NORT: Runtime Anomaly-Based Monitoring of Malicious Behavior for Windows

Narcisa Andreea Milea[1], Siau Cheng Khoo[1], David Lo[2], and Cristian Iuliu Pop[3,⋆]

[1] National University of Singapore
{mileanar,khoosc}@comp.nus.edu.sg
[2] Singapore Management University
davidlo@smu.edu.sg
[3] Microsoft, Redmond USA
cristi.pop@gmail.com

Abstract. Protecting running programs from exploits has been the focus of many host-based intrusion detection systems. To this end various formal methods have been developed that either require manual construction of attack signatures or modelling of normal program behavior to detect exploits. In terms of the ability to discover new attacks before the infection spreads, the former approach has been found to be lacking in flexibility. Consequently, in this paper, we present an anomaly monitoring system, NORT, that verifies on-the-fly whether running programs comply to their expected normal behavior. The *model of normal behavior* is based on a rich set of discriminators such as minimal infrequent and maximal frequent iterative patterns of system calls, and relative entropy between distributions of system calls. Experiments run on malware samples have shown that our approach is able to effectively detect a broad range of attacks with very low overheads.

1 Introduction

Many techniques have been proposed to ensure the safety of computing systems. Security policies on the flow of sensitive information [2] and encryption target only the safety of highly sensitive data while neglecting the presence of malware and infections. Traditional antivirus system target infections by searching for known patterns of malware *statically*, within system files. Host-based intrusion detection systems (IDS), on the other hand, monitor the *dynamic* behavior of a computing system in order to detect infections.

Misuse IDS [11] are similar to traditional antivirus systems. They model *known* intrusions and scan running programs to detect signatures of attacks. While they benefit from a high degree of accuracy their main drawback is the inability to detect novel attacks. Consequently, attackers exploit this weakness by using various obfuscation techniques or developing new attacks. Built as a response, *anomaly-based IDS* learn the normal behavior of programs and protect

⋆ Research conducted by co-author when attached to National University of Singapore.

them by observing the events they generate and comparing them to the *expected* behavior, thus are capable of detecting new attacks. The models of *expected* behavior can be obtained either by static analysis [21, 8, 9] or dynamic analysis [7, 13, 23, 24, 18, 6]. Although conservative static analysis approaches do not exhibit false positives they suffer from generating and using imprecise models due to the need to handle non-determinism, non-standard control flows, function pointers, libraries, etc. Dynamic analysis, on the other hand, leverages specific program's input to yield more accurate models; it however admits false positives.

In this work we propose an *anomaly-based IDS*, called NORT, that models *dynamic* behavior of programs and detects attacks by discovering deviations from the expected behavior. Our motivation lies with the very nature of malware that will usually reach our computers by exploiting vulnerabilities in running programs, getting installed as a start-up service by using legitimate services and hiding itself by modifying legitimate programs.

We build upon the work of Forrest et al [7] that was the first to propose a simple yet effective model, based on contiguous sequences of system calls, to describe the behavior of programs. We add to their success and strive to attain better performance by considering arguments, return values and probability distributions of system calls (in addition to temporal information) in our model. More importantly, we capture *both frequent* and *infrequent system call patterns* and relative entropy between distributions of calls to distinguish between acceptable and unacceptable behavior. Compared to other techniques based on data mining [24, 13] one of our contributions is the richness of our feature set: NORT is the first work that uses *iterative patterns* (of system calls) to model normal behavior. Iterative patterns permit gaps between adjacent calls found in the patterns, allowing for faster convergence and both effective and efficient detection of variants of malware. They also succinctly capture repetitive call sequences, resulting in patterns of shorter length and far less overhead in pattern manipulations. We are also the first to employ relative entropy to detect anomalies in a host machine (previously, this has mostly been used in network IDS). By adding this extra layer of security we raise the likelihood of an intrusion being detected.

NORT addresses current security issues, including the zero-day attacks, and the emergence of more a advanced malware phenomenon also known as Malware 2.0. These new security situations entail the development of adaptive methods, such as NORT, that can detect attacks and intrusions without prior knowledge about the malware itself. The contributions of our work are:

1. A new mining algorithm of frequent and infrequent patterns and a practical application to runtime verification and malware detection.
2. An effective model to describe the dynamic behavior of programs, incorporating not only the temporal ordering of input events but also data-flow information. This differs from past work on pattern based specification mining[15].
3. A prototype system, implemented and tested on Windows, to verify that programs comply to their expected behavior. Most IDS so far focused on Unix systems. However recent attacks such as the one on Google [1] showed the need for an IDS for a commodity operating system such as Windows.

The experiments aimed at evaluating our prototype system have shown a good balance between the three main concerns of dynamically built program models: accuracy, training convergence, and efficiency. Accuracy makes the model useful while efficiency and the rate of convergence make it usable, especially on-the-fly. Our results showed fast training convergence for both simple applications such as the Windows printing service and complicated applications such as Internet Explorer and Adobe Reader. In terms of accuracy, NORT showed the ability to detect a broad range of attacks with runtime overhead of less than 10%.

