Scalable, Behavior-Based Malware Clustering

Ulrich Bayer*,Paolo Milani Comparetti*,Clemens Hlauschek*,Christopher Kruegel§, and Engin Kirda¶

* Secure Systems Lab, Technical University Vienna
{pmilani,ulli,haku}@seclab.tuwien.ac.at

§ University of California, Santa Barbara
chris@cs.ucsb.edu

¶ Institute Eurecom, Sophia Antipolis
kirda@eurecom.fr

Abstract

Anti-malware companies receive thousands of malware samples every day. To process this large quantity, a number of automated analysis tools were developed. These tools execute a malicious program in a controlled environment and produce reports that summarize the program's actions. Of course, the problem of analyzing the reports still remains. Recently, researchers have started to explore automated clustering techniques that help to identify samples that exhibit similar behavior. This allows an analyst to discard reports of samples that have been seen before, while focusing on novel, interesting threats. Unfortunately, previous techniques do not scale well and frequently fail to generalize the observed activity well enough to recognize related malware.

In this paper, we propose a scalable clustering approach to identify and group malware samples that exhibit similar behavior. For this, we first perform dynamic analysis to obtain the execution traces of malware programs. These execution traces are then generalized into behavioral profiles, which characterize the activity of a program in more abstract terms. The profiles serve as input to an efficient clustering algorithm that allows us to handle sample sets that are an order of magnitude larger than previous approaches. We have applied our system to real-world malware collections. The results demonstrate that our technique is able to recognize and group malware programs that behave similarly, achieving a better precision than previous approaches. To underline the scalability of the system, we clustered a set of more than 75 thousand samples in less than three hours.

1 Introduction

One of the major threats on the Internet today is malicious software, often referred to as malware. In fact, most Internet security problems have malware as their underlying root cause. For example, botnets are commonly used to send spam and host phishing web sites that are more difficult to track down and blacklist. Malware comes in a wide range of forms and variations, such as viruses, worms, botnets, rootkits, Trojan horses, and denial of service tools. To spread, malware exploits software vulnerabilities in browsers and operating systems, or uses social engineering techniques to trick users into running the malicious code.

An anti-malware company typically receives thousands of new malware samples every day. These samples are submitted by users who have found suspicious code on their systems, by other anti-malware companies that share their samples, and by organizations (e.g., MWCollect [5], ShadowServer [7], VirusTotal [9]) that use technologies such as honeypots [42] to collect malware. For each sample, it is important to understand the actions that this program can perform. This is necessary to determine the type and severity of the threat that the malware constitutes. Also, this information is valuable to create detection signatures and removal procedures. In some cases, the sample may turn out to be harmless. Furthermore, in many cases, the malware may turn out to be a variant of a well-known malware instance. In fact, although the malware may remain the same, its signature might change just because the malware author is using a simple obfuscation or polymorphism technique [34, 38, 44].

Because of the growing need for automated techniques to examine malware, dynamic malware analysis tools such as CWSandbox [3], Norman Sandbox [6], and ANUBIS [1, 14] have increased in popularity. These systems execute the malware sample in a controlled environment and mon-

itor its actions. Based on the execution traces, reports are generated that aim to support an analyst in reaching a conclusion about the type and severity of the threat imposed by a malware sample. However, while automating the analysis of the behavior of a single malware sample is a first step, it is not sufficient. The reason is that the analyst is now facing thousands of reports every day that need to be examined. Thus, there is a need to prioritize these reports and guide an analyst in the selection of those samples that require most attention. One approach to process reports is to cluster them into sets of malware that exhibit similar behavior. The ability to automatically and effectively cluster analyzed malware samples into families with similar characteristics is beneficial for the following reasons: First, every time a new malware sample is found in the wild, an analyst can quickly determine whether it is a new malware instance or a variant of a well-known family. Moreover, given sets of malware samples that belong to different malware families, it becomes significantly easier to derive generalized signatures, implement removal procedures, and create new mitigation strategies that work for a whole class of programs.

Grouping individual malware samples into malware families is not a new idea, and clustering and classification methods have already been proposed previously [13, 26, 30, 31, 28]. These approaches, however, generally do not scale well and are too slow for the size of malware sets that anti-malware companies are confronted with. Moreover, these techniques are imprecise, either because their notion of similarity is not tied to a program's actual behavior or because it does not capture a program's behavior well enough. Imprecise in this context either means putting samples of different types into the same group or failing to recognize similar malware programs.

In this paper, we present a novel clustering technique that scales well and produces more precise results than previous approaches. Our technique is based on a dynamic analysis system that monitors the execution of a malware sample in a controlled environment. Unlike many previous systems that operate directly on low-level data such as system call traces, we enrich and generalize the collected data and summarize the behavior of a malware sample in a behavioral profile. These profiles express malware behavior in terms of operating system (OS) objects and OS operations. Moreover, profiles capture a more detailed view of network activity and the ways in which a malware program uses input from the environment. This allows our system to recognize similar behaviors among samples whose low-level traces appear very different. Finally, we cluster the analyzed samples according to their behavioral profile. We employ a scalable clustering algorithm that avoids calculating n^2 distances between all pairs of n samples, and thus, is suitable for clustering large, real-world malware collections. To summarize, the contributions of this paper are as follows:

- We present a novel, precise approach to capture a malware program's behavior. To this end, we monitor the execution of a program and create its behavioral profile by abstracting system calls, their dependences, and the network activities to a generalized representation consisting of OS objects and OS operations.

- We present an efficient and fast algorithm for clustering large sets of malware samples that avoids calculating n^2 distances between all pairs of n samples, and thus, is suitable for clustering large, real-world malware collections.

- We have evaluated our system on large, real-world data sets. Our experiments demonstrate that our technique achieves more precise clustering results than previous approaches and scales to tens of thousands of malware samples.

2 System Overview

The goal of our system is to cluster large collections of malware-samples based on their behavior. That is, we want to find a partitioning of a given set of malware programs so that subsets share some common traits. As illustrated in Figure 1, clustering malware samples is a multi-step process. It consists of an initial, dynamic malware analysis phase, a subsequent extraction of behavioral profiles, and a final clustering phase.

Dynamic Analysis. The first step in the clustering process is the automated analysis of malware samples. For this purpose, we have extended ANUBIS, our system for automated, dynamic malware analysis [14]. This system is based on Qemu [15], a whole-system emulator for PCs and the Intel x86 architecture. The analysis system works by executing binaries in the emulated environment, producing a trace of the system calls that this binary invokes.

We first extended ANUBIS with taint tracking. Similar to tainting systems in previous work [23, 37, 39], we attach (taint) labels to certain interesting bytes in memory and propagate these labels whenever they are copied or otherwise manipulated. Our taint tracking system builds upon the taint tracking implementation used in a previous prototype [37]. While the propagation of taint labels works the same as in [37], the tainting systems differ in their use of taint sources. In our system, system calls serve as taint sources. More precisely, we taint the out-arguments and return values of all system calls. At the same time, we check whether any in-argument of a system call is tainted. The goal is to identify how the program uses information that it obtains from the operating system.

While the idea of taint tracking itself is not new, we leverage this information to obtain a number of novel, im-

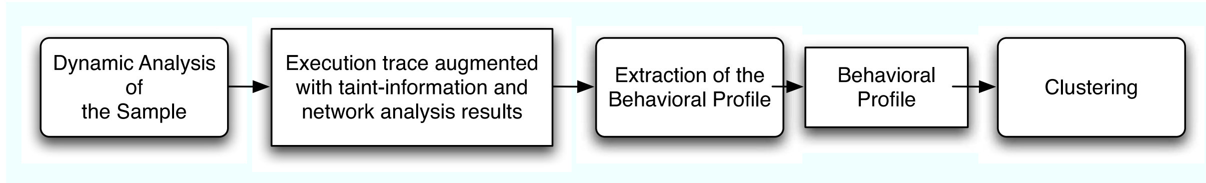

Figure 1. System overview.

portant features that better capture the behavior of a malware program. For example, we can observe when a program uses the return value of the `GetDate` system call in a subsequent `CreateFile` call. This allows us to determine that a file name depends on the current date and changes with every malware execution. As a result, the filename is generalized appropriately. Furthermore, we taint the entire code of the executable. This allows us to uncover cases in which a program reads its own code segment. This is helpful to detect important propagation patterns, such as a worm sending itself over the network or a Trojan horse copying itself into the Windows system directory. Finally, we record program control flow decisions that are based on tainted data. This allows us to identify similarities between programs that perform the same date checks or that attempt to shut down the same anti-virus software.

