AccessMiner: Using System-Centric Models for Malware Protection

Andrea Lanzi
Institute Eurecom,
Sophia-Antipolis, France
lanzi@eurecom.fr

Davide Balzarotti
Institute Eurecom,
Sophia-Antipolis, France
balzarotti@eurecom.fr

Christopher Kruegel
University of California,
Santa Barbara, USA
chris@cs.ucsb.edu

Mihai Christodorescu
IBM T.J. Watson Research
Center, Yorktown Heights,
NY 10598, USA
mihai@us.ibm.com

Engin Kirda
Institute Eurecom,
Sophia-Antipolis, France
kirda@eurecom.fr

ABSTRACT

Models based on system calls are a popular and common approach to characterize the run-time behavior of programs. For example, system calls are used by intrusion detection systems to detect software exploits. As another example, policies based on system calls are used to sandbox applications or to enforce access control. Given that malware represents a significant security threat for today's computing infrastructure, it is not surprising that system calls were also proposed to distinguish between benign processes and malicious code.

Most proposed malware detectors that use system calls follow a program-centric analysis approach. That is, they build models based on specific behaviors of individual applications. Unfortunately, it is not clear how well these models generalize, especially when exposed to a diverse set of previously-unseen, real-world applications that operate on realistic inputs. This is particularly problematic as most previous work has used only a small set of programs to measure their technique's false positive rate. Moreover, these programs were run for a short time, often by the authors themselves.

In this paper, we study the diversity of system calls by performing a large-scale collection (compared to previous efforts) of system calls on hosts that run applications for regular users on actual inputs. Our analysis of the data demonstrates that simple malware detectors, such as those based on system call sequences, face significant challenges in such environments. To address the limitations of program-centric approaches, we propose an alternative detection model that characterizes the general interactions between benign programs and the operating system (OS). More precisely, our system-centric approach models the way in which benign programs access OS resources (such as files and registry entries). Our experiments demonstrate that this approach captures well the behavior of benign programs and raises very few (even zero) false positives while being able to detect a significant fraction of today's malware.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

Categories and Subject Descriptors

D.2.8 [**OPERATING SYSTEM**]: Security and Protection

General Terms

Security

Keywords

System Call, Malware, Anomaly-Based Detector

1. INTRODUCTION

Malicious code is a significant problem and the root cause of many security threats on the Internet. For example, the majority of spam mails are sent by malware-infected hosts [12], botnets are responsible for large-scale denial of service attacks, and malicious code is used by cyber criminals to compromise online banking accounts [11].

Given the importance and the security impact of malware, it is not surprising that there exists a significant body of research, both in the scientific community and the commercial world, on ways to protect machines from becoming infected and on techniques to detect and contain malware programs once they are on the host. The most popular approach to identify malware is based on signatures. These signatures are typically byte strings (or instruction sequences) that are characteristic for a particular malware instance or a family of malicious code [27]. Unfortunately, code obfuscation and polymorphism have long proved to be effective tools in the arsenal of a malware author to evade signature-based detection.

To address the limitations of signature-based detection techniques, behavior-based detection was introduced as a novel approach to identify malicious code [5, 15]. Behavior-based detectors do not examine the (static) content of a binary, but rather focus on the (dynamic) actions that the program performs, or might perform. The idea is that even

when the syntactic layout of the program is different, the semantics of the code should remain unchanged between polymorphic variants of the same malware (or even, between different members of the same malware family).

Arguably the most popular way to characterize the behavior of programs is based on some kind of analysis of the system calls (or Win32 API functions) that a program invokes; or that it can invoke, in case the code is examined statically. Various models range from looking at sequences of system calls [22], over bags of system calls [13], to system call patterns based on data flow dependencies [21, 15]. Yet other techniques look at individual system calls, but take into account argument information [14].

In most cases, authors achieved good results with system-call-based malware detectors, and they reported high success rates with low numbers of false positives. However, a closer examination of the presented results reveals that most experiments are performed on a relatively small scale. In particular, this is true for the analysis of false positives. That is, authors collect traces only for a small set of benign applications. In addition, these programs are exercised in a limited fashion, often using synthetic inputs or launching simple test cases. As a result, it is not clear whether the observed system call traces produced by these benign applications are representative for the diverse set of applications that is used by actual users. Moreover, in most cases, the experiments are performed on a single machine. It is not clear how the detection results will generalize to a larger installation base. Thus, the reported number of false positives might be underestimated.

For this paper, we set out to analyze the diversity of system call information and the robustness of simple detection techniques when looking at data that is collected on a larger scale (especially when compared to previous efforts, as discussed in Section 5). To this end, we developed a lightweight system call collection module that was installed on ten machines that were used by people to carry out their normal activities. Over a period of several weeks, these modules collected more than 1.5 billion system calls that were invoked by 362 thousand operating system processes. In total, we observed 242 different applications.

Our analysis of the collected data shows that there is a large diversity in the system calls that benign programs invoke. This makes it difficult for system-call-based detectors to establish a concise model of normality that allows for a clean separation between acceptable and malicious behavior. In other words, the fact that a previously unseen system call sequence is observed is not a good indication that the process is malicious. It might just be that an existing application is used in an unexpected fashion, or that a new, benign application is installed.

A problem with many behavior-based detection techniques, especially those that focus on system calls, is that they follow a *program-centric* approach. That is, these techniques aim to model the execution of individual programs. As a result, the models lack context that captures how benign programs – in general – interact with their environment, and in particular, the operating system. Also, detectors often take a narrow view on the execution history of a program (such as looking at a sliding window of k consecutive system calls). This myopic view fails to capture program behavior at a higher level of abstraction. The result is that the models are specific to a small set of benign applications (those that were used for training). However, they fail to identify activity that is common, and characteristic, to benign applications in general. As a result, detectors raise an alert whenever something "new" is encountered. Typically, this leads to (unacceptably) high false positive rates.

In this paper, we propose a *system-centric* view to model the activity of benign programs. More precisely, we argue that benign programs in general follow certain ways in which they use OS resources (such as the file system and the registry). For example, programs typically write only to their own directories (the ones they create as part of the installation) and to temporary directories. As another example, certain regions in the registry are only read, but never written. Malicious programs, on the other hand, often attempt to tamper with other applications and critical system settings, and hence, write to resources that benign applications never touch.

Leveraging the insights from our observation of many benign program executions, we propose techniques to automatically generate an *access activity model* that captures how programs interact with operating system resources. Of course, this model needs to abstract from individual programs and learn, for example, that the folder `c:\Program Files` contains private directories (these are folders that are supposed to be written only by their respective "owners"). Thus, even when our access activity model is deployed on a system that has a program installed that was never seen before, our model accounts for the fact that this program is expected (and allowed) to write to its sub-folder under `C:\Program Files`. An access activity model can be used to identify processes as malicious, or to automatically deny access to certain resources. As a result, our proposed system is an anomaly-based (behavior-based) detector for unknown and known malware, based on monitoring interactions of a program with persistent OS resources (files, registry).

In some ways, an access activity model is similar to access control policies defined by SELinux or the Windows Group Policy Editor. Both provide policies that define the acceptable use of resources. However, there is an important difference. Access control policies define the (minimal set of) resources needed by individual applications, and hence, they are program-centric. Our access activity model is a single policy derived and abstracted from the observation of how a broad set of diverse applications interact with the operating system in general. That is, while SELinux-style policies attempt to minimize the damage that a compromised application can cause, our policies attempt to distinguish between malicious and benign applications based on the way they interact with the OS.