2 Overall Picture

NORT is designed as a system that offers individual computers one more layer of security, besides the ones already used: firewall, network IDS and antivirus.

It relies on the fact that software is used in a consistent manner and it detects malicious changes by a two phase-system: first it learns the normal behavior of the system and then it monitors the dynamic system to detect deviations from normal. During the *learning* phase a stochastic vector capturing the distribution of system calls for each *program* is computed, frequent iterative call patterns discovered across programs are mined and minimal infrequent call patterns exhibited within each program are identified and stored in a *normal behavior database* (NBDB). When the *detection* mode is activated, NORT computes the relative entropy between the stochastic vectors of the *running processes* and the corresponding *learned programs* in the NBDB, mines maximal frequent and minimal infrequent iterative patterns and compares them against those stored in NBDB.

Similar to usual dynamic machine learning approaches, a major challenge that we face is with the ***incompleteness*** of the training data. In the ideal case the normal database would contain all variations in normal behavior and we could regard a single mismatch found to be significant. Unfortunately, in real environments, it is practically impossible to collect all normal variations. Our solution to this problem is to attach, to each process, a *trust barometer* that increases the trust level when normal behavior is observed and decreases when anomalies are detected. Different types of anomalies have different weights associated to them. The weights of high entropy and new frequent patterns are heavier since these are less likely to occur in traces while the weight of new infrequent patterns is lighter such that several anomalies must occur before an alert is raised. The weight associated with normal behavior is much smaller than those associated with anomalies and has the effect of ignoring isolated anomalies.

The **architecture** of NORT is modular, being comprised of a kernel-driver, an engine and the user interface, as seen in Figure 1(a). The *kernel instrumentation* module acts as a sensor, recording system calls with their parameters and passing them to the engine. Because these have to be done at real-time, kernel-level buffers are used. The *engine* consists of several modules and is the core of the system. The first module *preprocesses* the data and passes the results through the learner/detector modules: *entropy* and *data miner*. The learner/detector modules store or query information from the *storage* module. The graphical user interface allows the user to choose the programs to monitor and to view the reports of specifications learned or detected and alerts generated.

3 Data Preprocessing

For runtime systems that deal with an infinite sequence of input events, it is important to have efficient data collection and preprocessing techniques. This section is thus dedicated to describing the *kernel instrumentation module* that handles the interception of the input events (system calls) and the *data preprocessing module* where we structure the stream of system calls by considering call arguments and employing techniques such as aggregation.

3.1 System-Calls and Kernel Instrumentation

System calls represent the basic interaction unit between programs and the OS kernel. We assume that *any harmful attack to a system will require the compromised applications to interact with the OS*. Thus, we focus on inspecting system calls, their arguments and return values to discriminate between normal and abnormal dynamic program behaviors.

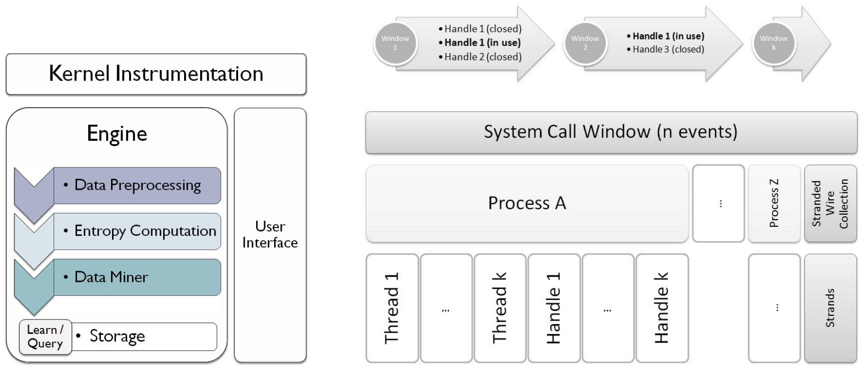
Several approaches have been proposed for intercepting system calls. User-level mechanisms [18, 6] are deemed unsuitable, as they usually incur run-time overheads in the range of 100% and 250% due to the additional task switching operation required at each interception. Techniques that intercept system calls *within* the kernel, through kernel modifications, incur much lower overheads. We therefore adopt the latter approach and use techniques from BindView's strace open-source application to install a kernel driver. We also provide users the option to monitor either all or part of the system calls and their parameters.

3.2 Handling Complex Behavior and Overcoming Obfuscation

Signature based antivirus programs easily become ineffective when viruses employ obfuscation techniques [4]. Several types of obfuscation techniques are described in [4] classifying viruses as either *polymorphic* or *metamorphic*. A polymorphic virus tries to avoid detection by encrypting itself and applying transformations to its decryption routine such as inserting instructions without effect (*nop*, *dead-code*), changing the order of instructions and inserting jump instructions to preserve the effect of the code (*code transposition*) or making use of other registers. Metamorphic viruses change their code to an equivalent one by employing more complex techniques such as code transposition, equivalent instruction sequence substitution, and code insertion to the entire host binary.

