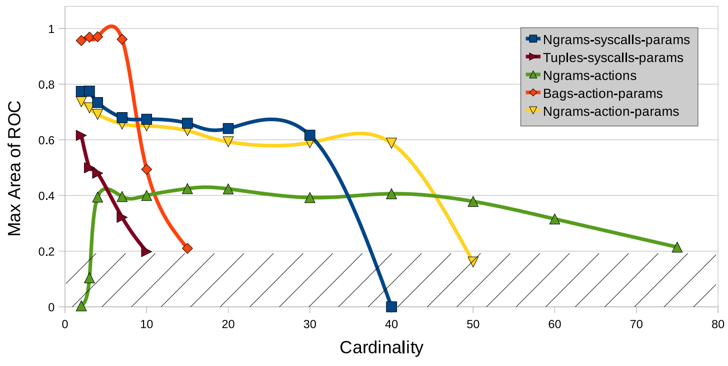
This gives us confidence that even though the absolute values we present may be slightly inaccurate due to the unavoidable pruning approach, the global trends and the messages we discuss in the rest of the section are valid independent of the approximate algorithm used to extract the signatures.

5.4 Impact of Model Threshold

One question we tried to answer with our experiments is whether there exists an optimal range of threshold values that improve the accuracy for all models. As we already described in Section 4, both the detection and the false positive



(a) Matching threshold for different model structures



(b) Impact of Cardinality on the V_{max} of various models (interpolated)

Figure 3: Matching thresholds and impact of cardinality for various models.

rates monotonically decrease when the matching threshold is increased. The first thing we observed for the experiments is that the drop is faster for the models based on a semantically rich set of atoms (e.g., the syscalls with parameters) than on those that are based on simpler atom sets.

Another important observation of our experiments is that the effect the threshold has on a model's accuracy depends on the structure of the model. Figure 3(a) shows three typical detection rate curves obtained for bags, sequences, and tuples of length 4. The graph shows that n-gram models are much more sensitive to the threshold than bag models. In fact, while the detection rate for all n-gram models quickly drops when increasing the threshold, for the bags the accuracy charts exhibit "plateaus" that decrease more slowly.

To summarize, the models that generate a large number of *less constrained* signatures, such as bags, are less sensitive to the threshold than the models that generate a small number of *more specific* signatures. Indeed, we found some models (e.g., bags of actions with arguments), for which increasing the threshold between 1 and 1,000 produces a drop of less than 2.5% on the detection rate.

5.5 Impact of Signature Cardinality

In our experiments, for each signature type, we repeated the test with increasing cardinality values (in the set [1, 2, 3, 4, 7, 10, 15, 20, 30, 40, 50, 60, 75]), until we noticed that the detection rate was constantly dropping and the maximum area of the ROC curve decreased below 0.2.

Figure 3(b) shows how the cardinality affects the signature effectiveness in distinguishing benign from malicious behaviors. Because of space constraints, the chart reports only few models; however, a similar behavior was also observed for the other structures as well. For low values of cardinality, in the range of 2 to 10, adding more atoms to the model can improve the results. However, values greater than 10 do not seem to provide any further benefit. In this case, increasing the cardinality only had the effect of producing signatures that overfit the training malware dataset, thus reducing the detection rate.

The accuracy of certain models (like bags and tuples) drops fast when increasing the cardinality, while it decreases slowly for n-grams and models based on actions. Therefore, while bags and tuples reached our $V_{max} = 0.2$ threshold at a size of 10 or 15, for n-grams we had to run experiments up to the 75-grams of actions. The fact that increasing the cardinality does not help to improve the accuracy is an interesting and positive result because extracting and matching signatures that contain a large number of elements is extremely time consuming.

The relation between the cardinality of models and the number of extracted signatures is less clear. For models based on n-grams, the signature number keeps growing linearly with the size of the models. However, the number of signatures that actually contribute to the detection (i.e., the ones that actually match at least one of the samples) shows an opposite negative trend (i.e., they slowly decrease). Again, this is the consequence of the fact that the signatures are overfitting the training set. Models based on bags generated a high number of signatures also for small cardinality. In this case, however, the number of matching signatures is also quite large, often an order of magnitude higher than for any other model. This is instead an example of signatures that are too general, and therefore, easier to match.

Finally, we tested nine cardinality combinations for each recursive model (inner cardinality = 2,3,4 and outer cardinality = 2,3,4). For example, we evaluated 3-bags of 4-grams or 2-tuples of 4-bags. Our considerations for simple models are still valid also for the recursive ones – the only difference being that all the curves dropped much faster. For example, the 3-tuples of n-bags starts with a V_{max} of over 0.873 with $n = 2$, decreases to 0.199 with $n = 3$, and dropped to 0 for $n = 4$. This results from the fact that, structure aside, a 3-bag of 4-tuples contains 12 atoms and therefore is roughly equivalent (from a cardinality point of view) to a basic structure of that size.