To address the problem that a program's network activity is not sufficiently captured by system call traces, we have built a network analysis component that operates on the network traffic itself. The problem is that on the system-call level, all network activities are performed through calls to a single, native API function called `NtDeviceIoControlFile` - only differing in their arguments. Ideally, we would like to know what emails are sent, what HTTP downloads are performed, what IRC conversations take place, etc. To this end, our network analysis component leverages Bro [40] and makes use of its capabilities to recognize and parse application-level protocols (such as HTTP, SMTP, and IRC).

The output of the analysis step is an execution trace that is augmented with taint information. This trace lists all system calls together with their argument values. Moreover, it provides taint information for each argument. This taint information allows us to connect the return values (and out-arguments) of one system call with the in-arguments of subsequent calls.

Behavioral Profile. In this step, we process the execution traces provided by the previous step. More precisely, for each sample, we extract a behavioral profile that accurately describes the runtime activity of the binary and serves as input to our clustering algorithm.

Unlike existing systems [26, 31], our clustering algorithm does not operate directly on system calls. The reason is that system call traces can vary significantly, even between programs that exhibit the same behavior. For example, consider the different ways to read from a file: Program A might read 256 bytes at once, while program B calls `read` 256 times, reading 1 byte with each call. Moreover, it is easily possible to interleave the read calls with other, independent system calls so that the system call trace changes. For this reason, we abstract system call traces into a set of operating system objects, together with a set of operations (such as read, write, create) that were performed on these objects.

An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. For example, our behavioral profile might include the file object `C:\Windows` and its accompanying operation `query_directory`. An OS operation is a generalization of a system call that unifies different system calls with similar semantics but different function signatures (e.g., the system calls `NtCreateProcessEx` and `NtCreateProcess` both map to the same operation).

Based on the information that the tainting system provides, we infer dependences between OS objects. Copying a file, for example, is represented as a dependence between the source file OS object and the destination file object. Dependency information implicitly captures the order of certain operations. This is important, because we do not explicitly consider the order of OS operations that are performed on a specific OS object. The reason is that a behavior profile should not rely on the order in which unrelated operations are executed. Moreover, dependences help to determine resource names that are derived from data sources whose values change between execution traces (such as random values or the current time). This information allows us to generalize the corresponding OS object names.

The output of this step is an abstraction of the program's execution trace that contains information about the OS objects that the program operates on, as well as of the type of operations and dependences. These abstractions are called a *behavioral profile*.

Scalable Clustering. In this step, we cluster a set of behavioral profiles such that samples that exhibit similar behavior are combined in the same cluster. Given the rapidly increasing number of malware programs, it is clear that one of the

most important requirements for a clustering algorithm is scalability. It must be possible to cluster a large amount of malware, such as a hundred thousand samples, in a reasonable time. Most clustering methods require the computation of the distances between all pairs of points, which invariably results in a computational complexity of $O(n^2)$. This might lead to systems that take three hours to process 400 samples [31].

In this paper, we propose to efficiently solve the clustering problem using an approximate, probabilistic approach. Our clustering algorithm is based on locality sensitive hashing (LSH), which was introduced by Indyk and Motwani [29]. LSH provides an efficient (sublinear) solution to the approximate nearest neighbor problem (ϵ-NNS). Clustering is one of the main applications of this technique: LSH can be used to perform an approximate clustering while computing only a small fraction of the $n^2/2$ distances between pairs of points. Leveraging LSH clustering, we are able to compute an approximate, single-linkage hierarchical clustering for a data set of more than 75,000 samples in less than three hours.

3 Dynamic Analysis

Dynamic malware analysis systems have become increasingly popular because they deliver good analysis results even in case of obfuscated or self-modifying code and analysis resistance techniques [34, 38, 44]. Since meaningful analysis results are a prerequisite for good clustering results, we have chosen to further extend ANUBIS, our existing, dynamic analysis system [14]. More precisely, we have added support to track dependences between operations on system code objects, as well as support to analyze control flow decisions that involve tainted data operands, and we have improved the network analysis.

3.1 System Call Dependences

Data tainting is a well-known technique for tracking information flows in a whole-system emulator. In this work, we are leveraging the tainting approach to capture the dependences between system calls. As noted by Christodorescu et al. [21], system call dependences provide valuable insights into the behavior of an application. For example, knowledge about dependences allows one to see when a program searches for files with a specific filename pattern and then opens all files that were found. In our analysis, system call dependences are used in the following contexts:

- *Generalization of execution traces*: An execution trace inherently includes many execution-specific events and names (filenames, host names). These execution--specific tokens change every time the binary is executed. Based on dependence analysis, such execution-specific artifacts can be recognized. For example, knowing that a filename depends on the current time helps to remove the filename as a characteristic for a program's behavior. Also, tainting the return value of the Windows function `GetTempFileName` puts us into a position where we can identify temporary file names.
- *Copy Operations*: Tainting allows us to recognize data movements, such as the case when data is copied. This allows us to determine the malware's propagation vector. For example, we see when the malware copies itself to the Windows system directory or sends a copy of its code over the network.

Taint Sources. Although our behavioral profile is primarily based on system calls and their dependences, we are not focusing on the native API interface[1] alone. Instead, we also include several Windows API functions. This is different from previous systems, which either operate on the system call interface [25, 39] or perform whole-system taint analysis [46]. In the latter case, taint sources are typically devices such as a network card or the keyboard.

The Windows API is a large collection of user mode library routines, which in turn invoke native API functions when necessary. Considering the Windows API is important for several reasons: First, some functionality is managed and provided exclusively by user-mode portions of the operating system. That is, no calls to native API functions are performed. Among other things, this is true for the random number generator, the path-related Windows API functions (e.g. `GetTempFileName`, `GetTempPath`), and DLL-management functions (e.g., `GetProcAddress`). Second, there are Windows API functions that have semantically-equivalent native API functions, but, because of performance reasons, have been implemented in a way that does not require invoking the appropriate system service. An important instance are the time-related Windows API function, such as `GetTickCount` or `GetSystemTime`. These functions do not invoke a system call but work by reading from a special page in the virtual address space. This page is always mapped read-only to a fixed address in the user-mode, virtual address space of a processes. The kernel maps the same page with write access and updates the time-related information in the timer interrupt handler.

Memory-Mapped Files. Memory-mapped files, officially termed section objects in Windows NT, pose a special challenge to the analysis system. When a process maps a file into its virtual address space, reading and writing to the file is possible by simply reading and writing to the mapped

[1] In Windows NT, the operating system call interface is termed native API.

memory region. These read and write operations do not result in any system calls. Thus, in current analysis systems such as [1, 3, 6, 8], all read and write activity to a memory-mapped file will go unnoticed. However, it is crucial to add support for Windows section objects to obtain a complete view of the operations of a program.

To keep track of indirect write operations to a file, we modified the function that is responsible for writing to the physical memory in our emulation. Whenever the process writes to memory, we check whether the address is in a memory-mapped area. If this is the case, we report this operation as a write to the corresponding, mapped file. Tracking read operations from a memory-mapped file requires tainting the appropriate region in memory whenever a program maps a file into its address space. However, Windows does not load the contents of the file into physical memory at the time of the section creation. Instead, Windows defers loading (portions of) the file into the physical memory until the time when a virtual address in this region is accessed. Because our tainting system is only able to taint values in the physical memory and the CPU registers, we have to wait until the file is eventually mapped into physical memory before we can taint it. We solve this problem by monitoring all invocations of the page fault handler. When the page fault handler brings in a page that is part of a memory-mapped region, we taint the page after the handler returns.

3.2 Control Flow Dependences

Taint information is useful to track dependences between system calls. However, it is also interesting to analyze how tainted data is used by the program itself. More specifically, we would like to identify the control flow decisions that involve data that a process has obtained via system calls. Information about such control flow decisions reveals many interesting aspects about a program. For example, it allows us to discover which processes a malware sample potentially wishes to terminate by observing all comparisons that take place as the program iterates over the list of running processes. Since the list of running processes has to be retrieved by means of system calls, the process names that this system call returns are tainted. Hence, we are aware of all comparisons that involve the retrieved process list as argument.

On the x86 architecture, many different assembler instructions for comparing two values exist. Fortunately, inside the Qemu intermediate language, all of these different compare instructions (such as `CMP`, `CMPS`, `SCAS`) map to the same intermediate language construct. Thus, we can easily handle all compare instructions by building our analysis on top of Qemu's intermediate language. The REP instruction prefix is correctly handled as a consecutive execution of the same compare instruction. However, as explained in the following paragraphs, consecutive compare instructions are merged into a single one during our analysis.