By performing data mining on system calls sequences, it is possible to eliminate threats from many of these obfuscation techniques. However, mining system call patterns naively may not be effective, as it may attempt to distinguish call patterns which differ by the ordering of calls made on different resources. For instance, when a program opens two files and mixes reads and writes from these two files, two call patterns describing this file operation behavior may deem distinct as they capture different orderings of file operations.

NORT performs clever data mining on system calls. It first overcomes the call ordering problem mentioned earlier by using system call's parameters. NORT groups together system calls that refer to the same resource, thus ignoring the order of operations that apply to different resources. Next, contiguous strings of the same repeating system calls are aggregated into one system call; cf. [5]. These techniques also help in learning more complex behaviors and speeding up the convergence rate of normal behavior as we will explain in the section on Experiments.



(a) Architecture (b) A Sliding Window and Trace Partitioning

Fig. 1. NORT

3.3 Trace Partitioning

As NORT is required to analyze data in the form of possibly infinite sequences of calls, we apply a *sliding window model* to split the data and mine and compute relative entropy for windows of events. A definition of this model would be: each element arrives at time t and expires at time $t + w$, where w is the window size.

On top of splitting the stream of system calls into windows we further structure it by process ids and then in thinner strands by handle and thread ids (Figure 1(b)). We also apply an aggregation technique which involves adding a system call to a strand only if it differs from the previous one. A thread strand will contain all system calls generated by the thread execution and that are not related to a handle. A handle strand will contain system calls that act upon an object such as a file, socket, button (handles are some of the most important data objects in Win32). Because handles may be reused in the process context, they can be either *in use* (opened) or *old* (closed). Handles that are in use get special treatment when a window expires. Specifically, in order to avoid losing information, handles in use are kept beyond window expiration until they are closed or they have reached a certain age (in terms of number of windows).

During preprocessing we also check the return values of system calls. If a system call has executed successfully we will add its id to the preprocessing database. Otherwise we will add an anomaly score (greater than the highest id ever assigned to a system call) to its id and add this new value to the preprocessing database. By using this additional information the training convergence of normal behavior is not adversely affected. Also, we are able to detect more attacks such as the PDFKA attack on Adobe Reader.

4 Analysis Engine

To detect anomalous behavior we make use of both statistical analysis and a specification mining approach that extends the algorithm proposed by Lo et al [15]. In this section we describe these two layers of detection and the method for storing the specifications that will be used for real time detection of malware.

4.1 Statistical Analysis

As the first layer of defence we build upon the approach expounded in [16] to compute relative informational entropy [14] which captures the distance between the regularity of two datasets. Our motivation lies with the fact that, similar to Internet traffic, *system call patterns* have both randomness and structure, and malware can alter both. Specifically, when most malware enter their infection and multiplication phase, they start accessing files, creating network connections, sending emails thus changing the randomness and patterns of system calls. To detect such changes we use relative entropy which can be defined as follows:

Definition 1. *The relative entropy between two probability distributions P and Q that are defined over the same class C_x is:*

$$H_{rel}(P \mid Q) = \sum_{x \in C_X} P(x) \log \frac{P(x)}{Q(x)}.$$

In our interpretation of relative entropy, x represents a system call, the class C_X refers to the set of all system calls under consideration while the two probability distributions P(x) and Q(x) refer respectively to the *learned* and *current* distributions of system calls generated by a process. Specifically, Q(x) is implemented as a stochastic vector that captures the degree of randomness of a process in the *currently* processed window and P(x) is the corresponding stochastic vector computed and stored in the NBDB (for each program) during the *learning* phase by aggregating the results from *multiple* windows. These two vectors are used to compute the *relative entropy* between the current distribution for each process and window and the learned distribution of the corresponding program (found in NBDB). These relative entropies obtained during the learning phase are used to determine the maximum relative entropy exhibited during learning for each program. The relative entropies obtained during the *detecting* phase, on the other hand, are compared against a threshold and the trust levels of the appropriate programs are decremented or incremented according to the result. The threshold chosen is relative to the maximum relative entropy observed in the learning phase. It is thus unique for each program as a global absolute value would not be appropriate due to differences among programs.

As entropy only measures the randomness of system calls, not all malware could be detected via entropy measures. To further enhance the malware detection capability, we use a novel data mining algorithm that looks for both frequent and infrequent patterns in the call stream, as described in the next subsection.

4.2 Specification/Pattern Mining

To capture relevant patterns of software behavior we adapt two existing mining algorithms – *Efficient Mining of Iterative Patterns for Software Specification Discovery* [15] and *Towards Rare Itemset Mining* [20] – and obtain *FEELER*: *Frequent and infrEquent itErative sequentiaL pattErn mineR*. In this subsection we describe formally the notion of iterative patterns and how they are used by FEELER to detect malware.