Our experiments show that the access activity model is successful in identifying a large fraction of malware samples with a very low false positive rate. With a few minor tweaks to the model (that are due to incomplete training data), the false positive rate can be lowered to zero. Of course, access activity models cannot detect all possible types of malware. They can only detect cases in which malicious code attempts to tamper with the binaries or the settings of other applications or the core OS itself. As our experiments show, this is true for a large fraction of malware – after all, malware often attempts to interfere with or modify the execution of legitimate programs or the OS, or, at the very least, establish a foothold on the system. However, it is conceivable that certain kinds of malware do not tamper

with the integrity of the system. For example, one could imagine a bot that sends spam as long as the machine is not rebooted. In this case, our policies would need to be extended to network traffic as well.

To summarize, the contributions of this paper are the following:

- We performed a large scale data collection of system calls that are invoked by a diverse set of benign applications under realistic conditions. Our data set contains more than 1.5 billion system calls invoked by 242 different applications that were executed more than 362,000 times.
- We analyzed the diversity of the collected system calls and explored how system-call-based, program-centric detectors would perform in light of this data. We conclude that simple approaches based on system call sequences raise many false positives, and hence, are unlikely to work in practice.
- We propose a system-centric approach for malware detection. This approach is based on an access activity model that captures the generalized interactions of benign programs with operating system resources. We demonstrate that this model characterizes well the operations of benign programs, and it can be used to detect a substantial fraction of malware programs with very low (or even no) false positives.

In the following Section 2, we describe in more detail our infrastructure to collect system call data. Then, in Section 3, we discuss how we analyzed the data and present our conclusions about the diversity of the data set. In Section 4, we introduce our system-centric detection approach and demonstrate that it achieves good detection result.

2. SYSTEM CALL DATA COLLECTION

In this section, we discuss our efforts to collect a large and diverse set of system call traces. Our requirements are geared towards imposing the least impact on the users whose machines are part of the data collection effort. Thus, the data collection framework must have minimal impact on the performance of those machines, must operate with and without network connectivity, must ensure that private information does not leave the user's machines, and must make almost no assumptions about the run-time environment. For example, requiring that users make use of virtual machines would significantly restrict the practical applicability of our data collection. Additionally, the data collection framework must be capable of extracting a rich set of attributes for each event (i.e., system call) of interest. Unfortunately, none of the existing system call tracing tools satisfy these requirements, so we built and deployed our own data collection framework.

Our system consists of software agents, which, once installed on user's machines, automatically collect, anonymize, and upload system call logs, and a central data repository, which receives logs from each machine and normalizes the data in preparation for further analysis. The software agents can be installed by users on their own machines and are mindful of system load, available disk space, and network connectivity. Furthermore, users can enable and disable the collection agent as they wish.

Data description. We are interested in performing different types of analyses on the collected data. Thus, the data elements collected for each system call must allow analyses along many dimensions. For each system call we collect its arguments, its result (return) code, the process ID, the process name, and the parent process ID. Each log entry is a tuple (see an example in Figure 1):

$$\langle timestamp, program, pid, ppid, system\ \ call, args, result \rangle$$

This data allows us to perform our analyses within a single process, across multiple executions of the same program, or across multiple programs.

2.1 Raw Data Collection

The software agent that collects data is a real-time component running on each user's machine. This agent consists of a data collector and a data anonymizer. We implemented our agent for Microsoft Windows, as it is the OS targeted the most by malware. The description in the remainder of this section provides details specific to the Microsoft Windows platform. The data collector is a Microsoft Windows kernel module that traces system call events and annotates them with additional process information. The data anonymizer transforms the collected system call data according to privacy rules and uploads it to the remote, central data repository.

Kernel collector. The main goal of this component is to collect system call and process information *across the entire system*. In order to intercept and log system call information, the kernel data collector hooks the SSDT table [10]. The kernel collector logs information for 79 different system calls in five categories: 25 related to files, 23 related to registries, 25 to processes and threads, one related to networking, and five related to memory sections. We selected the same subset of system calls that are used in Anubis [1], which covers the relevant operations that manipulate persistent OS resources.

A challenge arises from the fact that the kernel collector does not necessarily observe the start of a new process. One reason is that the user can disable and re-enable the software agent at any point. Another reason is that the kernel collector is started as the last kernel module in the system boot process. This means that the kernel collector might observe system calls that refer to previously acquired resource handles, but without having any information about which resources those handles point to. As a special case, some resource handles (e.g., handles to the registry roots) are automatically provided to a process by the OS at process-creation time. Consequently, if we log only the parameters for each individual system call that we observe, we lose information about previously (or automatically) acquired resources. To address this problem, we query the open handler table for each process we have not seen before. This allows the kernel collector to retrieve the open objects already associated with a new process. We store the path names of these objects for later use when we intercept a system call (such as `NtOpenKey`) that references a pre-existing handle.

Log anonymizer. To protect the privacy of our users, we obfuscate or simply remove arguments of various system

Program	*System call*	*Arguments*		*Return value*
		In	*Out*	
SVCHOST.EXE	NtCreateFile	131208,"...\ACGENRAL.DLL",7,0	2600	0
SVCHOST.EXE	NtQueryInformationFile	2600,6,0		-144573084
SVCHOST.EXE	NtClose	1004		0
CLIENT.EXE	NtClose	1560		0
CLIENT.EXE	NtCreateNamedPipeFile	2148532480,"...\NamedPipe\...041",3,32	288	0
CLIENT.EXE	NtOpenFile	1074790528,"...\NamedPipe\...041",3,96	264	0
firefox.exe	NtReleaseSemaphore	404		0
firefox.exe	NtReadFile	780		0

Figure 1: Sample system-call log. Due to formatting constraints, some values are abbreviated and the timestamp, process ID, and parent process ID fields are not shown.

calls before sending the log to the data repository. The obfuscation consists of replacing part or whole of a sensitive argument value with a randomly-generated value. Every time a value repeats, it is replaced with the same randomly-generated value, so that we can recover correlations between system call arguments. We consider as sensitive all arguments whose values specify non-system paths (e.g., paths under `C:\Documents and Settings` are sensitive), all registry keys below the user-root registry key (`HKLM`), and all IP addresses. Furthermore, we remove all buffers read, written, sent, or received, thus both providing privacy protection and reducing the communication to the data repository. The data repository indexes the logs by the primary MAC address of each machine.

Impact on performance. We designed the software agent to minimize the overhead on users' activities. The kernel module collects information only for a small subset of 79 system calls. Log are saved locally and processed out of band before being sent to the server, when network connectivity is available. Users can turn data collection on and off, based on their needs. Local logs are uploaded to the repository when they reach 10 MB in size and logging is automatically stopped if available disk space drops below the 100 MB threshold. Each 10 MB portion of the system call log is compressed using ZIP compression, for an 95% average reduction in size (from 10 MB to 500 KB). Given these techniques, we are confident that users were able to use their computers with the data collector present as they would normally do, and thus the collected system call logs are representative of day-to-day usage.

2.2 Data Normalization

The purpose of this component is to process the raw system call logs and extract the fully qualified names of the accessed resources as well as the access type. For files and directories, the fully qualified name is the absolute path, while for registry keys, it is the full path from one of the root keys.

To compute fully qualified resource names, we track for each process the set of resources open at any given time, via the corresponding set of OS handles. When a resource (file or registry key) is accessed relative to another resource (either opened by the process or opened by the OS automatically for the process), we combine the resource names to obtain a fully qualified name. Symbolic links are handled by observing the actual open operation of the target (linked) file. To handle hard links, we include the locations (paths) of all hard-linked aliases of a file.