5.6 Impact of Atoms and Signature Structure

The impact of the atom abstraction strictly depends on the structure used to combine the atoms together. In fact, this is the only dimension without a clear winner, with each combination having its own advantages and disadvantages.

For example, n-grams are the only structure that can produce some results also with models based on actions without arguments. However, our experiments show that models based on atoms without arguments contain signatures that are too generic. Therefore, also in the few cases in which they were able to reach the 90% detection level, the number of false alarms was, in the best scenario, already over 30%.

N-grams are also the best structure for low-level atoms (e.g, system calls), while bags and tuples provides the best results when combined with high-level atoms (e.g., actions). At a closer inspection, for models based on *system calls*, n-grams perform better than bags which in turn perform better than tuples. For models based on *actions*, the accuracy impact is reversed, with tuples performing better than bags, which perform better than n-grams.

Finally, the experiments on recursive structures confirmed the fact that tuples and bags tend to provide better results than sequences, especially when combined with actions with arguments. In fact, the "2-bags of 2-tuples of actions with arguments" was the best models among all our experiments (with respect to the max area of the ROC curve) achieving a $V_{max} = 0.99$ and a $V_{90} = 0.4\%$.

5.7 A Quick Look Inside the Models

In order to better understand why some models perform better than others, we took a closer look at some of the models we generated in our experiments. In particular, we extracted for each model the top 20 signatures contributing to the detection rate, and the top 20 signatures that were responsible for the false positives.

The first observation is that the signatures based on bags or tuples tend to be more homogeneous, picking only the combination of atoms that may appear less frequently in normal programs. On the other side, n-grams are forced to combine atoms that appear in a strict order, and therefore often contain more common calls about registry and file system operations. The advantage of tuples over bags is the fact that they can contain repetitions. For example, one of the top signatures of the 2-tuple of n-grams with arguments contains a repetition of the same `NtReleaseMutant` system call. The same call never appears in the top 20 for bags structures. In fact, the presence of only one of these calls was one of the main cause of false alarms.

Looking at the type of system calls included in the top signatures does not provide any useful insight. For example, `NtQueryValueKey` is an important item in the detection of models based on bags, but it is a main cause of false alarms for n-grams models. On the contrary, the picture is more clear when looking at models based on actions. In this family, one clear trend is the fact that the detection rate of good models is always dominated by the `LoadLibrary` atom. This high-level action summarizes many operations in the system-call world, and was therefore more difficult to express in models based on them. A second result is that the signatures responsible for false positives are very often based on actions related to registry operations.

To conclude, we also looked at the presence of particularly sensitive parameter values. For example, we checked ten registry keys associated to autostart locations, reported by Bayer et al. [2] as common in malicious samples. Most of the keys never appeared in the top-20 signature sets and few of them only appear as causes for false positives. However, there was one interesting exception, in that the registry key `SOFTWARE/Microsoft/Active Setup/Installed Components/` was present in the top-20 signatures of 7 different n-grams models, but never appeared as a top cause of false alarms.

6. THE IMPORTANCE OF TESTING

Analyzing models beyond the ones built on signatures with simple structure requires a concerted effort to address the explosion in the number of candidate models. For example, if the number of tuple models over atoms is factorial in the size of the execution trace, then computing all models of bags of tuples is not feasible. Therefore, it may be tempting to follow some a priori rules, maybe based on intuitions about the models and their accuracies. Unfortunately, our experimental results indicate that, in general, no valid rules of this form exist, and that pruning rules should not be derived analytically without a solid experimental testing support.

Fallacy: Increasing the specificity of a component in a model improves the model's accuracy.

When the cardinality of a signature is increased to add more atoms to it, the signature becomes more specific and, therefore, less likely to match on both the goodware and the malware datasets. The result is that, *intuitively*, a model based on 3-grams should generate less false positives than a model based on 2-grams.

However, trying to extend the property of a signature to the property of the models based on that signature is a very common and dangerous pitfall. In fact, since altering the behavioral model alters the number of signatures that are generated, the overall results could often go against the common sense (e.g., making the signatures more specific can increase the number of matches of the entire model).

For example, we can consider the very simple scenario illustrated in Figure 4. The malware set contains only one sample that executes five atoms ($a_1 ... a_5$), and the goodware dataset contains only one process that executes seven atoms (again $a_1 ... a_5$ but in a different order). The bottom part of the figure shows some of the models that can be extracted from the two datasets (the signatures in each model are the ones that match the malware samples but never appear in the goodware one). The 3-grams model contains three signatures, the first of which, $\langle a_1, a_2, a_3 \rangle$, is not covered by any signature in the 2-grams model. Therefore, any program executing those three actions in a row would trigger a false alarm in that model, but not in the one with a lower cardinality (i.e., by increasing the cardinality we possibly increase the detection and false positive rates). Therefore, the actual effect of changing the cardinality of a model can only be measured experimentally, as we did earlier in this section.