Basic Definitions. Let I be a set of distinct events which are the system calls under consideration. Let a sequence S be an ordered list of events. We write S as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where each e_i is an event from I. The input to the mining algorithm is a set of sequences also referred to as the sequence database (SeqDB).

A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ where $e_1 = f_{i_1}, \ldots, e_n = f_{i_n}$. We denote this subsequence relationship by $P_1 \subseteq P_2$.

Concatenation of two patterns $P_1(\langle e_1, \ldots, e_n\rangle)$ and $P_2(\langle f_1, \ldots, f_m\rangle)$ is defined as follows: $< e_1, \ldots, e_n > +\!\!+ < f_1, \ldots, f_m >=< e_1, \ldots, e_n, f_1, \ldots, f_m >$.

Semantics of Iterative Patterns. For the rest of this section, we use *iterative pattern* and *pattern* interchangeably. An *iterative pattern* is a pattern the instances of which conform to a specific requirement, as defined below:

Definition 2. *(Iterative Pattern Instance) Given an iterative pattern* $P = \langle e_1, e_2, \ldots, e_n\rangle$*, a substring* $SB = \langle sb_1, sb_2, \ldots, sb_n\rangle$ *of a sequence* S *in the sequence database is an instance of* P *if* SB *can be described by the Quantified Regular Expression:*

$$e_1; [-e_1, \ldots, e_n]; e_2; [-e_1, \ldots, e_n]; \ldots; e_n$$

A Quantified regular expression is very similar to a standard regular expression with ; *as the concatenation operator and* [-] *as the exclusion operator.*

Definition 3. *(Support) The support of a pattern P (denoted as sup(P)) wrt. to a sequence database SeqDB is the number of its instances in SeqDB.*

Definition 4. *(Frequent and Infrequent Patterns) A pattern P is considered* frequent *in SeqDB when its support, sup(P) is greater or equal to a certain threshold (min_sup). Otherwise if* $sup(P) < min\_sup$*, P is* infrequent *or* rare.

The following theorem, the proof of which is omitted, provides a valuable means to prune the search space during mining, rendering the mining process efficient.

Theorem 1. *(Anti-monotonicity Property) If a pattern Q is infrequent and P* $= Q +\!\!+ evs$ *(where evs is a series of events), then P is also infrequent.*

As there may be too many frequent and infrequent patterns, we mine for two compact sets of patterns: maximal frequent and minimal infrequent.

Definition 5. *(Maximal Frequent Patterns) An iterative pattern P is considered maximal frequent in a sequence database SeqDB if P is frequent and there exists no super-sequence Q such that $P \subseteq Q$ and Q is frequent in SeqDB.*

Definition 6. *(Minimal Infrequent Patterns) An iterative pattern P is considered minimal infrequent (minimal rare) in a sequence database SeqDB if P is rare and there exists no sub-sequence R such that $R \subseteq P$ and R is rare in SeqDB.*

Generation of Iterative Patterns. Our algorithm for mining, FEELER, adopts a depth-first pattern growth and prune strategy to obtain maximal frequent and minimal infrequent *iterative patterns*. Its input comes from the preprocessing module in the form of all strands (handle and thread) of system calls corresponding to one process. These strands constitute the sequences in the *SeqDB*. The output of FEELER, the iterative patterns obtained for each running process in a window of calls, are stored in the NBDB during the learning phase and checked against the corresponding ones in the NBDB in the detection phase.

```
Procedure MinePat
Inputs :   $min\_sup$ : Minimum Support Threshold
Outputs : $freqDB$ : Max. Frequent Patterns, $infreqDB$ : Min. Infreq. Patterns
1 : Let $FreqEv$ = All single events $e$ where $sup(e) \geq min\_sup$
2 : Let $infreqDB$ = All single events $e$ where $0 < sup(e) < min\_sup$
3 : Let $freqDB = \{\}$
4 : For each $f\_ev$ in $FreqEv$ do
5 :     Call GrowPat $(f\_ev, min\_sup, FreqEv, freqDB, infreqDB)$

Procedure GrowPat
Inputs : $Pat$ : A frequent pattern
         $min\_sup$ : Minimum Support Threshold
         $EV$ : Frequent Events
         $freqDB$ : Max. Frequent Patterns, $infreqDB$ : Min. Infrequent Patterns
 6 : Let $NxtFreq = \{Pat ++e \mid e \in EV \wedge (sup(Pat ++e) \geq min\_sup)\}$
 7 : Let $NxtInfreq = \{Pat ++e \mid e \in EV \wedge (0 < sup(Pat ++e) < min\_sup)\}$
 8 : For each $iPat \in NxtInfreq$
 9 :     If $(\nexists R.\ (R \in infreqDB \wedge R \subseteq iPat))$ then
10 :         $infreqDB = infreqDB \setminus \{Q \mid Q \in infreqDB \wedge (iPat \subseteq Q)\}$
11 :         $infreqDB = infreqDB \cup \{iPat\}$
12 : If $|NxtFreq| = 0$ then
13 :     If $(\nexists Q.\ (Q \in freqDB \wedge Pat \subseteq Q))$ then
14 :         $freqDB = freqDB \setminus \{R \mid R \in freqDB \wedge (R \subseteq Pat)\}$
15 :         $freqDB = freqDB \cup \{Pat\}$
16 : Else For each $fPat$ in $NxtFreq$
17 :     Call GrowPat$(fPat, min\_sup, EV, freqDB, infreqDB)$
```