Computing the access type (e.g., read, write, or execute) requires tracking the access operations performed on a resource. This is more tricky than expected. When a resource is acquired by a program (e.g, a file is opened), the program specifies a desired level of access. This information, however, is not sufficiently precise for our needs. This is because, often, programs open files and registry keys at an access level beyond their needs. For example, a program might open a file with `FULL_ACCESS` (i.e., both read and write access), but afterward, it only reads from the file. Since we are interested in the actual access type, we track all of the operations on a resource, and only when the resource is released (on `NtClose`), we compute the access type as a union of all operations at all on the resource. If the program performs no operations on a resource, then we use the initially-requested access (provided at resource acquisition) as actual access. This heuristic is used to cope with memory-mapped files. In fact, with such files, we might not see any read/write operation at the system call level, although the file is accessed.

In Microsoft Windows, there is no single system call that starts a new process from a given executable file. In order to retrieve the execution path and file name, the normalizer needs to recognize the `NtOpenFile` system calls that belong to the process-creation task. When a process is created, the OS executes a set of system calls to allocate resources, load the binary executable, and start the new process: `NtOpenFile`, `NtCreateSection` with desired executable access, and `NtCreateThread`. Consequently, we automatically identify occurrences of this pattern and extract the executable path and file name.

2.3 Experimental Data Set

We deployed our data collection framework on ten different machines, each belonging to a different user, all running Microsoft Windows XP. The users have different levels of computing expertise and different computer usage patterns. Based on use, the machines can be classified as follows: two were development systems, one was an office system, one was a production system, four were home PCs, and one was a computer-lab machine.

Overall we collected 114.5 GB of data, consisting of 1.556 billion of system calls, from 362,600 processes and 242 distinct applications. 1 provides detailed information for each machine. Our system collected data from each machine at an average rate of 8.2 MB/minute, with highly used machines producing logs at 40 MB/minute and idle machines

Machine	*Data* (GB)	*System calls* ($\times 10^6$)	*Processes* ($\times 10^3$)	*Applications*
1	18.0	285	55.1	90
2	4.5	70	22.4	87
3	5.6	89	17.7	46
4	32.0	491	110.9	41
5	34.0	514	125.6	42
6	14.0	7	2.8	73
7	1.3	19	3.7	49
8	1.2	18	3.0	22
9	1.6	27	8.5	47
10	2.3	36	12.9	26
Total	114.5	1,556	362.6	242

Table 1: Characteristics of our data set.

Machine	*Usage*	*Data*	*Time*		*Data rate*
			Logged	*Total*	
		(GB)	(hours)	(days)	(MB/minute)
1	office	18.0	12	3	8
2	home	4.5	4	3	6.25
3	home	5.6	3	4	7.77
4	prod.	32.0	12	3	14
5	prod.	34.0	12	3	15
6	lab	14.0	8	3	11
7	home	1.3	3	2	4
8	home	1.2	3	2	4
9	dev.	1.6	2	2	6
10	dev.	2.3	2	3	6.4

Table 2: Data rates during collection.

producing 1.5 MB/minute. In 2, we report the logging time for the ten different machines. For each machine, we show the machine's usage profile, the size of data collected, the total time during which data was actually collected, the time period between the first log entry and the last log entry, and the average data rate. For example, the fourth row indicates that machine 4 was a production server that generated 32 GB of system call logs, over a period of 3 days, during which data collection was active for 12 hours. Our training data includes every OS task that was executed during the monitoring period, including software installation and updates.

3. ANALYSIS OF SYSTEM CALL DATA

The previous section has described the data set of system call traces that we collected in the wild from ten real-world users. In this section, we seek to answer the following question: "How diverse is the collected system call data?" For this analysis, we focus on the *types* of system calls that an application invokes, and ignore argument values. The reasons for this decision are twofold. First, the use of system call type information as the sole source for modeling application behavior has a long tradition in the security community. It dates back at least 15 years to the first anomaly intrusion detection system that proposed to establish a "sense of self" for Unix processes [9]. As a result, a majority of previously proposed techniques to model the behavior of applications uses only system call type information, even though the models and the techniques to extract these models vary.

The second reason for using system call types in our analysis is that many models proposed for this kind of input data share common characteristics. Given that the data is a stream of system call numbers associated with the execution of a program, there are limits to the ways in which this data can be modeled. In particular, most models rely upon characteristic patterns (or sequences) derived from the observed system calls. As a result, we hope that our analysis results are valid in a broader context than just for the specific model(s) that we chose to study.

We analyze the diversity of the system call data in relationship to a particular model used to capture program behaviors. More precisely, we seek to understand how well the collected system call data can be characterized by a given model. If the data is diverse, it will be difficult to build a model that is successful in characterizing the executions of benign programs. If the data is uniform, then benign programs behave similarly from the model's point of view, and it is easy to capture their executions. That is, we cast the problem of studying the diversity of our data set as the problem of understanding whether a model is able to capture the data in a precise fashion.

For this paper, we decided to use n-grams as the basic technique to models system calls. The n-gram model is very popular and has been used as part of many different security solutions. For example, n-grams were used to model program activity to detect software exploits and to identify malicious code in network payloads. Other examples include the detection of malware-infected documents and, of course, the detection of malicious processes based on the invoked system calls. More complex models are possible, for example, those that take into account data flows (taint information) between system call arguments. However, collecting and/or enforcing data flow at end hosts is very challenging, especially on real machines and with minimal performance impact. Also, our collected data set does not contain data flow information.

3.1 Creating n-gram Models

To obtain a set of n-grams for a program, the sequence of system calls invoked by the running program is scanned with a sliding window of size n. Each unique sequence of length n (called an n-gram) is added to the model for this program. For example, consider that an application invokes five system calls in the following order: $12, 3, 17, 9, 11$. In this case, the 3-gram model for this application would be the set of triples $\{< 12, 3, 17 >, < 3, 17, 9 >, < 17, 9, 11 >\}$. We use only system call numbers for building n-gram models. Directly including parameter values in the model is hard because there is no agreed-upon mechanism to generalize and model system call parameters.

When generating an n-gram model for malware detection, we follow a "standard" approach. That is, we first extract n-gram models for a set of malware programs and a set of benign programs (called the training data). Then, we find all n-grams that appear in some of the malware models but *not* in any of the models built for the benign programs. The intuition is that those n-grams are characteristic for malware, since they were only seen in the context of the execution of malicious code.

Using the n-grams that uniquely characterize malware, we can perform detection. To this end, we monitor the execution of a process and inspect the sequence of system system

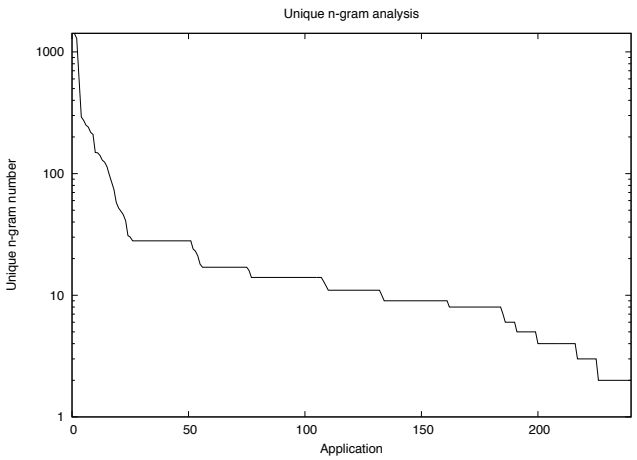

Figure 2: Unique n-grams for all 242 applications in our data set.