To detect comparisons with tainted values, the following extensions were necessary:

Representation. In the Qemu intermediate language a compare instruction has two operands with a size of either one, two, or four bytes. For each comparison, we can examine the taint labels that are attached to the bytes of both arguments (if there are any). When a taint label is present, we can determine the system call and the exact argument where the corresponding data byte was retrieved from. Based on this information, we can also determine the original data type of a tainted byte. This is possible because the Windows native API header files declare all system calls and the types of their arguments. Based on the data type, we consider the operand of a comparison as signed/unsigned integer or as a character string. Knowing the data type also allows us to pinpoint the exact member of a structure. For example, we do not only see that a value '6' is compared with `struct_SYSTEMTIME`, but we can also determine that the value is compared to the struct's `wDay` member.

One problem arises when more complex data structures (e.g., structs, strings, etc.) are involved in a comparison. In this case, we observe several, consecutive `cmp` instructions that operate on a few bytes of the data structure. To handle such cases, consecutive compares on successive labels are merged. Also, when comparing for equality, the comparison terminates as soon as the first differing byte is encountered. In these cases, we cannot see the complete values that the program actually compares. However, the complete data structure exists in the computer's main memory. Thus, when string comparisons are involved, we try to recover the entire string by reading it from the main memory. To this end, we assume that the operand being compared marks the beginning of the string and check until a null byte is found.

There are two types of comparisons that we record as part of an execution trace: A comparison of a labeled value (i.e., at least a single byte of an operand is tainted) with an unlabeled value (called a *label-value comparison*) and a comparison of a labeled value with another labeled value (called a *label-label comparison*). In both cases, we do not output the concrete values of the labeled (tainted) data but the source where this data originates from. More precisely, for tainted data, we record the function name, the function argument, and, if applicable, the name of the structure that holds the data together with the member name. This allows us to identify which inputs to the program are used for comparisons. In case of a label-value comparison, we also learn the concrete value that the program checks for.

Filtering. An important part of analyzing control flow dependences is to filter out the irrelevant ones. Compare instructions occur very frequently, and a raw execution

trace typically contains millions of compares with tainted operands. To focus on compare instructions that are done by the actual malware program, we discard those that were executed on behalf of (user-mode) Windows API functions. In this way, we ignore comparisons that do not represent the direct intent of the program's author, but that are present as a result of standard Windows behavior. The only exception to this rule are a number of Windows API functions that are used for comparing more complex data types, such as strings or dates. Obviously, the comparisons that occur inside these API functions are the direct consequence of the programmer's intent. For this reason, we catch all comparisons that take place inside `strcmp`, for example.

3.3 Network Analysis

The network activities of a malware sample provide one of the most important and characterizing insights into a sample's behavior. Thus, the analysis of a sample's network activity plays an important role in our approach.

Environment. A successful network analysis requires that a sample is able to perform the network activities that it has been programmed to do. Dynamic analysis cannot observe email activity of a program when it fails to establish a TCP connection to a mail server. Thus, as a first step, we run the sample in an environment that permits a sample to perform its built-in network activities.

To create the environment for malware execution, we allow an analyzed sample to download files via HTTP and to contact IRC servers directly on the Internet. All other traffic is rerouted to a specially-prepared server, called the victim machine, which has been configured to accept incoming connections on a number of ports that are frequently used by malware programs. For example, the victim machine runs its own SMTP server that answers all SMTP requests (but does not deliver any emails). Moreover, we have set up nepenthes [11] - a honeypot system that emulates known vulnerabilities of popular services. Of course, we are not using the nepenthes server as a honeypot system in the usual sense, i.e., as a way to gain new malware samples. Instead, we have deployed nepenthes only for having a basic service listening on ports that are frequently used for spreading (such as the Windows Samba ports).

Analysis. The goal of our network analysis is to extract high-level semantic operations from the low-level socket system calls. For example, instead of reporting that a TCP connection was established, together with the amount of bytes that were exchanged, we aim to report that an HTTP GET request was sent to download the file "foo.bar." We have chosen to build our analysis on top of the packets that are sent and received at the network level. This is easier and more comprehensive than attempting to infer all information from the arguments of the `DeviceIoControlFile` system call, which serves as a funnel for all network-related activity on Windows. To capture network traffic, we have modified our system emulator's network card to simply dump all packets to a log file in PCAP-format. This way, we have a wide range of standard network analysis tools at our disposal to aid us in our analysis efforts. Also, by parsing network packets and parsing application protocols, such as HTTP, we are able to identify network activity on a higher level of abstraction. We use Bro [40] for our analysis, a system that has built-in support for identifying and parsing HTTP, IRC, SMTP, and FTP protocols. For these protocols, we extract information such as names of downloaded files, names of IRC channels, or mail subjects.

4 Behavioral Profile

When the dynamic analysis step finishes processing a sample, the next task is to transform the augmented execution trace into a behavioral profile. As mentioned previously, a behavioral profile captures the operations of a program at a higher level of abstraction. To this end, we model a sample's behavior in the form of OS objects, operations that are carried out on these objects, dependences between OS objects and comparisons between OS objects. More formally, a behavioral profile P is defined as an 8-tuple

$$P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$$

where O is the set of all OS objects, OP is the set of all OS operations, $\Gamma \subseteq (O \times OP)$ is a relation assigning one or several operations to each object, and $\Delta \subseteq ((O \times OP) \times (O \times OP))$ represents the set of dependences. CV is the set of all compare operations of type label-value, while CL is the set of all compare operation of type label-label. $\Theta_{CmpValue} \subseteq (CV \times O)$ is a relation assigning label-value compare operations to an OS object. $\Theta_{CmpLabel} \subseteq (CL \times O \times O)$ is a relation assigning label-label compare operations to the two appropriate OS objects.

OS Objects. An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. Formally, an OS object is a tuple of the following form:

```
OS Object ::= (type, object-name)
type ::= file|registry|process|job|
         network|thread|section|
         driver|sync|service|random|
         time|info
```

That is, an OS object has a name and a type that together uniquely identify the object in the operating system. The 'file' type covers file, named pipe, and mailslot resources, 'registry' consists of registry keys, 'process' includes processes, and 'job' denotes Windows NT jobs, which allow

for combining individual processes into a group. The ‘network‘ category describes network objects, ‘thread‘ represents thread activity, ‘section‘ refers to memory-mapped files, and ‘driver‘ captures the loading and unloading of Windows device drivers. The type ‘sync‘ abstracts all synchronization activities, such as operations on semaphores and mutexes, and ‘service‘ contains objects that represent Windows services. The type ‘random‘ includes several sources of randomness, each of which can be used by a program to generate a random number. The type ‘time‘ consists of time sources, and ‘info‘ contains only two objects. One is the object *info-executable*, which represents the loaded executable. The other one is *info-general*, which represents information such as pathnames of the windows system directory and the temporary directory.

OS Object		*OS Operation*	
Type	*Name*	*Name*	*Attributes*
net	http_server	contact	‘www.gson.com‘,‘80‘
net	http_request	get	‘/down/s.htm‘
net	dns_resolver	query	‘Type A‘,‘mx.gmx.net‘
net	port_listener	listen	‘TCP‘,‘6777‘
net	smtp_attmts	send	‘fpw.exe‘
net	smtp_content	send	‘Test yep.‘
net	smtp_subjs	send	‘Hi‘

Table 1. Example network OS objects.

To create OS objects, we search the execution trace for all system calls that produce new OS resources. For example, the function `NtCreateFile` creates new files. For each such system call, we extract the object name from the argument list, deduce the object type from the type of the system call, and then create a new OS object. Typically, native API calls have a parameter, named `ObjectAttributes`, that can be directly translated to an object name. In a few cases, it is more difficult to determine the object name. For example, `NtCreateProcess` expects a handle argument that points to a section object (a memory-mapped file), instead of an argument that specifies the filename of the executable. To address this problem, we have extended our system call logger to resolve handles to NT kernel objects and provide this information.

Since network activities are not directly represented in the execution trace, we rely on the network analysis component for extracting the virtual network OS objects. Depending on the type of network traffic observed, we create different kinds of network objects. Table 1 lists some example network objects, together with their corresponding operations.

OS Operations. An OS operation is a generalization of a system call. Formally, an operation is defined as:

```
OS operation ::= (operation-name,
                  operation-attributes?,
                  successful?)
```

An operation must have a name, it may have one or more attributes that provide additional information about the operation, and it may have a value describing whether the operation was successful.

We map system calls to OS operations with the intent of abstracting from API-specific details. For example, we ignore whether a process is created by means of `NtCreateProcess` or `NtCreateProcessEx` and unify these two system calls into the single OS operation `create`. Our mapping function only considers the most essential system calls, such as functions for reading, writing, and creating operating system objects. This allows us to abstract from many unimportant details. For example, we ignore all functions relating to NT's Local Procedure Call functionality, because this is an undocumented feature that is not available via the Windows API. Currently, we map 130 native API and Windows API functions to 55 OS operations.