Fig. 2. FEELER Mining Algorithm

The main procedure of FEELER, *MinePat*, shown in Figure 2, will first find frequent patterns of length one (Line 1) and then call *GrowPat* which recursively grows each pattern (Line 5). The length-1 patterns that are infrequent ($support < min\_sup$) and minimal are added to *infreqDB* (Line 2).

Procedure *GrowPat*, shown at the bottom of Figure 2, receives as inputs a frequent pattern *(Pat)*, the support threshold, the set of frequent events and the sets of maximal frequent *(freqDB)* and minimal infrequent *(infreqDB)* iterative patterns. The recursive algorithm will grow the current pattern Pat by a single event and collect the resultant frequent and infrequent patterns (Lines 6-7). For each infrequent pattern $iPat$, GrowPat will check if any of its subsequences is in *infreqDB* (Line 9) and will add $iPat$ to *infreqDB* if no pattern is found. The patterns in *infreqDB* that are not minimal are also removed (Line 10). For each frequent pattern $fPat$ it will try to grow further by calling $GrowPat$ recursively (Line 17). If however the growth of Pat resulted in non-frequent patterns, Pat is added to *freqDB* if none of its super sequences is found in *freqDB*(Lines 12-15).

4.3 Storage

The Storage module interacts with the entropy and miner modules and manages the extracted specifications. The unique feature that enables this module to efficiently respond to queries is the use of bloom filters to store patterns [3](one bloom filter for the frequent patterns from *all* the running processes and one *per* process for the infrequent patterns). These data structures generate and store a unique binary hash of a pattern and allow us to query for patterns without having to enumerate them. However, depending on the bloom filter size and hashing, the queries might have false positives. We have determined empirically the size of all the bloom filters to be 4MB in order to reduce the false positives.

5 Experiments

Dynamically built program models for runtime intrusion detection can be evaluated on three criteria: accuracy, training convergence, and efficiency. Greater accuracy makes the model useful while efficiency and fast convergence make the model usable. In order to evaluate our model we gathered different types of real world exploits and legitimate applications and confronted them against our prototype. All the experiments were performed on a Quad Core i7 running Windows XP SP 2 with 2GB of RAM inside VMWare Player on a Windows 7 host.

We started the experiments by first constructing models of normal behavior for the internal components of the operating system (winlogon, explorer, spoolsv, services), the pre-installed programs (Internet Explorer, notepad), and several legitimate applications such as Adobe PDF Reader. The obtained models showed fast training convergence rates for both simple and complex applications. Second, we ran a series of cross-validation tests. These tests consisted of learning the models of normal behavior and then feeding Nort in detection mode with new data from a clean installation, data that has never been learned before. No false positives were exhibited at the end of the tests. We next experimented with a broad range of malware samples and observed significant changes caused by the exploits in legitimate applications (as exhibited by the introduction of new frequent and infrequent patterns and a significant increase in entropy) that resulted in all attacks being detected. Lastly, we evaluated the efficiency of NORT by computing the runtime overhead of the system.

During the course of the experiments we used a window size of 10,000, a minimum support of 20 for mining and a threshold of 150% for relative entropy. From the list of 284 system calls of Windows XP we monitored a subset of 274.

5.1 Training Convergence

All anomaly detection techniques factor the time required to train and the convergence of the model in their evaluation. The rate of convergence is of particular interest as it governs the training time needed to attain a given level of false positives. The faster the convergence rate the smaller the training time needs to be.

We ran experiments with Internet Explorer, Adobe PDF Reader, spoolsv, and the internals of the operating system (services such as explorer, svchost, etc.). Internet Explorer and Adobe Reader were chosen as we wanted to show how learning converges for complex applications with interfaces, and where user behavior is perceived to yield slow convergence. The spoolsv printing service, for which all executions differ, was chosen to demonstrate that fast convergence can be attained by aggregating multiple reads and multiple writes.