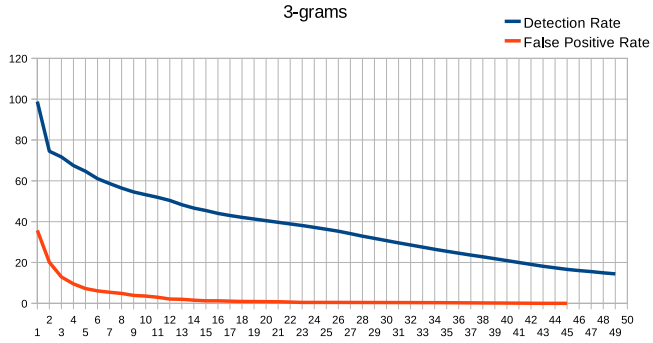

Figure 3: Detection results for 3-gram detector. The detection threshold k is on the X-axis (e.g., $k = 5$ means that a system-call trace must match five 3-grams for an alert). The Y-axis represents the detection and false-positive rates, respectively.

calls that it invokes. When this sequence contains more that k instances of n-grams that are considered malicious, the process is identified as malware.

3.2 Detection Results

To understand the diversity of our collected system call data set, we performed a number of experiments. First, we examined the number of unique n-grams that can be found in each of the 242 different applications that we observe. Under the assumption that n-grams are a good model to capture program behavior in general, we would expect that the number of such unique sequences is low. Otherwise, every time a new application is introduced, the model will observe system call sequences that have never been seen before. Thus, a detector would either raise a false alarm or miss malicious software that contains such sequences.

The number of unique n-grams for all 242 applications is shown in Figure 2, sorted in descending order. We can see that the traces for almost all applications contain unique n-grams. In fact, new n-grams were found for all but two applications (small utilities, `instdrv.exe` and `nwiz.exe`)! The y-axis is in logarithmic scale to better demonstrate the long tail. Interestingly, those applications for which we found the largest number of unique n-grams are also those that are frequently used (the top-5 applications were `explorer.exe`, `svchost.exe`, `acrotray.exe`, `firefox.exe`, and `iexplore.exe`). This finding suggests that n-grams do not closely relate to benign or malicious behavior; instead, more and different n-grams are observed simply when more processes are executed.

We then performed a number of experiments that analyze the detection capabilities of n-gram-based techniques. For these experiments, we introduce an additional data set that captures the execution of 10,838 malware samples. These samples, taken from recent Anubis [3] submissions, are a representative mix of current malware programs and contain worms, bots, and file infectors.

For a particular choice of values for n and k, we ran ten experiments, one for each of the ten machines in our data set. More precisely, we use the system call traces from nine machines and a random selection of two-thirds of the malware set to train an n-gram model. Then, we use this model to perform detection on the last third of the malware samples (to determine the true positives) and the system call traces taken from the tenth machine (to determine the false positives). The reported detection results are averaged over the ten runs (one for each machine).

When computing false positives, we used the number of different applications as a basis, and not the number of processes. That is, when the detector raises alerts for ten processes, but all these processes were running the same application, we count this as a single false positive. The reasons for this decision are the following: First, we did not want to bias the results to the popularity of a certain application and the number of times that it appears in our data set. Second, when a detector raises frequent alerts for the same application, it is easy to white-list this particular program.

We ran experiments with a number of different parameter settings, varying the length n of the n-grams and the number k of malicious n-grams that need to be encountered before a program is classified as malicious. The results for $n = 3$ and k ranging over an interval from 1 to 50 are shown in Figure 3. The results for $n = 2$ and $n = 4$ are worse. Figure 4 shows the detection results for an n-gram-based detector with $n = 2$, while Figure 5 shows results for $n = 4$. It can be seen that for $n = 2$, detection rates are very poor. The results for $n = 4$ are significantly better, but still worse than the ones for $n = 3$.

One can see that the detection rates are very high, especially for small values of k. However, the false positive rates are very high as well (almost 40% of all benign applications are reported as malicious). The reason is that the data set is very diverse. That is, it is common that benign applications produce system call sequences that match n-grams that were previously considered to be indicators of malicious activity. One could lower the false positives by increasing the threshold k. However, in this case, detection rates drop significantly as well.

4. SYSTEM-CENTRIC MODELS AND DETECTION

The results presented in the previous section have shown that models based on system call sequences (n-grams) have

difficulties in distinguishing normal and malicious behaviors. The main reason is that system-call sequences invoked by benign applications are diverse. As a result, a malware detector will likely encounter previously-unseen n-grams when monitoring benign processes. Unfortunately, the presence of such n-grams is not a good indication that the code that is executing is actually malicious. A problem is that while n-grams might capture well executions of individual programs, they poorly generalize to other applications. The reason is that the model is closely tied to the execution(s) of particular applications; we refer to this as a program-centric detection approach.

In this section, we propose a model that attempts to abstract from individual program runs and that generalizes how, in general, benign programs interact with the operating system. For capturing these interactions, we focus on the file system and the registry activity of Microsoft Windows processes. More precisely, we record the files and the registry entries that Windows processes read, write, and execute (in case of files only). It is possible to integrate other kinds of interactions into our model (in particular, the network), but we leave this for future work.

Our model is based on a large number of runs of a diverse set of applications, and it combines the observations into a single model that reflects the activities of *all* programs that are observed. For this to work, we leverage the fact that

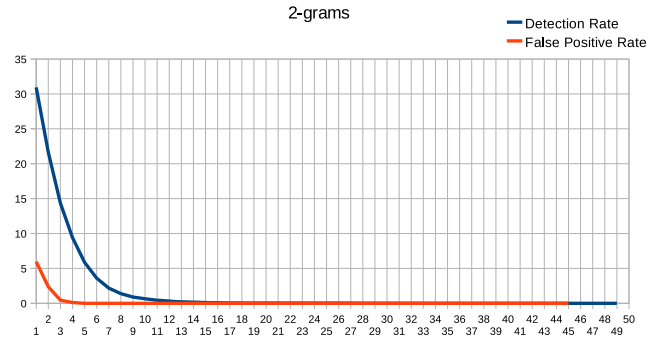

Figure 4: Detection results for 2-gram detector and varying settings of k.

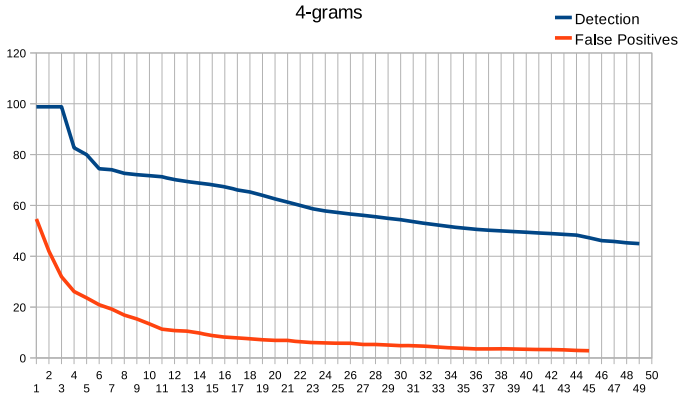

Figure 5: Detection results for 4-gram detector and varying settings of k.

we see "convergence." That is, even when we build a model from a subset of the observed processes, the activity of the remaining processes fits this model very well. Thus, by looking at program activity from a system-centric view – that is, by analyzing how benign programs interact with the OS – we can build a model that captures well the activity of these programs. Of course, this would not be sufficient by itself. To be useful, our model must also be able to identify a reasonably large fraction of malware. To demonstrate that this is indeed the case, we have performed a number of experiments that are described in more detail in Section 4.3. Note that it is also possible that program-centric models converge at one point. However, the results presented in the previous section indicate that a large amount of data is needed before this point is reached; more than we collected in our experiments, and definitely more than previous research has used to demonstrate that system-call-sequence-based detection works (for a more detailed discussion of the data examined in previous work, refer to Section 5).