System calls that operate on a resource typically have a (handle) parameter that references the target resource. This is necessary for the OS to know the resource to which an operation should be applied. We make use of these handles to map operations to the appropriate OS objects. There are few cases where a function that logically constitutes an operation on an object does not have a handle parameter that specifies this object. The `NtQueryAttributesFile` function, for example, uses a filename instead of a handle to indicate the file object that it works on. After assigning operations to OS objects, our implementation stores all of an object's operations in a set. As a consequence, the order of OS operations is irrelevant. This is important, because it is very easy to reorder system calls on a resource without changing the semantics of a program. Thus, we are able to generalize our behavioral profile by neglecting the order of operations. System call dependences are used to capture the order between those OS operations where the actual order is implied by a data dependence. Moreover, the number of operations on a certain resource does not matter in our system. This sacrifices some precision, but makes the behavioral profile more general, and thus, harder to evade by introducing superfluous operations.

Example of a Behavioral Profile. Figure 2 shows an example of a behavioral profile. Note that although this example is shown in C code, our profile extraction algorithm works on execution traces. This example shows code that copies the file `C:\sample.exe` to `C:\Windows\sample.exe` by memory-mapping the source file. As one can see, independent of the number of times the write operation in Line 14 is executed, the write operation appears only once in the corresponding behavioral profile. It is also noteworthy that the `NtQuery-`

```
0: // open the source-file as a memory-mapped file
1:  HANDLE src = NtOpenFile("C:\sample.exe");
2:  HANDLE sectionHandle = NtCreateSection(src);
3:  void *base = NtMapViewOfSection(sectionHandle);
4:
5:  // don't overwrite the target
6:  if (NtQueryAttributesFile("C:\Windows\sample.exe") !
=
7:     STATUS_OBJECT_NAME_NOT_FOUND)
8:   exit(1);
9:  // open the target
10:  target = NtCreateFile("C:\Windows\sample.exe");
11:
12:  void *p = base;
13:  while(p < base + fileLen) {
14:    NtWriteFile(target, p++);
15:  }
```

Pseudo Code Fragment

```
File|C:\sample.exe
        open:1
Section|C:\sample.exe
        open:1, map:1, mem_read: 1
File|C:\Windows\sample.exe
        query_file:0, create:1, write:1
Section|C:\sample.exe -> File|C:\Windows\sample.exe
        mem_read – write: (fileLen)
```

Behavioral Profile

Figure 2. Example Behavioral Profile

`AttributesFile` operation in Line 6 is assigned to the object `C:\Windows\sample.exe`, although it does not use a handle argument to reference its OS object.

Object Dependences. We abstract dependences between system calls to dependences between OS objects. While a system call dependence is a dependence relation between two system call instances, an OS object dependence is a dependence between two OS objects and their operations. For each existing system call dependence, we first check whether the two involved system calls map to OS operations. If this is the case, we introduce an object dependence between the corresponding OS objects. The behavioral profile shown in Figure 2 contains a dependence between the section OS object of the source file and the file object of the destination file. This dependency reflects the fact that data from the source was copied to the destination file.

Due to the fact that all our object dependences originate from system call dependences, we would lack network-related dependences. As explained previously, this is because the extraction of network OS objects is a separate process that is mostly based on the captured network traffic. To address this problem, we have partly reverse-engineered the semantics of the `NtDeviceIoControlFile` function. `NtDeviceIoControlFile` is a universal interface that allows user-mode programs to communicate with device drivers, including the network stack. It is possible to recognize network-related invocations of `NtDeviceIoControlFile` by checking two of its arguments, the handle argument as well as its IO control code. In addition, `NtDeviceIoControlFile` has an input buffer and an output buffer argument for transferring data. For each call to `NtDeviceIoControlFile` that represents network activity, we insert an artificial system call into the execution trace that represents a decoded form of the original call. In particular, we have to decode the buffer arguments. In the case of network activities, `NtDeviceIoControlFile`'s buffer arguments contain pointers to network-specific structs. There are four different artificial system calls:

```
AfdSend(SocketHandle h, char *buffer)
AfdReceive(SocketHandle h, char *buffer)
AfdBind(SocketHandle h, short localPort)
AfdConnect(char *foreignAddress,
           short foreignPort)
```

We insert `AfdSend` when we determine that a process calls `NtDeviceIoControlFile` to send data. Analogously, we insert `AfdReceive` when data is received, `AfdBind` when a socket is bound to a specific port number, and `AfdConnect`, when a TCP connection is established. The arguments of the four artificial calls reflect the

taint information of their corresponding system calls. The SocketHandle parameter allows us to attribute the individual invocations to the appropriate network connection.

Based on our representation of objects and their dependences, it is straightforward to find execution-specific artifacts. For example, we recognize random or temporary filenames by checking whether there is a dependence between a file object and a random source. If this is the case, we do not want to keep the actual object in the profile, since it is different for each execution. Thus, we replace the concrete object name with a placeholder token that indicates the source of the object name (such as TEMPORARY for a temporary filename). Moreover, we append the value of a counter that is increased by one until the object name becomes unique in this profile. When comparing two behavioral profiles that both contain objects with temporary filenames, it is possible to match these two objects. However, we have to avoid that an object a_1 of profile A matches with object b_1 of profile B, when the operations associated with the object make it actually more similar to object b_2 of profile B. We address this problem by calculating a checksum over all OS operations, using the resulting value as part of the new object name. That is, execution-specific names are replaced with a new name of the form `<token><checksum><counter>`. The checksum guarantees that only objects with the same OS operations will receive the same name in two different profiles, and consequently match.

Control Flow Dependences. Control flow dependences are translated into comparisons between OS objects. Depending on the type of the comparison, a control flow dependency is associated with either one or two OS objects. A label-label comparison involves two OS objects (one for each operand), while a label-value comparison involves only a single one. To find the appropriate OS resource, the labels are used. That is, we search for the OS operation that created a particular label. Then, we search for the object that the operation is associated with.

5 Scalable Clustering

Clustering a set of n points in a high-dimensional space is a computationally expensive task. Most clustering algorithms require to compute the distances between all pairs of points in the set. In this case, computational complexity is at least $O(n^2)$ evaluations of the distance function, which is unacceptable for large data sets.

There exist algorithms, such as the k-means algorithm (Lloyd's algorithm) [36], that only compute the distance from the n points to k cluster centers, and repeat this computation for each of i iterations required to converge to a local optimum. The computational complexity is, therefore, $O(nki)$ evaluations of the distance functions. Unfortunately, there are no guarantees that the value of i is small (in fact, the number of iterations is super-polynomial in n in the worst-case [10]). Furthermore, the accuracy of k-means is limited (the solution is only locally optimal), and the number of clusters k has to be specified *a priori*.

In this work, we employ locality sensitive hashing (LSH), introduced by Indyk and Motwani [29], to compute an approximate clustering of our data set that requires significantly less than n^2 distance computations. Our clustering algorithm takes as input the set of malware samples $A = a_1, .., a_n$, where $a_i \subseteq F$, and F is the set of all features. LSH algorithms have been proposed for metric spaces where the similarity between two points is defined by one of a few simple functions, such as Jaccard index [16], or cosine similarity [20]. In this work, we employ the Jaccard index as a measure of similarity between two samples a and b, defined as $J(a, b) = |a \cap b|/|a \cup b|$. A similarity value of $J(a, b) = 1$ indicates that two samples have identical behavior. While other, more complex similarity functions, such as normalized compression distance [13], may be more accurate measures of the similarity between behavioral profiles, choosing this simple set similarity measure allows our clustering approach to leverage LSH and to scale up to the size of real-world malware collections.

In the following Section 5.1, we explain how we map a behavioral profile into a set of features that are suitable for LSH. Section 5.2 briefly explains the LSH algorithm. In Section 5.3, we discuss how we can use the output of the LSH algorithm to compute an approximate, hierarchical clustering of our malware sample set. Finally, in Section 5.4, we discuss the asymptotic performance of our approach.