The rate of convergence was measured in terms of the number of *unique* frequent and infrequent patterns (as functions of the number of system calls) required to learn applications. As shown in Figure 3, despite the initial surge, the

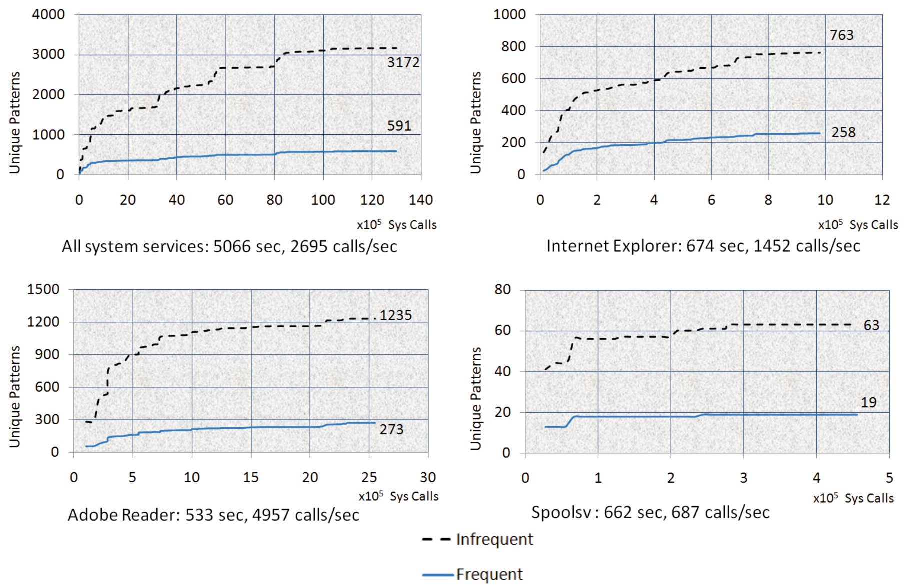


Fig. 3. Training convergence

increase in new patterns tapers off after reasonable amount of training time. We also ran alternative experiments that did not split the system calls by handles; there we observed a larger number of learned patterns (due to the interleaving of calls on resources such as keys, threads, etc.) and much longer times are needed for learning to converge.

5.2 Performance Study

An improvement in security comes at a cost: the performance degradation caused on a running system. As NORT has to continuously monitor legitimate applications without disrupting overall usability of the system, performance is critical. To evaluate the efficiency of NORT we measured the impact of on-the-fly monitoring on the runtime of 7-zip (a well known compression application) and on the startup time of three interactive applications (Figure 4). We also observed the system during viral tests, and found neither noticeable slow-down nor loss in usability.

As the runtime overhead caused by NORT depends on the type and rate of system calls it processes we ran experiments that would show a range of system usage scenarios. In the first 7-zip test, a simple compression benchmark was run mimicking the case in which an application is performing a CPU-bound computation. In the second test, a mixed workload scenario was simulated. Here, 7-zip was used to compress and archive a folder that contained 733 MB of data (404 files in 74 subfolders). The third test depicts an IO-bound workload scenario. Here, 7-zip was used to archive the same folder without performing compression. These results, as summarized in the left table in Figure 4, show that running NORT causes very low overhead (less than 10%) for all applications.

The other set of experiments we ran were aimed at measuring the impact of NORT on the startup time of Microsoft Office and Adobe Reader. To this end we used a program to launch the tested applications and monitor the initialization status through the WaitforInputIdle() API. The results – depicted at the right of Figure 4 – showed that the monitoring activities only incur a slight overhead on the tested applications although a higher rate of system calls was generated.

	Baseline	Monitored
7-zip Benchmark	5:29s	5:31s(0.6%)
7-zip Compress	2:43s	2:45s(1.22%)
7-zip Archive	18s	19s(5.55%)

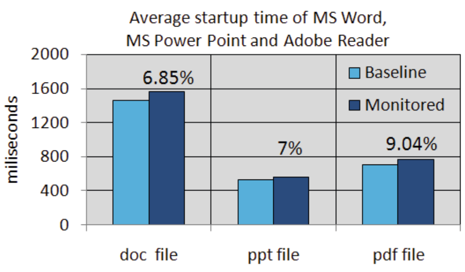


Fig. 4. NORT monitoring runtime overhead

We also ran tests to find the rate of system calls NORT can intercept and process. During the course of these experiments we found that when the OS is not intensively used (cmd, notepad, calc, etc.) the average number of system calls/second is 1,600 whereas the peak is 26,000. When the system is used intensively (a Trend Micro Office scan or running MS Visual Studio while browsing the Web) the average number of calls/second is 12,476 and peaks at 65,000. These findings are shown in Figure 5. In addition, the last two bars in this figure also show NORT's processing capability in two running modes. In the online mode NORT was able to handle a high rate of calls while in the offline mode (all calls are written in a MySql file) NORT is able to handle a smaller rate.

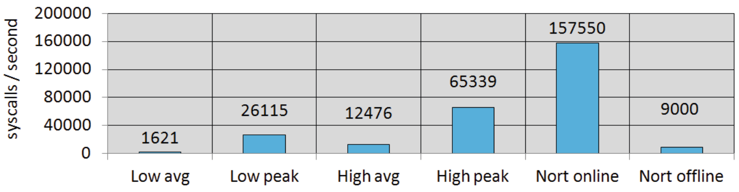


Fig. 5. NORT processing capability

During the course of all experiments NORT handled high rates of system calls with a small runtime impact. We attribute these results to the efficient interception of system calls, the small size of the window on which the algorithms are run, and the aggregation of calls.