4.1 Creating Access Activity Models

To capture normal (benign) interactions with the file system and the Windows registry, we propose *access activity models*. An access activity model specifies a set of labels for operating system resources. In our case, the OS resources are directories in the file system and sub-keys in the registry (sub-keys are the equivalent of directories in the file system). In the following, we refer to directories and sub-keys as folders.

Note that we do not specify labels directly on files or registry entries. The reason for this was that the resulting models are significantly smaller when looking at folders only. As a result, the model generation process is faster and "converges" quicker (i.e., less training data is required to build stable models). Moreover, in almost all cases, the labels for the folder entries (files or registry keys) would be similar to the label for that folder itself. Thus, the sacrifice in precision was minimal.

A label L is a set of access tokens $\{t_0, t_1, \ldots, t_n\}$. Each token t is a pair $\langle a, op \rangle$. The first component a represents the application that has performed the access, the second component op represents the operation itself (that is, the type of access).

In our current system, we refer to applications by name. In principle, this could be exploited by a malware process that decides to reuse the name of an existing application (that has certain privileges). In the future, we could replace application names by names that include the full path, the hash of the code that is being executed, or any other mechanism that allows us to determine the identity of the application that a process belongs to. In addition to specific application names, we use the star character (`*`) as a wildcard to match any application.

The possible values for the operation component of an access token are `read`, `write`, and `execute` for file-system resources (directories), and `read` and `write` for registry sub-keys.

Initial access activity model. An initial access activity model precisely reflects all resource accesses that appear in the system-call traces of all benign processes that we monitored (we call this data set the training data). Note that for this, we merge accesses to resources that are found in differ-

ent traces and even on different Windows installations. In other words, we build a "virtual" file system and registry that contains the union of the resources accessed in all traces.

Whenever an application `proc` opens or reads from an existing file `foo` in directory `C:\path\dir`, we insert the directory `dir` into our "virtual" file system, including all directories on the `path` to `dir`. When a prefix of the directories along `path` already exist in our virtual file system, then these directories are re-used. All directories that are not already present (including `dir`) are added to the virtual file system tree. Then, we add the access token $\langle$`proc`, `read`$\rangle$ to the label associated with `dir`.

When a process creates or deletes a file in a directory `dir`, or when it writes to a file, then we use the operation `write` for the access token. Similar considerations apply for read and write operations that are performed on the registry. Finally, whenever a binary is executed (loaded by the OS loader), then we add a token with `execute` to the directory that stores this binary.

For example, consider that file `C:\dir\foo` is read by pA on machine A, and that file `C:\dir\sub\bar` is written by pB on another machine B. Then, the resulting virtual file system tree would have `C:\` as its root node. From there, we have a link to the directory `dir`, which in turn has a link to `sub`. The label associated with `dir` is $\langle pA, read\rangle$, and the label associated with `sub` is $\langle pB, write\rangle$.

Pre-processing. Before model generation can proceed, there are two additional pre-processing steps that are necessary. First, we need to remove a small set of benign processes that either read or execute files in many folders. The problem is that these applications appear in many labels and could lead to an access activity model that is less tight (restrictive) than desirable. We found that such applications fall into three categories: Microsoft Windows services (such as Windows Explorer or the command shell) that are used to browse the file system and launch applications; desktop indexing programs; and anti-virus software. The number of different applications that belong to these categories is likely small enough so that a manually-created white list could cover them. In our system, we remove all applications that read or execute files in more than ten percent of the directories. We found a total of 15 applications that fit this profile: nine Windows core services, two desktop indexing applications, and six anti-virus (AV) programs. Identifying such applications automatically is reasonable, because we assume that our training data does not contain malicious code. However, the number of white-listed applications is so small that the entries can be easily verified manually.

The second pre-processing step is needed to identify applications that start processes with different names. We consider that two processes with different names belong to the same application when their executables are located in the same directory. We have found 14 applications that start multiple processes with different names. These include well-known applications such as MS Office, Messenger, Skype, and RealPlayer. Of course, all Windows programs that are located in `C:\Windows\system32` are also aggregated (into a single meta-application that we refer to as `win_core`). Merging processes that have different names but that ultimately belong to the same application is useful to create tighter access activity models.

Model generalization. Based on the initial access activity model, we perform a generalization step. This is needed because we clearly cannot assume that the training data contains all possible programs that can be installed on a Windows system, nor do we want to assume that we see all possible resource accesses of the applications that we observed. Also, the initial model does not contain labels for all folders (recall that the access is only recorded for the folder that contains the accessed entity).

The generalization step performs a post-order traversal of both the virtual file system tree and the virtual registry tree. Whenever the algorithm visits a node, it performs the following four steps:

Step 1: First, the algorithm checks the children of the current node to determine whether access tokens can be *propagated* upward in the tree. Intuitively, the idea is that whenever we inspect a folder (node) and observe that all its sub-folders are accessed by a single application only, we assume that the current folder also belongs to this application.

More formally, the upward propagation rule works as follows: For each operation *op*, we examine the labels of all child nodes and extract the access tokens that are related to *op*. This yields a set of access tokens $\{t_1, \ldots, t_n\}$. We then inspect the applications involved in these accesses (i.e., the first component of each token t_i). When we find that all accesses were performed by a *single* application `proc`, we add the access token $\langle proc, op\rangle$ to the current label.

Step 2: The upward propagation rule of Step 1 is used to identify parts of the file system or the registry that belong to a single application. However, this is problematic when considering *container* folders. A container is typically a directory that holds many "private" folders of different applications. A private folder is a folder that is accessed by a single application only (including all its sub-folders). A well-known example of a container is the directory `C:\Program Files`, which stores the directories of many Windows programs.

Since a container holds folders owned by many different applications, its label would deny access to all sub-folders that were not seen during training. This might be more restrictive than necessary. In particular, we would like to ensure that whenever an application accesses a previously-unseen folder in a container, this should be allowed. Intuitively, the reason is that this access follows an expected "pattern," but the specific folder has not been seen during training. To handle these cases, we introduce a special flag that can be set to mark a folder as a container.

The following rule is used to mark a folder as a container: Similar to before, we examine the labels of all child nodes and extract the access tokens that are related to each operation *op*. We then inspect the set of access tokens that is extracted $\{t_1, \ldots, t_n\}$. When the applications in these accesses are different, but there is *no wildcard* present in any access token, then the folder is marked as container. We explain the implications of a *container flag* for detection in Section 4.2.

Step 3: Next, the access tokens in the label associated with the current node are *merged*. To this end, the algorithm first finds all access tokens that share the same operation *op* (second component). Then, it checks their application names (first components). When all tokens share the same application name, they are all identical, and we keep a single copy. When the application names are different, or one

token contains the wildcard, then the tokens are replaced by a single token in the form $\langle *, op \rangle$. Merging is useful to generalize cases in which we have seen multiple applications that perform identical operations in a particular folder, and we assume that other applications (which we have not seen) are also permitted similar access.