5.1 Transforming Profiles into Features Sets

Before we can run the clustering algorithm, we have to transform each behavioral profile into a feature set. Informally, a feature is a behavioral characteristic of a sample, such as "file xy was created." We use the following algorithm to transform a behavioral profile $P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$ into a set of features: For each object $o_i \in O$, and for each assigned $op_j \in OP|(o_i, a) \in \Gamma$, create a feature:

$$f_{ij} = \text{"}op|\text{"} + name(o_i) + \text{"}|\text{"} + name(op_j)$$

where $name()$ is a function that returns the name of an OS object, operation, or comparison as string, quotes (") denote a literal string, and + concatenates two strings. Moreover, for each dependence $\delta_i \in \Delta = ((o_{i1}, op_{i1}), (o_{i2}, op_{i2}))$, we create a feature:

$$f_i = \text{"}dep|\text{"} + name(o_{i1}) + \text{"}|\text{"} + name(op_{i1}) +$$

$$+" \rightarrow " + name(o_{i2}) + "|" + name(o_{i2})$$

For each label-value comparison $\theta_i \in \Theta_{CmpValue} = (cmp, o)$, we create a feature:

$$f_i = "cmp_value|" + name(o) + "|" + name(cmp)$$

For each label-label comparison $\theta_i \in \Theta_{CmpLabel} = (cmp, o_1, o_2)$, we create a feature:

$$f_i = "cmp_label|" + name(o_1) +$$
$$+" \rightarrow " + name(o_2) + "|" + name(cmp)$$

The output of this transformation step is a set of features that captures the behavioral characteristics of a sample in a form that is suitable for the clustering algorithm. We then discard all features of a sample that are unique with regards to all other samples in the data set. That is, we do not consider a feature for clustering when it does not occur in at least one other sample's feature set. This is because a unique feature of a sample does not help us to find other samples that behave similarly (i.e., the information gain of this feature is very low). Moreover, our experiments show that the robustness of our clustering to the selection of the threshold t improves when we discard such unique outliers.

5.2 Locality Sensitive Hashing (LSH)

The idea behind locality sensitive hashing is to hash a set A of points in such a way that near (or similar) points have a much higher collision probability than points that are distant. We achieve this by employing a family H of hash functions such that $Pr[h(a) = h(b)] = similarity(a, b)$, for a, b points in our feature space, and h chosen uniformly at random from H. By defining the locality sensitive hash of a as $lsh(a) = h_1(a), .., h_k(a)$, with k hash functions chosen independently and uniformly at random from H, we then have $Pr[lsh(a) = lsh(b)] = similarity(a, b)^k$.

In the case of sets for which the Jaccard index is used as similarity measure, a family of hash functions H with the desired property has been introduced in [16]. A hash in H imposes a random order on the set of all features. The hash value for a feature set a is then determined by the index of the smallest element of a according to this order. Since it is inefficient to generate truly random permutations, random linear functions in the form $h(x) = c_1 x + c_2 \mod P$ are used instead [27], with P a prime number larger than the total number of features in F.

Given a similarity threshold t, we employ the LSH algorithm to compute a set S which approximates the set T of all near pairs in $A \times A$, defined as $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Given the threshold t, we first choose the number k of hash functions in each LSH hash, and the number of iterations l. Furthermore, we initialize the set S of candidate near pairs to the empty set. Then, for each iteration, the following steps are performed:

- choose k hash functions $h_1, .., h_k$ at random from H
- compute $lsh(a) = h_1(a), .., h_k(a)$ for each $a \in A$
- sort the samples based on their LSH hashes
- add all pairs of samples with identical LSH hashes to S

LSH Parameters. For a given similarity threshold t, we must choose appropriate values of k and l. For a pair $p = (a, b)$ such that $similarity(a, b) = v$, we have $Pr[p \in S] = 1 - (1 - v^k)^l = g(v)$. Thus, given t, we can choose k and l such that $g(t)$ is close to 1 and $g(t/(1 + \epsilon))$ is small, for any $\epsilon > 0$. That is, t is the only parameter that needs to be chosen. For a threshold value of $t = 0.7$ we selected $k = 10$ and $l = 90$.

5.3 Hierarchical Clustering

The result of the locality sensitive hashing step is a set S, which is an approximation of the true set of all near pairs $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Because LSH only computes an approximation, S might contain pairs of samples that are not similar. To remove those, for each pair a, b in S, we compute the similarity $J(a, b)$ and discard the pair if $J(a, b) < t$. Then, we sort the remaining pairs by similarity. This allows to produce an approximate, single-linkage hierarchical clustering [35] of A, up to the threshold value t. Single-linkage clustering allows us to simply iterate over the sorted list of pairs to produce an agglomerative clustering. We stop the clustering when there are no more near pairs left.

In some cases, one would like to continue the hierarchical clustering process until all elements are merged into a single cluster. However, all subsequent clustering steps would require to merge two clusters that have a similarity value below t. Of course, this information is not readily available. The reason is that the LSH algorithm avoids the calculation of distances between elements that have a similarity value below t. To solve this problem and to obtain an exhaustive, hierarchical clustering, we use the following technique: We choose a representative element for each cluster, calculate the distances between all representatives, and then perform exact, hierarchical clustering between these elements. We create the representative element r of a cluster C by adding all features to r_C that exist in at least half of all the feature sets in C. Of course, exact hierarchical clustering has a complexity of $O(n^2)$. This is acceptable because the number of representatives is very low.

5.4 Asymptotic Performance

The LSH scheme described previously requires the computation of nkl hashes. The computational complexity of

each hash of a sample a is $O(|a|)$. Therefore, the overall complexity of the hashing step is $O(nkld)$, where $d = avg(|a|), \quad a \in A$, is the average number of features in a sample. After hashing, $|S|$ similarity functions must be computed.

The set S is an approximation of the true set of all near pairs T. We may, therefore, have false negatives ($T - S$), and false positives ($S - T$). We have $|S| \leq |T| + |S - T|$. Clearly, $|T| < nc$, where c is the maximum cluster size for the given threshold. Unfortunately, we cannot provide a theoretical bound for the fraction of false positives $|S - T|/|S|$ without making some assumptions on the distribution of the distances between pairs in A. However, in practice, the value is small (below 0.19 in our experiments). Therefore, the number of similarity computations is limited by the size of $|T|$ and the complexity of $O(nc)$. Since a single similarity computation is $O(d)$, computational complexity of this step is $O(ncd)$. Finally, the pairs in S need to be sorted to perform hierarchical clustering. This step is $O(nc\log(nc))$.

For large data sets, the cost of the similarity computations, which is $O(ncd)$, dominates. Note that while in practice nc is significantly smaller than n^2, the asymptotic performance has not improved. The reason is that c can still be $O(n)$ in the worst case. Consider, for instance, a trivial dataset where all n samples are identical. Clearly, for such a dataset we would have a single cluster of size n (and, therefore, $c = n$) for any t. More generally, if the threshold value t is too low, it may lead to most samples being concentrated in a few large clusters. However, for meaningful datasets and reasonable values of t, nc is significantly smaller than n^2. The performance gained by using LSH is therefore sufficient to allow us to cluster large, real-world malware data sets, as we will show in Section 6.3.

For extremely large datasets, on the other hand, more aggressive approximate clustering techniques may need to be employed (at the cost of some accuracy), such as the ones described in [27]. In [27], LSH is used to generate the set of approximate near pairs $|S|$, but there are no similarity computations. A pair $(a, b) \in S$ is not verified to be near by computing $similarity(a, b)$, but by using a faster approximate method that is based on the already computed hashes.

6 Evaluation

To verify the effectiveness of our approach, we used our system to cluster real-world malware data sets. In the next section, we discuss the quality of the generated clusters. Then, in Section 6.2, we compare our solution with previously-proposed clustering techniques [13, 31]. In Section 6.3, we present performance measurements of running our prototype on a very large data set. Finally, in Section 6.4, we discuss some examples of the clusters produced by our tool and of the insight they provide to the malware analyst.

6.1 Quality

Assessing the quality of the results that are produced by a clustering algorithm is an inherently difficult task. Obviously, it is possible to quantify the number of clusters, the average number of samples per cluster, or the relative sum of all pairwise distances for a cluster. Alternatively, one could randomly pick a few clusters and manually verify that the samples in these clusters are similar. The best option for demonstrating the correctness of a produced clustering, however, is to compare it with an existing reference clustering. Unfortunately, no such reference clustering exists for malware samples[2]. As a result, to verify that our clustering approach is meaningful, we first needed to create a reference clustering.

Reference Clustering. To create a reference clustering, we took the following approach: First, we obtained a set of 14,212 malware samples that were submitted to ANUBIS [1] in the period from October 27, 2007 to January 31, 2008. These samples were contributed by a number of security organizations and individuals, spanning a wide range of sources (such as web infections, honeypots, botnet monitoring, peer-to-peer systems, and URLs extracted from other malware analysis services). Then, we scanned each sample with six different anti-virus programs. For the initial reference clustering, we selected only those samples for which the majority of the anti-virus programs reported the same malware family (this required us to define a mapping between the different labels that are used by different anti-virus products). This resulted in a total of 2,658 samples. For each sample, we examined the corresponding ANUBIS [1] report and manually corrected classification problems.