5.3 Accuracy

In this subsection, we show that NORT is able to accurately detect attacks by real-world malware. Each viral experiment we performed involved, as the first step, the analysis of the malware to find information about the applications compromised and the nature of the changes. We then learned the behavior of legitimate applications and ran the malware and NORT (in detection mode) to capture the changes on legitimate applications. To avoid mixing malware, the virtual machine was brought to its initial installation before every experiment.

W32/Virut.n is a polymorphic virus that infects PE and HTML files and downloads other malware. In order to modify critical files the virus first disables the System File Protection (SFP) by injecting code into winlogon. The injected code modifies sfc_os.dll in memory (which allows it to infect files protected by SFP) and downloads malicious files such as worm_bobax.f and worm_bobax.bd. The observable effects of the intrusion are an increase in entropy, new frequent operations on files and the registry, new infrequent patterns containing unseen system calls related to network activity (as winlogon downloads malicious files), to the creation of processes and threads (as winlogon creates processes from the files downloaded), to the deletion of keys or values from the registry, etc.

PDFKA takes the form of an innocent PDF document and is accounted for 42.97% of all exploits detected by Kaspersky in the first quarter of 2010 [10]. In our experiments, running Win32.Pdfka.bo resulted in a large number of unseen frequent and infrequent patterns in AcroRd32Info, AcroRd32 and

AdobeARM. AcroRd32Info started exhibiting infrequent network activity and infrequent activity on processes, threads and virtual memory. The new behavior of AdobeARM was mostly related to activity on processes (virtual memory accesses and threads creation) as AdobeARM will open all running processes. The new patterns generated by AcroRd32 were detected from the anomalous return values reflecting the corrupted nature of the PDF file being opened.

Project Aurora surfaced in December 2009 when security experts at Google identified a highly sophisticated attack targeted on their corporate infrastructure and another 20 large companies such as Adobe, Yahoo, etc. The 0-Day widespread attack exploits a vulnerability in the way Internet Explorer handles a deleted object which results in IE referencing a memory location where the attacker dropped malicious code. In order to test this exploit we connected two virtual machines (the attacker and the victim) and used reverse tcp as payload. Employing the attack caused IE 6, running on the victim, to connect to the attacker and execute commands (getting the user id and a screenshot). The new patterns detected by NORT include system calls that create named pipes (the new code in IE initiates communication with the attacker) and system calls on threads, processes and virtual memory (memory is allocated for the payload).

Win32.Hydraq is a family of backdoor Trojans that was first used by the Aurora Project, as a payload. The carefully crafted attack takes advantage of the svchost process in Windows (a common technique used by malware to persist on a compromised computer) and can be detected by our approach due to a new service and the new patterns generated by it. The attack caused svchost to exhibit network activity directed to 360.homeunix.com and was detected by means of: new frequent patterns of network usage, of memory mapped IO and files; infrequent patterns on the registry that returned anomalously, etc.

Other attacks For a more thorough evaluation we tested several more malware taken from the *Top 10 malware list* [19]. *Z0mbie.MistFall.3*, one of the best metamorphic viruses, which infects other executables and causes many running programs to exhibit abnormal behavior, was detected by NORT. We are also able to detect *NetSky.y* and *Mytob.x* in different processes as these worms overwrite other executables and try to exploit components of the operating system (*services.exe* and *svchost.exe*). In the experiments with *Zhelatin.uq* (*aka Storm*), the newly installed malicious service component was detected as anomalous.

6 Related Work

Many models have been proposed that try to define normal system behavior in such a way that the models are sensitive to dangerous foreign activity.

Models of program behavior obtained by dynamic analysis. Forrest et al [7] were the first to propose the use of fixed length contiguous sequences of system calls (n-grams) to define the expected behavior of programs. Their results showed a fast convergence and good discrimination which made system calls based IDS the most popular approach in detecting novel attacks. The downfall of the simple n-gram model is that due to not allowing gaps between system calls forming sequences, one single misplaced system call will cause multiple mismatches. In our mining approach we mitigate this weakness by allowing for flexible gaps between system calls forming patterns of various sizes. We also consider the frequencies of patterns and split the stream of system calls by threads and handles to address the complexity due to concurrency.

Lee and Stolfo adopt a data mining approach by generalizing fixed length sequences of system calls as a set of concise association rules [13]. They reported a good degree of success in accurately detecting new attacks. We further improve their success by (1) mining on variable length sequences thus allowing for flexible gaps among call events and capturing long term correlations (2) involving call arguments and return values in our mining.

In [24] Wespi et al introduce a technique based on the Teiresias algorithm to create a table of *maximal* variable length patterns. While their model uses variable-length maximal patterns and aggregation, they do not allow gaps in patterns and do not consider using infrequent patterns and relative entropy.