Fallacy: Moving to a more selective model structure improves the model's accuracy.

Different structures are comparable in terms of their accuracy only when considering one signature at a time. For instance, sequences are more specific than tuples that, in turn, are more specific then bags. Hence, one may erroneously think that, keeping all the other parameters constant, moving from a sequence to a bag would improve the detection rate but also the false positives.

However, a quick look at the example in Figure 4 is enough to understand that this relationship between signatures does not hold at the model level. For example, the bag models are all empty, the 2-tuple contains only one signature, and the 2-grams contains two. The situation is not much better

Malware:	$(a_1, a_2, a_3, a_4, a_5)$
Goodware:	$(a_3, a_1, a_2, a_5, a_4, a_2, a_3)$
2-grams :	$\langle a_3, a_4\rangle, \langle a_4, a_5\rangle$
3-grams :	$\langle a_1, a_2, a_3\rangle, \langle a_2, a_3, a_4\rangle, \langle a_3, a_4, a_5\rangle$
2-tuple :	$[a_4, a_5]$
3-tuple :	$[a_1, a_3, a_4], [a_1, a_3, a_5], [a_1, a_4, a_5], [a_2, a_3, a_4], [a_2, a_3, a_5], [a_3, a_4, a_5]$
k-bags :	$\varnothing$

Figure 4: Example of Behavioral Models

when we consider recursive structures. In general, there is no clear relation in terms of false positives and detection rate, between a simple model (e.g., a 5-grams) and its aggregated version (e.g., a 3-tuple of 5-grams).

This discussion illustrates that it is *not* possible to generalize results in a closed form. Therefore, an analysis of the model space has to be necessarily informed and driven by a comprehensive experimental evaluation. Otherwise, the use of intuitive yet wrong rules would erroneously eliminate certain models from consideration.

7. CONCLUSION

In this paper, we propose a systematic approach to measure how the choice of behavioral models influences the quality of a malware detector. We achieve our goal through a large set of testing experiments in which we explore the space of many possible behavioral models. Our findings confirm that the accuracy of certain models is very poor, independently of the values of their parameters. In general, the best models are the ones that rely on few, high-level atoms with their arguments. Finally, our experiments show how each parameter impacts the final result and how they should be chosen in order to maximize the accuracy and reduce the false alarms.

8. ACKNOWLEDGEMENTS

The research leading to these results was partially funded from the EU Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257007, by the NSF grant CNS-1116777, and by the i-Code project.

We would like to thank Seeweb srl for lending us part of the machines in their cluster for conducting our experiments.

9. REFERENCES

[1] http://anubis.iseclab.org, 2011.

[2] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *Proc. LEET'09*, 2009.

[3] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *Proc. DIMVA*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143, 2006.

[4] M. Christodorescu and S. Jha. Testing malware detectors. In *Proc. ISSTA'04*, 2004.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. IEEE S&P*, pages 32–46, 2005.

[6] M. Christodorescu, C. Kruegel, and S. Jha. Mining specifications of malicious behavior. In *Proc. ESEC/FSE*, pages 5–14, 2007.

[7] N. Falliere. Stuxnet introduces the first known rootkit for industrial control systems. Published online at http://www.symantec.com/connect/blogs/stuxnet-introduces-first-known-rootkit-scada-devices. Last accessed on February 10, 2011.

[8] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Kruegel, editors, *Proc. DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187, 2005.

[9] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proc. USENIX Security*, 2006.

[10] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. USENIX Security*, pages 351–366, 2009.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proc. RAID*, volume 3858 of *LNCS*, pages 207–226, 2005.

[12] G. Kurtz. Operation "Aurora" hit Google, others. Published online at http://siblog.mcafee.com/cto/operation-%E2%80%9Caurora%E2%80%9D-hit-google-others/. Last accessed on February 10, 2011, 2011.

[13] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proc. 6th IEEE SMC Workshop on Information Assurance*, pages 64–71. United States Military Academy, 2005.

[14] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behavior. In *Proc. RAID*, pages 78–97, 2008.

[15] McAfee Labs. McAfee threats report: Fourth quarter 2010. Publishsed online at http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2010.pdf. Last accessed on February 10, 2011.

[16] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *Proc. ACSAC'07*, 2007.

[17] S. Mukkamala, A. Sung, D. Xu, and P. Chavez. Static analyzer for vicious executables (SAVE). In *Proc. ACSAC*, pages 326–334, 2004.

[18] G. Nebbett. *Windows NT/2000 Native API Reference.* 2000.

[19] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proc. IEEE S&P*, pages 38–49, 2001.

[20] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *Proc. DIMVA*, 2007.

[21] P. Ször. *The Art of Computer Virus Research and Defense.* 2005.

[22] J. Xu, A. H. Sung, P. Chavez, and S. Mukkamala. Polymorphic malicious executable scanner by API sequence analysis. In *Proc. HIS*, pages 378–383, 2004.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. CCS*, pages 116–127, 2007.