Precision and Recall. To evaluate the quality of the clustering produced by our algorithm, we compared it to the reference clustering described above. To quantify the differences between the two clusterings, we introduce two metrics, precision and recall.

The goal of *precision* is to measure how well a clustering algorithm can distinguish between samples that are different. That is, precision captures how well a clustering algorithm assigns samples of different types to different clusters. Intuitively, we strive for results where each cluster contains only elements of one particular type. More formally, precision is defined as follows: Assume we have a reference clustering $T = T_1, T_2, .., T_t$ with t clusters and a clustering $C = C_1, C_2, .., C_c$ with c clusters (for a sample

[2] In fact, providing a reference clustering for a set of malware samples is a difficult problem by itself, mostly because it requires human expertise to compile such a clustering or confirm the correctness of existing results.

set $A = a_1, a_2, .., a_n$). For each $C_j \in C$, we calculate a cluster precision value as:

$$P_j = max(|C_j \cap T_1|, |C_j \cap T_2|, .., |C_j \cap T_t|)$$

The overall precision value is:

$$P = \frac{(P_1 + P_2 + .. + P_c)}{n}$$

In addition to precision, we use *recall* to measure how well a clustering algorithm recognizes similar samples. That is, recall captures how well an algorithm assigns samples of the same type to the same cluster. Clearly, we prefer a clustering where all elements of one type are assigned to the same cluster. We formally define recall as follows: Assume we have a reference clustering $T = T_1, T_2, .., T_t$ with t clusters and a clustering $C = C_1, C_2, .., C_c$ with c clusters. For each $T_j \in T$, we calculate a cluster recall value as:

$$R_j = max(|C_1 \cap T_j|, |C_2 \cap T_j|, .., |C_c \cap T_j|)$$

The overall recall value is:

$$R = \frac{(R_1 + R_2 + .. + R_r)}{n}$$

The primitive algorithm that creates a cluster for each sample achieves optimal precision, but the worst recall. The algorithm that combines all samples in a single cluster, instead, achieves optimal recall but the worst precision. In practice, an algorithm should provide both high precision and recall. That is, each cluster should contain all samples of one type, but no more.

Clustering Results. We have run our clustering algorithm on the reference set of 2,658 samples. For this run, we selected a similarity threshold of $t = 0.7$. The value of this threshold was determined based on our experience with initial experiments on a small malware sample set with less than a hundred programs. Later in this section, we discuss in more detail the considerations for selecting an appropriate threshold. Moreover, we will show that the algorithm is quite robust with regard to the choice of the concrete threshold value.

Our system produced 87 clusters, while the reference clustering consists of 84 clusters. For our results, we derived a precision of 0.984 and a recall of 0.930. This demonstrates that our approach has produced a clustering that is very close to the reference set. The excellent precision shows that the system was able to differentiate well between different malware classes. The recall shows that, in almost all cases, samples of the same class were grouped in the same cluster. A quantitative comparison to other clustering techniques is presented in the following Section 6.2. In Section 6.4, we discuss a number of interesting, qualitative observations about the clustering that our system produced.

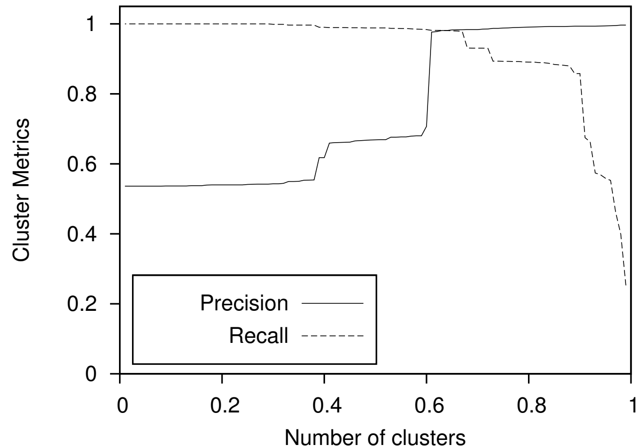

Figure 3. Precision and recall.

Threshold Selection. The value of the similarity threshold t determines how aggressively the clustering algorithm considers two different profiles as similar. Therefore, selecting a correct threshold often depends on the desired level of granularity of the clustering. For example, an analyst might be interested only in a rough partitioning of a set of malware samples into a few high-level categories (such as dialer, worm, or bot). Another analyst, instead, could be more interested in splitting a single malware family into different variants. In these cases, the first analyst would select a small t, while the second one would use a larger value for t.

For our experiments, we decided to use a threshold value such that our results would differentiate between malware families (that is, only similar variants of the same family should be clustered). As mentioned previously, a concrete value of $t = 0.7$ was selected, based on our experience with initial, small-scale experiments. However, the selection of the correct value of t is quite robust. Figure 3 shows how precision and recall vary with respect to different choices of t. One can see that a broad range of choices for $t \in [0.6, 0.9]$ yield good results for both precision and recall.

6.2 Comparative Evaluation

In the previous section, we have shown that our system has performed accurate clustering. However, we need to put these numbers into context with other approaches to be able to better assess the quality of our results. In this section, we present a comparative evaluation with the current state-of-the-art clustering approach, introduced by Bailey et al. [13]. Moreover, we analyze the impact of our behavioral abstraction and compare our clustering to one that is directly based on system call traces [31].

Bailey et al. [13] proposed a system for clustering malware based on the Normalized Compression Distance (NCD), using zlib-compression. NCD is based on the Kolmogorov complexity theory [33] and exploits the fact that similar data, when concatenated, compresses better than more differing data. Moreover, Bailey performs a coarse-grain abstraction from system calls and also uses profiles to represent malware behavior (we refer to these profiles as *Bailey-profiles* from now on). The difference to our approach is that Bailey-profiles contain only behavior in terms of non-transient state changes that a malware sample causes on the system (i.e., changes to the file-system, registry), as well as names of spawned processes and some basic information about network connections and scans. A detailed impression of the contents of Bailey-profiles can be gained from [12]. To evaluate Bailey's system on our reference data set, we adapted our dynamic analysis system to generate Bailey-profiles. Concerning NCD, we made use of the library provided by the Complearn-Toolkit [22].

A number of previous systems (e.g., [31]) based their behavioral profiles essentially on the raw system call traces. Thus, to evaluate the performance of such systems, and to obtain a baseline that shows the improvements due to generalized behavior profiles, we also performed clustering on the raw system call traces.

We used our reference clustering and the precision and recall metrics to directly compare the quality of the produced clusters for the different techniques. As an overall measure of clustering *quality*, we use the product of $precision * recall$. For each of the combinations of profile-types, similarity measures, and clustering methods presented in Table 2, we selected the threshold value which produces the highest quality score. In the clustering column, "exact" means that all $n * n/2$ distances between pairs of samples were computed, while "LSH" means that locality sensitive hashing was used. The last two rows show that the difference between exact and LSH-based clustering is minimal, demonstrating the effectiveness of LSH-based clustering as an approximation.

As can be seen in Table 2, the quality of our clustering approach (last two rows) outperforms the clustering proposed by Bailey et al. (first row). This is because our profiles represent the actual behavior of a malware sample in a more comprehensive and accurate way. For example, certain samples exhibit behavior that cannot be captured using Bailey-profiles. As a result, such profiles remain empty, or almost empty. Even more troublesome is the fact that Bailey's approach produces significantly worse results when using the Jaccard index as a similarity metric instead of NCD (second row). Unfortunately, a clustering algorithm based on NCD cannot take advantage of LSH to avoid computing all n^2 distances. Thus, a clustering approach that uses Bailey-profiles [13] either produces results that are significantly less precise than ours (by using the Jaccard index and LSH), or it does not scale to real-world datasets (when using NCD). When analyzing the results for raw system call traces (third row), the results are significantly worse than for the other two techniques. This is not surprising, since the traces contain far too much noise to effectively find similarities between even closely-related malware instances.

6.3 Performance

To demonstrate the scalability of our clustering algorithm, we ran our system on a set of 75,692 malware samples (obtained from the complete database of ANUBIS). We performed our experiments on a XEN virtual machine that was hosted on a PowerEdge 2950 server equipped with two Quad-Core Xeon 1.86 GHz CPUs and 8 GB of RAM. We allocated about 7GB RAM and one physical CPU to the XEN VM.

As shown in Table 3, our prototype implementation succeeded to cluster the set of 75,692 samples in 2 hours and 18 minutes. This time could be further reduced by exploiting the inherent parallelism: Both the LSH hashing and the distance calculation step can be easily performed in parallel. The memory requirements of our prototype never exceeded 3.7 GB of virtual memory. For each sample, we store a behavioral profile on disk, which consumes about 96 KB of disk space on average. To load the samples, the clustering algorithm had to read and process $6.9GB$ of behavioral profiles.