Sekar et al [18] propose profiling normal system behavior via finite state automata with the states corresponding to the values of the innermost program counter located at a static location and the transitions to system calls. They thus are not able to characterize the behavior exhibited by dynamically linked libraries which, as demonstrated in [6], may significantly impair their accuracy.

Feng et al. use both program counters and stack history to capture normal behavior in [6]. This enables the detection of any attack that modifies the return address of a function. The overheads reported in [18, 6] are unfortunately in the range of 100 to 250%.

Pattern-Based Specification Mining. There have been a number of works on mining patterns as specifications of a program [15, 17, 12]. In our approach we leverage the efficiency of iterative patterns [15] and extend it to mine for both minimal infrequent and maximal frequent events. This new algorithm combined with a smart preprocessing of the input events and statistical analysis proved to be an expressive way of modeling behavior and effective in detecting malware.

7 Conclusions and Future Work

As Malware 2.0 threatens to be more adaptive than what we have experienced so far, we believe that data mining and artificial intelligence techniques will play more prominent roles in managing the new security problem.

This paper describes a prototype system (NORT [1] that integrates advanced pattern mining techniques and relative entropy to effectively and efficiently detect malware intrusions. By using these layers of defence our prototype attained a reasonably fast rate of training convergence for all applications and detected all malware intrusions with at most 10% slowdown. Although further investigation is required, we believe that by using frequencies, distributions, and relative entropy of system calls our system should be robust enough to mimicry attacks, an invention of Wagner and Soto [22] tasked at evading IDS detection.

[1] The prototype can be found at `http://www.comp.nus.edu.sg/%7Especmine/nort/`

NORT is not meant to be a substitute of an antivirus, but rather to complement one. The ability of NORT to detect malicious activity as anomalous behavior can prevent spreading of viruses or worms and provide a modern tool for security specialists to determine the installation of rootkits inside systems.

We envisage the use of NORT in an environment with multiple similar host systems. Here, distributed data mining can be used to better detect intrusions or to identify points of malware entry. Patterns discovered from all the machines will be gathered and then compared, taking in account the time-window and frequencies. Notifications can be redirected to a security specialist or a larger cross-institution knowledge base of known patterns.

Acknowledgment. We are thankful to the reviewers for their valuable comments. This work is partially supported by the research grants R-252-000-403-112 and R-252-000-318-422.

References

[1] Operation aurora (2010), `http://en.wikipedia.org/wiki/Operation_Aurora#Response_and_aftermath`
[2] Agency, N.S.: Security-enhanced linux (2008), `http://www.nsa.gov/selinux`
[3] Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics (2004)
[4] Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Tech. rep., University of Wisconsin, Madison (2003)
[5] Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: India Software Engineering Conference. ACM (2008)
[6] Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proc. IEEE S&P (2003)
[7] Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: Proc. IEEE S&P (1996)
[8] Giffin, J.T., Jha, S., Miller, B.P.: Detecting manipulated remote call streams. In: Proc. USENIX Security Symposium (2002)
[9] Gopalakrishna, R., Spafford, E.H., Vitek, J.: Efficient intrusion detection using automaton inlining. In: Proc. IEEE S&P (2005)
[10] Kaspersky: Adobe: the number one target for hackers in the first quarter (2010), `http://www.kaspersky.com/news?id=207576094`
[11] Kumar, S., Spafford, E.: A pattern matching model for misuse intrusion detection. In: Proc. National Computer Security Conference (1994)
[12] Lee, C., Chen, F., Rosu, G.: Mining parametric specifications. In: Proc. ICSE (2011)
[13] Lee, W., Stolfo, S.J.: Data mining approaches for intrusion detection. In: Proc. USENIX Security Symposium (1998)
[14] Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection. In: Proc. IEEE S&P (2001)
[15] Lo, D., Khoo, S.C., Liu, C.: Efficient mining of iterative patterns for software specification discovery. In: Proc. ACM SIGKDD (2007)
[16] Nucci, A., Bannerman, S.: Controlled chaos. IEEE Spectrum (December 2007), `http://www.spectrum.ieee.org/dec07/5722`

[17] Safyallah, H., Sartipi, K.: Dynamic analysis of software systems using execution pattern mining. In: Proc. IEEE ICPC (2006)
[18] Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proc. IEEE S&P (2001)
[19] Sophos: Top 10 malware (June 2008), `http://www.sophos.com/security/top-10/`
[20] Szathmary, L., Napoli, A., Valtchev, P.: Towards rare itemset mining. In: Proc. IEEE International Conference on Tools with Artificial Intelligence (2007)
[21] Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proc. S&P (2001)
[22] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proc. CCS (2002)
[23] Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusions using system calls: Alternative data models. In: Proc. IEEE S&P (1999)
[24] Wespi, A., Dacier, M., Debar, H.: Intrusion Detection Using Variable-Length Audit Trail Patterns. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 110–129. Springer, Heidelberg (2000)