Step 4: Finally, the algorithm adds access tokens that were likely missed because of the fact that the training data is not complete. More precisely, for each access token that is related to a *write* operation, we check whether there exists a corresponding *read* token. That is, for all applications that have written to a folder, we check whether they have also performed read operations. If no such token can be found, we add it to the label. The rationale for this step is that an application that can write to resources in a folder can very likely also perform read operations. While it is possible to configure files and directories for write-only access, this is very rare. On the other hand, adding read tokens allows us to avoid false positives in the more frequent case where we have simply not seen (legitimate) read operations in the training data.

When the generalization algorithm completes, all nodes in the virtual file system and the registry tree have a (possibly empty) label associated with them.

Note that, for building the access activity model, we do not require any knowledge about malicious processes. That is, the model is solely built from generalizing observed, good behavior. This is an advantage compared to the n-gram model introduced in the previous section, which requires training traces captured from malware runs to identify those n-grams that are unique to benign applications.

4.2 Model Enforcement and Detection

Once an access activity model M is built, we can deploy it in a detector. More precisely, a detector can use M to check processes that attempt to read, write, or execute files in directories or that read or write keys from the registry.

The basic detection algorithm is simple. Assume that an application *proc* attempts to perform operation *op* on resource r located in `\path\dir`. We first find the longest prefix P shared between the path to the resource (i.e., `\path\dir`) and the folders in the virtual tree stored by M. For example, when the virtual file system tree contains the directory `C:\dir\sub\foo` and the accessed resource is located in `C:\dir\sub\bar`, the longest common prefix P would be `C:\dir\sub`. We then retrieve the label L_P associated with this prefix and check for all access tokens that are related to operation *op* (actually, after generalization, there will be at most one such token, or none). When no token is found, the model raises an alert. When a token is found, its first component is compared with *proc*. When the application names match or when the first component is $*$, the access succeeds. Otherwise, an alert is raised.

The situation is slightly more complicated when the folder that corresponds to the prefix P is marked as *container*. In this case, we have the situation that a process accesses a sub-folder of a container that was not present in the training data. For example, this could be a program installed under `C:\Program Files` that was not seen during training. In this case, the access is *permitted*. Moreover, the model is dynamically extended with the full path to the resource, and all new folders receive labels that indicate that application *proc* is its owner. More precisely, we add to each label access tokens in form of $\langle proc, op \rangle$ for all operations. This ensures that from now on, no other process can access these newly "discovered" folders. This makes sense, because it reflects the semantics of a container (which is a folder that stores sub-folders that are only accessed by their respective owners).

Whenever an alert is raised, we have several options. It is possible to simply log the event, deny that particular access, or terminate the offending process.

4.3 Detection Results

In this section, we evaluate the effectiveness of a detector based on access activity models. Similar to the analysis for the n-gram model, we ran ten experiments. More precisely, for each experiment, we picked one of the machines. We then used the system call data recorded on the other nine hosts to generate the access activity model, as described in the previous section. Finally, we used this model for detection. For this, we first check the resource accesses performed by all processes on the machine that was *not* used for model generation. Then, we examine the accesses performed by the malware samples. For each experiment, we evaluate the detection capabilities and false positives of the file system model alone, the registry model alone, and both models combined.

File system access activity model. On average, the file system access activity model contains about 100 labels. These labels contain tokens that restrict read access to about 70 directories, write access to about 80 directories, and execute access to about 30 directories. The results for the file system model are shown in Table 3. In this table, we see a number of different columns for the detection rates and the false positive rates. These are discussed in the following paragraphs.

When using the original model to check all read, write, and execute accesses, we see an average detection rate of 66% for the malware samples (column *Detection rate*) and a false positive rate of almost 15% (column *False positive rate*). Note that, similar to the experiments with the n-gram models, the false positive rates are computed on the basis of applications and not processes.

At first glance, the results appear sobering. However, a closer examination of the result reveals interesting insights. First, we decided to investigate the false negative rate in more detail. When looking at the execution traces of the malware programs, we observed that many samples did not get far in their execution but quickly exited or crashed. Interestingly, a substantial fraction of malicious samples never wrote to the file system or the registry, and they did not open any network connections. It is difficult to confirm that these samples exhibit any malicious activity at all. In fact, this calls into question the occasionally very high detection rates of the n-gram-based model, and it further confirms our previous insight that system call sequences are not closely related to actual malicious behavior. As a result, we decided to remove from our malware data sets all samples that never perform a write operation or open a network connection (or socket). This decreases our malware data set to 7,847 samples that exhibit at least some kind of activity. It also improves our detection rate to more than 90%, as can be seen in column *Adjusted detection rate* of 3. For the remainder of

Machine	*Detection rate*	*False positive rate*	*Adjusted detection rate*	*Rates of detected access violations*			*Detection rate (only writes)*	*Final detection*	
				Read	*Write*	*Execute*		*FP rate*	*Det. rate*
1	0.656	0.225	0.906	0.000	0.022	0.222	0.864	0.0	0.864
2	0.657	0.173	0.907	0.000	0.011	0.172	0.902	0.0	0.902
3	0.657	0.154	0.907	0.000	0.130	0.043	0.902	0.0	0.902
4	0.657	0.156	0.907	0.024	0.049	0.122	0.902	0.0	0.902
5	0.657	0.143	0.907	0.024	0.024	0.095	0.902	0.0	0.902
6	0.635	0.242	0.877	0.014	0.055	0.242	0.868	0.0	0.868
7	0.657	0.267	0.907	0.020	0.041	0.265	0.901	0.0	0.901
8	0.657	0.045	0.907	0.000	0.045	0.000	0.902	0.0	0.902
9	0.657	0.025	0.907	0.000	0.025	0.000	0.902	0.0	0.902
10	0.657	0.050	0.907	0.000	0.038	0.038	0.902	0.0	0.902
Average	0.655	0.148	0.904	0.008	0.044	0.137	0.895	0.0	0.895

Table 3: Detection based on our file-system access activity model.

this paper, all reported detection rates are computed based on the adjusted malware data set.

In the next step, we investigated the false positives in more detail. Table 3 shows the access violations for each machine, divided into violations due to read (column *Read*), write (column *Write*), and execute (*Execute*) access attempts. It can be seen that execute violations account for a significant majority of false positives. However, we also found that they are only marginally important for detection. Thus, for the next experiment, we decided to use only the access tokens that refer to `write` operations. This is justified by the fact that we are most interested in preserving the integrity of the operating system resources. The detection results for the new *write-only* detection approach are presented in column *Detection rate (only writes)* of 3. As can be seen, the numbers remain high with 89.5%. This confirms that write access violations are a good indicator for malicious activity. With this approach, the false positives are identical to the write violations, which are shown in column *Write*.

We further examined the reasons for the remaining write violations. It turned out that these violations were due to two root causes. One set of false positives was caused by our own system-call logging component that wrote temporary files directly into the `C:\` directory before sending the data over the network. The second reasons was due to software updates. More precisely, we detected a number of cases in which an application was writing to its folder in `C:\Program Files`. Of course, only this program had read-/execute access to that directory. However, we never saw a write access during training, and as a result, the directory was considered read-only. To accommodate for updates, we manually added a rule to the model that would grant write permission to applications that "own" directories in `C:\Program Files`. Moreover, we granted our component write access to `C:`. With more extensive training, both access activities would have very likely been added automatically. The model that incorporated our minor adjustments generated no more false positives, as shown in column *Final detection/FP rate*. However, the detection capabilities of the model remain basically unchanged, as shown in column *Final detection/det. rate*.