We ran the clustering algorithm with the same threshold value $t = 0.7$. The LSH algorithm computed a set S, our approximation of the set of near pairs, that contained 66,528,049 pairs. Only 57,024,374 pairs were indeed above the similarity threshold t, i.e., LSH hashing resulted in about 14% false positives. Nevertheless, employing LSH hashing allowed us to calculate only 66,528,049 instead of $(75,692^2)/2 = 2,864,639,432$ distances.

Algorithm Step	*Time*	*(Virt.) Mem. Used*
Loading the samples	58m	1.6 GB
l iterations of LSH hash.	1h 0m	3.6 GB
Distance calculation	16m	3.7 GB
Sorting all pairs	1m	3.7 GB
Hierarchical clustering	3m	3.7 GB
Total	2h 18m	3.7 GB

Table 3. Runtime for 75K samples.

Compared to previous work, our prototype shows significantly improved performance. To classify malware based on NCD as in Bailey et al. [13], all of the $n^2/2$ distances between the n samples need to be computed. Moreover, it is possible to derive from the run-time graphs presented in

Behavioral Profile	*Similarity Measure*	*Clustering*	*Optimal Threshold*	*Quality*	*Precision*	*Recall*
Bailey-profile [13]	NCD	exact	0.75	**0.916**	0.979	0.935
Bailey-profile [13]	Jaccard Index	exact	0.63	**0.801**	0.971	0.825
Syscalls [31]	Jaccard Index	exact	0.19	**0.656**	0.874	0.750
Our profile	Jaccard Index	exact	0.61	**0.959**	0.977	0.981
Our profile	Jaccard Index	LSH	0.60	**0.959**	0.979	0.980

Table 2. Comparative evaluation of different clustering methods.

their paper that a single distance calculation between two pairs takes about 1.25 milliseconds. As a result, the distance calculation step of their algorithm would require 995 hours (almost 6 weeks) to perform the necessary $75,692^2/2$ distance calculations. This is despite the fact that Bailey profiles are rather small (about 1KB on average). Applying our NCD implementation to the (much larger) behavioral profiles produced by our tool yields even more prohibitive computation times: a single NCD computation takes on average 43 milliseconds. Therefore, clustering 75,692 samples would take at least 6 months, even if the implementation were parallelized to run on 8 CPUs.

6.4 Qualitative Discussion of Clustering Results

In this section, we present a number of observations on the quality of our clustering techniques. First, we discuss the four largest clusters (with regard to the number of samples that they contain). These are Allaple.1 (1,289 samples), Allaple.2 (717 samples), DOS (179 samples), and GBDialer.j (106 samples). Together, they account for 86% of all samples.

Allaple.1 and Allaple.2 are two different variants of the Allaple worm [4]. Allaple is a polymorphic malware, which explains why there are so many different samples in each cluster. It also demonstrates the ability of our system to quickly dispose of polymorphic malware instances that appear different but exhibit the same behavior. Interestingly, we found that virus scanners were inconsistently assigning different *variant names* to samples in both clusters (recall that we only used the malware *family names* that the virus scanners reported to perform the initial reference clustering). However, closer manual analysis showed that our clustering correctly identified two different Allaple variants. While all of the samples in both clusters perform ICMP scans, the Allaple.2 variant is much more aggressive at immediately attempting to exploit the target systems using a wider variety of propagation vectors. For instance, almost all Allaple.2 samples perform DNS lookups for the addresses of hosts they have successfully scanned, and attempt to connect to TCP port 9988, which corresponds to the Windows remote administration service. On the other hand, in none of the samples in the Allaple.1 cluster is there any DNS or port 9988 activity. Furthermore, all samples in Allaple.1 make a copy of themselves to the file "C:\WINDOWS\system32\urdvxc.exe," while none of the samples in Allaple.2 do. Moreover, in the Allaple.1 cluster, we observe the following, interesting object dependences:

```
Section|C:\sample.exe->Network|TCP
File|C:\WINDOWS\system32\urdvxc.exe ->
  File|C:\(..)\Temporary Internet Files\
  \(..)\ccxebztz.exe
Random|Random Value Generator ->
  File|C:\(..)\Temporary Internet Files\
  \(..)\ccxebztz.exe
```

The first dependency indicates that the sample has succeeded in propagating itself over the network (to our nepenthes honeypot). Since our taint-system correctly handles memory-mapped files, we see that the malware propagates by reading a memory-mapped file and writing it to the network. The second and third dependences provide a strong indication that this is polymorphic malware, since data from the malware sample and from a random number generation API is written to the new file "ccxebztz.exe." This shows how system call dependences can provide valuable insight on malware behavior.

GBDialer.J is the biggest of several dialer clusters in our sample set. It is interesting that we were able to correctly group the samples in this cluster, because our analysis environment does not directly support the analysis of dialers. That is, there is no modem (emulation) present that would allow dialers to perform their main task. Nevertheless, the remaining behavior (such as startup actions and system modifications) was sufficiently characterizing to differentiate between the various dialer variants. This is not the case for the forth cluster, called "DOS." This cluster contains various DOS malware samples. The reason for not being able to distinguish between different DOS variants is that our analysis environment can only execute Windows PE executables. The Windows loader treats all non-Windows PE files as DOS executables, and attempts to execute them by emulating them in the `ntvdm.exe` process. This activity was recognized as similar behavior.

In addition to the four large clusters, there are several interesting, smaller clusters. For example, there is a cluster of

only two samples that are labeled as "Keylogger.Ghostbot" by the Kaspersky virus scanner. Our dynamic analysis discovered that this malware constantly checks for key presses using the Windows API function `GetKeyState`. The profile contains the following interesting comparisons:

```
cmp_val|registry|HKLM\SOFTWARE\MICROSOFT\
 \WINDOWS\CURRENTVERSION\RUN
 NtEnumerateValueKey-KeyValueInformation
 - PCCNTMON
```

This tells us that the malware looks for known anti-virus and firewall programs in the list of autostart registry values. Please note that the above is only an excerpt. In total, the profile lists 98 different program names that are compared against the result of `NtEnumerateValueKey`. We also have a cluster that consists of four samples that are recognized as "Mabezat" by the majority of virus scanners. Our behavioral profile shows that it is a file infector that searches for executable files on the local hard disk and infects them. This characteristic behavior was correctly identified and resulted in one cluster that precisely captured all four samples in the data set. We also discovered, with the help of control flow dependences, that the program is searching for different kinds of document files in the directory that Windows uses for temporarily storing data that is scheduled to be written to a CD. Again, we show only parts of the list of comparisons.

```
cmp_val|file|
C:\Documents and Settings\user\Local
 Settings\Application Data\Microsoft\
 \CD Burning\
 NtQueryDirectoryFile-FileInformation
  - .TXT
```

According to the virus description database of AVG [2], the malware program checks whether the current date is greater then 2012/10/16, and if so, starts encrypting user documents. Our system was only partly able to find this date check. Our profile is shown below:

```
cmp_val|time|System Time
 GetSystemTime-
 lpSystemTime.struct _SYSTEMTIME.wYear
 -2012
```

As one can see, the system correctly recognizes the fact that a comparison between the current year and the value 2012 takes place. As this comparison already fails, the rest of the date is not further checked. That is why we cannot determine the complete date. However, we are considering to improve our system with the ability to read the entire data structure from the main memory (in a fashion that is similar to our current approach for strings).

Of course, there are also malware programs for which our system did not produce the correct results. One common case is when a sample did not show any suspicious activity in our analysis environment. This could be because the malware program is damaged, or because it detects the presence of the analysis environment and exits prematurely. In any case, it underlines the dependence of our system on the quality of the behavioral profiles. One cluster in particular is composed of 25 samples which belong to 10 different clusters according to the reference clustering. Manual analysis reveals that these samples all crash, which causes the Dr. Watson debugger application to be executed, generate a crash report, and display a pop-up window asking the user permission to send the report to Microsoft. Clearly, this behavior is not specific to the malware family and it leads to misclassification.

7 Limitations and Future Work

Trace Dependence. As mentioned previously, a limitation of any dynamic malware analysis approach is that it is trace-dependent. Analysis results will be based only on the sample's behavior during one (or more) specific execution runs. Unfortunately, some of a malware's behavior may be triggered only under specific conditions. A simple example of trigger-based behavior is a time-bomb. That is, a malware that only exhibits its malicious behavior on a specific date. Another examples is a bot that only performs malicious actions when it receives specific commands through a command and control channel. Also, malware aimed at identity theft may only exhibit certain behavior when the user performs certain actions, such as logging into specific banking websites. Since we run malware samples automatically with no human interaction, such behavior will not occur in our traces.

Interestingly, our clustering may still succeed in grouping similar samples even when their most significant malicious behavior is not triggered, as is the case for the GBDialer.J cluster discussed in Section 6.4. The reason is that the behavioral features used for clustering encompass all malware behavior, not just malicious actions. Also, one could use techniques that explore multiple execution paths [37] to obtain a more comprehensive picture of the functionality of a program.

Evasion. Clearly, a malware author could manually modify a malware sample until its behavior is different enough from the original that the two are assigned to different clusters by our tool. We are not interested in this kind of labor-intensive, manual evasion. Instead, we consider an adversary who attempts to automatically produce an arbitrary number of mutations of a malware sample in such a way that all (or most) such mutations are assigned to different clus-

ters by our tool. To this end, a malware author could randomly mutate parts of the malware's behavior that are not essential to its functionality. An example would be the often arbitrary file names under which the malware copies itself on the file system. These could be replaced with random strings, hard-coded into each malware instance. Nonetheless, adding enough randomness to make each mutation different is not a simple task. A sample in our dataset has more than one thousand features on average, many of which represent behavior from inside system libraries that is only indirectly a consequence of the malware writer's intent. Also, since our tool discards features that are unique to a single malware instance, simple random variations would just lead to these features being discarded. In addition, we could add more aggressive generalization to our algorithm for extracting behavioral profiles. As an example, we could consider the name of any file created by the malware as irrelevant, and replace it with a special token (as we currently do for the names of temporary files).

Another issue is that dynamic data tainting of untrusted software is vulnerable to evasion. A malicious binary could inject fake data dependencies, using NOP-equivalent operations to taint clean data without modifying its value. Furthermore, it could hide data dependencies from our tool, using implicit flows to "clean" tainted data [19]. Unfortunately, there is no easy defense against such techniques. To address this issue, we would have to disable dynamic data tainting, sacrificing some of the system's accuracy.

8 Related Work

The recent advances in the field of automated malware analysis (e.g., [17, 25, 37, 45, 46]) have created a rising interest in the automatic grouping of the analysis results (and reports) that are created. For this purpose, researchers have proposed supervised as well as unsupervised machine learning techniques. Because it is crucial that these techniques can process a large number of samples, their scalability is one of the decisive properties.

At the core of every system that aims to find malware families is the notion of similarity. Therefore, these systems need to solve two problems. First, they need to find a suitable representation of a malware sample. Second, based on these representations, they need to compute a distance between two samples. In the literature, content-based and behavior-based comparison approaches have been proposed.

Content-Based Analysis. The first attempts to cluster malware samples were based on static analysis of the malware samples. In [26], the author proposes an automated virus classification system that works by first disassembling the binaries, and subsequently, comparing their basic code blocks. Other researchers have proposed to represent a malware program as a hex-dump of its code segment, building a classification system on top of this [30]. In [24], Dullien proposes a system for comparing executables based on their control flow graph.

All content-based analysis approaches share the problem that they need to disassemble the binary. This is often difficult or even impossible, given that malware is frequently obfuscated and packed. Also, it is possible to write semantically-equivalent programs that have large difference in their code. Thus, it is possible for malware authors to thwart content-based similarity calculations.

Behavior-Based Analysis. Recently, Holz et al. [28] presented a system that classifies unknown malware samples based on their behavior. A significant limitation is that the system requires supervised learning, using a virus scanner for labeling the training set. Lee et al. developed a system for classifying malware samples that relies on system calls for comparing executables [31]. The scalability of the technique is limited; the system required several hours to cluster a set of several hundred samples. Also, the tight focus on system calls implies that the collected profiles do not abstract the observed behavior.

The approach that is closest to ours was presented by Bailey et al. [13]. Their proposed system abstracts from system call traces and clusters samples that exhibit similar behavior. Unfortunately, Bailey's system does not scale well (it requires to compute $O(n^2)$ distances), and, compared to our system, their generated behavioral profiles lack important information that we can obtain via a fine-grained analysis and behavioral abstraction. This results in a clustering that is less accurate.

Leita et al. [32] suggest classifying malware based on the epsilon-gamma-pi-mu model. In this model, additional information on how the malware is originally installed on the target system is considered for classification. This can include information on the exploit and exploit payload used to install the malware dropper, and on the way the dropper in turn downloads and installs the malware. Since in [32] the malware itself is characterized by simply using anti-virus names, this approach is complementary to the one described in this paper.

Dynamic Data Tainting. Taint analysis is a technique that has been extensively used in the field of computer security. For example, it has been successfully applied to the detection of exploits that hijack the control flow of a program and, in some cases, automatic signature generation against detected threats [23, 39, 41]. Similar to our approach, there are systems that employ tainting for extracting characteristic information flows from malware binaries. Yin et al. [46] extended Qemu with data tainting to capture system-wide information flows. Recently, dynamic taint analysis has been also used for the automatic analysis of network protocols [18, 43].

9 Conclusion

In this paper, we propose a novel approach for clustering large collections of malware samples. The goal is to find a partitioning of a given set of malicious programs so that subsets exhibit similar behavior. Our system begins by analyzing each sample in a dynamic analysis environment that we have enhanced with taint tracking and additional network analysis. Then, we extract behavioral profiles by abstracting system calls, their dependences, and the network activities to a generalized representation consisting of OS objects and OS operations. These profiles serve as the input to our clustering algorithm, which requires less than a quadratic amount of distance computations. This is important to handle large data sets that are commonly encountered in the real world. Our experiments demonstrate that our techniques can accurately recognize malicious code that behaves in a similar fashion. Moreover, our results show that we can cluster more than 75 thousand samples in less than three hours.

Acknowledgments

This work has been supported by the European Commission through project FP7-ICT-216026-WOMBAT, by FIT-IT through the Pathfinder project, by FWF through the Web-Defense project (No. P18764) and by Secure Business Austria. We would like to thank Luca Foschini for his assistance and for providing his expertise in the area of large-scale data clustering.

References

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

[2] AVG Virus Database - Mabezat. http://www.avg.com/virbase?nam=win32/mabezat, 2008.

[3] CWSandbox. http://www.cwsandbox.org/, 2008.

[4] F-Secure Malware Information Pages - Allaple.A. http://www.f-secure.com/v-descs/allaple_a.shtml, 2008.

[5] MWCollect. http://www.mwcollect.org/, 2008.

[6] Norman Sandbox. http://www.norman.com/microsites/nsic/, 2008.

[7] Shadowserver. http://shadowserver.org/wiki/, 2008.

[8] ThreatExpert. http://www.threatexpert.com/, 2008.

[9] Virus Total. http://www.virustotal.com/, 2008.

[10] D. Arthur and S. Vassilvitskii. How slow is the k-means method? In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, New York, NY, USA, 2006. ACM.

[11] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. C. Freiling. The nepenthes platform: An efficient approach to collect malware. In D. Zamboni and C. Kruegel, editors, *RAID*, volume 4219 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2006.

[12] M. Bailey. Malware clustering results. http://www.eecs.umich.edu/~mibailey/malware/, 2008.

[13] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, September 2007.

[14] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[15] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference*, 2005.

[16] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997.

[17] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al.*, 2007.

[18] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.

[19] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[20] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002.

[21] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 5–14, New York, NY, USA, 2007. ACM.

[22] R. Cilibrasi and P. Vitányi. Complearn version 1.15. http://www.complearn.org/, 2008.

[23] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[24] T. Dullien and R. Rolles. Graph-based comparison of Executable Objects. In *In Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2005.

[25] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.

[26] M. Gheorghescu. An Automated Virus Classification System. In *Virus Bulletin conference*, 2005.

[27] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[28] T. Holz, C. Willems, K. Rieck, P. Duessel, and P. Laskov. Learning and Classification of Malware Behavior. In *Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 08)*, June 2008.

[29] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, pages 604–613, 1998.

[30] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, 2006.

[31] T. Lee and J. J. Mody. Behavioral Classification. In *EICAR Conference*, 2006.

[32] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *EDCC 2008, 7th European Dependable Computing Conference, May 7-9, 2008, Kaunas, Lituania*, 2008.

[33] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, second edition, 1997.

[34] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM.

[35] L.Kaufman and P. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. New York: John Wiley & Sons, 1990.

[36] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[37] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 231–245, 2007.

[38] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *ACSAC*, pages 421–430. IEEE Computer Society, 2007.

[39] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[40] V. Paxson. Bro: a system for detecting network intruders in real-time. *Comput. Networks*, 31(23-24):2435–2463, 1999.

[41] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[42] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[43] G. Wondracek, P. Milani Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[44] T. Yetiser. Polymorphic Viruses - Implementation, Detection, and Protection. `http://vx.netlux.org/lib/ayt01.html`, 1993.

[45] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

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