Registry access activity model. On average, the registry access activity model contains about 3,000 labels, significantly more than the file-system model. The labels contain tokens that restrict read access to about 1,600 keys and write access to about 2,800 keys (execute is not defined for registry keys).

Machine	*Detection rate*	*False positive rate*	*Det. rate (only writes)*	*FP rate (only writes)*	*Final det. rate*
1	0.567	0.063	0.530	0.063	0.521
2	0.557	0.107	0.540	0.053	0.521
3	0.566	0.179	0.530	0.128	0.062
4	0.557	0.000	0.530	0.000	0.540
5	0.557	0.000	0.530	0.000	0.540
6	0.557	0.015	0.530	0.000	0.540
7	0.597	0.133	0.530	0.000	0.540
8	0.557	0.067	0.530	0.067	0.537
9	0.561	0.100	0.530	0.025	0.521
10	0.557	0.000	0.530	0.000	0.540
Average	0.563	0.066	0.530	0.034	0.486

Table 4: Detection based on our registry access activity model.

The results for the registry model are shown in 4. The columns *Detection rate* and *False positive rate* show the detection rates and the false positive rate, respectively, for the original model. It can been seen that both the detection rate and the false positive rates are lower than for the file system model. We also examined the detection rate and the false positive rate when considering only write operations (columns *Det. rate (only writes)* and *FP rate (only writes)*). Similar to the file system case, the false positive rate drops significantly; there are five runs in which no false positives were reported at all. However, the detection rate remains (relatively) high.

We also examined the cases for which the registry access model raises false positives. We found that all registry write access violations can be attributed to the sub-tree `HKEY_USERS\Software\ Microsoft`. While this is an important part of the registry that contains a number of security settings, we wanted to understand the detection capabilities of a model that permits write access to these keys. To this end, we added a manual rule to allow writes to this subtree and re-run the experiments on the malware data set. We see that the model is still effective and achieves a detection rate of over 48% (shown in column *Final det. rate*

of 4) with no false positives. Considering the significantly larger size of the registry models compared to the ones for the file system, we expect that a larger training set would be required to effectively capture legitimate writes to the `Software\Microsoft` sub-tree.

Full access activity model. For the final experiment, we combined those improved file system and registry models that yielded zero false positives. The combined detection rate improves compared to the file system model alone, but only slightly (between 1% and 2% for all of the ten runs). The average detection improved from 89.5% to 91% (of course, with no false positives).

Discussion. When focusing on write operations only, our access activity model achieves a good detection rate (more than 90%) with a very low false positive rate. The false positive rate even drops to zero with minor manual adjustments that compensate for deficiencies in the training data, while still retaining its detection capabilities. This suggests that a system-centric approach is suitable for distinguishing between benign and malicious activity, and it handles well even applications not seen previously. This is because most benign applications are written to be good operating system "citizens" that access and manage resources (files and registry entries) in the way that they are supposed to. In fact, as we can observe from our results, out of 242 distinct applications seen in our experiments, policy violations occurred only for few, specific classes of programs (system utilities and AV software). On the other hand, violations of n-gram models occurred across the board.

Malicious programs frequently violate good behavior, often because their goals inevitably necessitate tampering with system binaries, application programs, and registry settings. Of course, we cannot expect to detect all possible types of malicious activity. In particular, our detection approach will fail to identify malware programs that ignore other applications and the OS (e.g., the malware does not attempt to hide its presence or to gain control of the OS) and that carry out malicious operations only over the network. For these types of malicious code, it will be necessary to include also network-related policies.

Our system-models can be enforced efficiently. As discuss in Section 2.1, the data collection overhead of the system call monitoring component is already very low. Enforcement is even faster, since no writes (for logging) occur.

5. RELATED WORK

The existing papers most relevant to the our current work focus on malware detection at the system call and the system library interfaces. These interfaces best describe the system resources manipulated by a program (e.g., files, other programs, other processes, configuration data, authentication and authorization information, network communication channels), making system call-based detectors comparable to our access activity model.

Malware Detection. Malware detection has looked at many ways to describe program behavior, and corresponding models evolved to keep pace with the increasing complexity of malware. Early detection mechanisms were based on particular byte sequences in the program binary that were indicative of malware. Over time, obfuscation strategies pursued by malware writers forced detectors to move to regular expressions over bytes [27], and eventually rendered them obsolete as byte patterns have little predictive power (i.e., they can accurately capture only previously seen malware). Other models such as byte n-grams [19], system dependencies of the program binary [25], and syntactic sequences of library calls [29, 22] have been proposed with limited success. Because these models have a strong syntactic aspect in that they capture artifacts of program binary unrelated to the malicious behavior, malware writers managed to evade such defenses and produce new, undetected malware. Our emphasis on a system-centric approach to modeling resource interactions bypasses such syntactic artifacts.

The software-diversity tactics employed by malware writers required new detection techniques that could capture more of the intent of the program and less of the syntactic characteristics of the program binary. The research efforts have focused on describing 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. Bruschi et al. [4] and Kruegel et al. [17] showed that some classes of obfuscations could be rendered innocuous by modeling programs according to their instruction-level control flow, while Christodorescu et al. [5] and Kinder et al. built obfuscation-resilient detectors based on instruction-level information flow. Nonetheless, instruction sequences are fungible and there are many ways to implement the same high-level functionality. Detection techniques then raised the bar by capturing information flow at the level of library calls, as proposed by Kirda et al. [14], system calls, as proposed by Kolbitsch et al. [15], Christodorescu et al. [6], Martignoni et al. [21], and Stinson et al. [26], and OS resources, as proposed by Yin et al. [30]. The respective evaluations of each of these techniques shows that as the models used in detection more closely describe actual OS resources, the detection rates significantly increase and the false-positive rates decrease. Unfortunately the library and system-call interfaces are rich enough that mimicry attacks are still possible [16, 28]. This observation guided our choice of system resources as the basic element in our models, discarding any information about the order in which resources are accessed. Furthermore we focus strictly on system resources that are shared across processes (i.e., files, registry, network connections) and we ignore single-process resources such as virtual memory. Beyond proposing a richer, system-centric model of program behavior, we made a concerted effort to improve an often overlook evaluation aspect, the external validity of the experimental settings. This concerns the number and diversity of benign and malicious programs used to evaluate a detection technique, as well as the environment in which they are exercised (in the case of detectors that rely on runtime information). For example, Kirda et al. [14] evaluated their system against 33 malware samples and 18 benign samples, each samples executed for 30–60 seconds. Kolbitsch et al. [15] used 563 malware samples and 10 benign samples, executed for up to 5 minutes. Christodorescu et al. [6] evaluated 16 malware samples and 6 benign samples for up to 4 minutes, similar to the test sets used by Martignoni et al. [21] (7 malware, 6 benign) and to Stinson et al. [26] (6 malware, 9 benign). Yin et al., in their PANORAMA system, evaluated 42 malicious samples and 56 benign ones, for 5 minutes. What is common to all of these

evaluations is that both the numbers of malicious samples and of benign samples are quite small. On current systems, regular users often run tens of interactive applications and hundreds of background processes, casting doubt on the relevance of results obtained from a few benign samples. Furthermore, evaluations in previous work were performed in virtualized, constrained environments, where interactive applications were exercised mechanically in ways that do not necessarily reflect real-life usage. We addressed these limitations by collecting execution traces of benign applications from actual users, during the course of their normal interaction with their personal systems. We designed our system to have low overhead and to anonymize all collected information, so that the users had no concerns and were not impacted in their regular use of their machine. The benign data we collected covered 242 distinct benign applications ran by ten users in their own environments.

It is worth pointing out the difference between our anomaly-based malware detection approach and the work by Forrest et al. [9], which aims to detect attacks against legitimate applications using models based on system call n-grams. This difference is subtle but important. Forrest builds one model for a given application. Only attacks against the specific, modeled application(s) are detected, and diversity is good because it makes mimicry attacks harder. In our case, we build one model for an entire set of applications (all benign programs). In our model, malware (the attacks in our context) are entirely new programs that we have not seen before. Diversity among benign programs is bad for our model, since it makes harder to build a tight model and easier for malware to mimic benign behavior.

Malware classification. Another research topic that is closely related to our work is that of classification of large sets of malware samples. Various models have been proposed, all focusing on system calls or on accesses to system resources. Lee and Mody performed classification of malware samples based on the similarity between sequences of system calls [18]. Bailey et al. [2] considered similarity between sets of accessed system resources, and Rieck et al. [23] considered various refinements by abstraction. Bayer et al. [3] used similarity between resource-based information flows for classification. All of these papers describe the classification task applied to large sets of malware (thousands or tens of thousands), and thus their results are representative. Yet, because their primary focus was on malware classification, it is not clear that the classification features that they derived are useful in malware detection. A classification feature (e.g., some particular resource accesses) might well distinguish botnet M_1 from botnet M_2, but it might not be able to distinguish botnet M_1 from a benign program B. Thus, our current work is orthogonal to malware-classification.

Access control and domain and type enforcement. Our system-centric access activity model is related to access-control mechanisms, and, in particular, to mandatory access control (MAC) systems. They both define acceptable uses of resources in a user-independent way via a central policy. There are numerous implementation of MAC systems, of which SELinux [20] is currently the most visible. Some MAC systems have been specifically designed to prevent malware from running in a system [24, 7], while others can enforce multi-level security policies. Based on this similarity, the system-centric model can be converted into a SELinux policy, for example, and our model-generation technique can be used as a practical tool to construct SELinux policies.

There is a fundamental distinction between MAC policies and our system-centric models. While a MAC policy necessarily enumerates all the programs and the program-specific rules, a system-centric model is more general in that it defines confidentiality and integrity rules for all programs. While it might appear that system-centric models are less restrictive, in our experimental evaluation, we observed a very good match between our models and real-life application executions. Additionally, MAC policy are often deployed to ensure the confidentiality and integrity of system files, at the cost of leaving user files poorly (if at all) secured and in need of additional mechanisms, such as the PinUP tool proposed by Enck et al. [8], which ties user files to particular applications. Our system-centric model covers system *and* user files, based on the observation that both system programs and applications satisfy some general ways in which they use OS resources.

6. CONCLUSIONS

In this paper, we performed an analysis of system call traces that were collected on ten hosts used by people in their daily work and pastime activities. Our results demonstrate that a detector that is based on simple, program-centric models, such as system-call sequences, faces significant challenges due to the diverse nature of system calls invoked by different applications. We also hope that the study of our collected system call traces could become a reference point for future work in this domain. In particular, our results could be used to call into question simple detection proposals that are evaluated on small data sets to determine false positive rates.

We also proposed a novel approach to capture the activities of benign programs and to detect certain types of malware (those tampering with binaries or settings of other applications or the OS). This approach takes a system-centric angle and models the way in which a broad set of benign applications interact with OS resources. More precisely, our approach builds an access activity model that captures permissible read and write operations on files and registry entries. Our experiments demonstrate that this model can discriminate well between malware and legitimate programs.

7. ACKNOWLEDGMENTS

With the partial support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme European Commission, Directorate-General Justice, Freedom and Security. This project (i-Code: Real-time Malicious Code Identification) has been funded with support from the European Commission. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. This work has also been supported in part by the European Commission through project IST-216026-WOMBAT funded under the 7th framework program, by the ONR under grant N000140911042 and by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537. We would also like to thank Secure Business Austria for their support for this research.

8. REFERENCES

[1] Anubis. http://anubis.seclab.tuwien.ac.at. 2008.

[2] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In C. Kruegel, R. Lippmann, and A. Clark, editors, *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197, Gold Goast, Australia, Sept. 2007. Springer-Verlag.

[3] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, San Diego, CA, USA, Feb. 2009.

[4] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *Proceedings of the 3rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer-Verlag, 2006.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, May 8–11, 2005. IEEE Computer Society.

[6] M. Christodorescu, C. Kruegel, and S. Jha. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 5–14, New York, NY, USA, 2007. ACM Press.

[7] M. Debbabi, M. Girard, L. Poulin, M. Salois, and N. Tawbi. Dynamic monitoring of malicious activity in software systems. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01)*, pages 1–10, Indianapolis, IN, USA, Mar. 2001.

[8] W. Enck, P. D. McDaniel, and T. Jaeger. Pinup: Pinning user files to known applications. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, pages 55–64, Anaheim, CA, USA, Dec. 2008. IEEE Computer Society.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P'96)*, pages 120–128. IEEE Computer Society Press, 1996.

[10] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2005.

[11] T. Holz, M. Engelberth, and F. Freiling. Learning more about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[12] J. John, A. Moshchuk, S. Gribble, and A. Krishnamurthy. Studying Spamming Botnets using Botlab. In *Usenix NSDI*, 2009.

[13] D. Kang, D. Fuller, and V. Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW)*, 2005.

[14] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, Vancouver, BC, Canada, August 2006.

[15] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, pages 351–366, Montréal, Canada, Aug. 2009. USENIX Association.

[16] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium (Security'05)*, Baltimore, MD, USA, August 2005.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, volume 3858 of *LNCS*, pages 207–226, Seattle, WA, September 2005. Springer-Verlag.

[18] T. Lee and J. J. Mody. Behavioral classification. In *Proceedings of the 15th Annual European Institute for Computer Antivirus Research Conference (EICAR'06)*, May 2006.

[19] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man, and Cybernetics (SMC) Workshop on Information Assurance*, pages 64–71, West Point, NY, June 2005. United States Military Academy.

[20] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[21] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (RAID'08)*, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] S. Mukkamala, A. Sung, D. Xu, and P. Chavez. Static analyzer for vicious executables (SAVE). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 326–334, Tucson, AZ, USA, Dec. 2004.

[23] K. Rieck, T. Holz, C. Willems, P. Düssel1, and P. Laskov. Learning and classification of malware behavior. In D. Zamboni, editor, *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'08)*, volume 5137 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2008.

[24] M. Salois and R. Charpentier. Dynamic detection of malicious code in COTS software. In *Proceedings of*

the Information Systems Technology Panel (IST) Symposium on Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", pages 16–1—16–13, Brussels, Belgium, Apr. 2000. NATO Research and Technology Organization.

[25] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P'01)*, pages 38–49, May 2001.

[26] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In C. Kruegel, R. Lippmann, and A. Clark, editors, *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, volume 4637 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[27] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[28] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS'02)*, pages 255–264, New York, NY, USA, 2002. ACM.

[29] J. Xu, A. H. Sung, P. Chavez, and S. Mukkamala. Polymorphic malicious executable scanner by API sequence analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems (HIS'04)*, pages 378–383, Kitakyushu, Japan, Dec. 2004. IEEE Computer Society.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, pages 116–127, New York, NY, USA, 2007. ACM.